Configuration for MicroProfile

Mark Struberg, Emily Jiang, John D. Ament

1.4-RC3, January 30, 2020
# Table of Contents

MicroProfile Config .................................................. 2
Architecture ............................................................. 3
    Rationale ............................................................ 3
Config Usage Examples .................................................. 4
    Simple Programmatic Example ................................... 4
    Simple Dependency Injection Example ........................ 4
Accessing or Creating a certain Configuration .................... 6
ConfigSources .......................................................... 8
    ConfigSource Ordering ........................................... 8
    Manually defining the Ordinal of a built-in ConfigSource ... 8
    Default ConfigSources ........................................... 8
        Environment Variables Mapping Rules ....................... 8
    Custom ConfigSources .......................................... 9
    Custom ConfigSources via ConfigSourceProvider ............. 10
Cleaning up a ConfigSource .......................................... 10
ConfigSource and Mutable Data ...................................... 10
Converter ............................................................... 11
    Built-in Converters .............................................. 11
    Adding custom Converters ...................................... 11
    Array Converters ................................................ 12
        Programmatic lookup ......................................... 12
        Injection model ............................................... 12
    Automatic Converters .......................................... 12
    Cleaning up a Converter ....................................... 12
Release Notes for MicroProfile Config 1.1 ......................... 13
    API/SPI Changes ................................................ 13
    Functional Changes ............................................. 13
    Specification Changes ........................................... 13
Release Notes for MicroProfile Config 1.2 ......................... 14
    API/SPI Changes ................................................ 14
    Functional Changes ............................................. 14
    Specification Changes ........................................... 14
    Other Changes .................................................. 14
Release Notes for MicroProfile Config 1.3 ......................... 15
    API/SPI Changes ................................................ 15
    Functional Changes ............................................. 15
    Specification Changes ........................................... 15
    Other Changes .................................................. 15
Specification: Configuration for MicroProfile

Version: 1.4-RC3

Status: Draft

Release: January 30, 2020

Copyright (c) 2016-2018 Contributors to the Eclipse Foundation

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
MicroProfile Config
Architecture

This specification defines an easy to use and flexible system for application configuration. It also defines ways to extend the configuration mechanism itself via a SPI (Service Provider Interface) in a portable fashion.

Rationale

Released binaries often contain functionality which needs to behave slightly differently depending on the deployment. This might be the port numbers and URLs of REST endpoints to talk to (e.g. depending on the customer for whom a WAR is deployed). Or it might even be whole features which need to be switched on and off depending on the installation. All this must be possible without the need to re-package the whole application binary.

MicroProfile Config provides a way to achieve this goal by aggregating configuration from many different ConfigSources and presents a single merged view to the user. This allows the application to bundle default configuration within the application. It also allows to override the defaults from outside, e.g. via an environment variable a Java system property or via a container like Docker. MicroProfile Config also allows to implement and register own configuration sources in a portable way, e.g. for reading configuration values from a shared database in an application cluster.

Internally, the core MicroProfile Config mechanism is purely String/String based. Type-safety is intentionally only provided on top of that by using the proper Converters before handing the value out to the caller.

The configuration key might use dot-separated blocks to prevent name conflicts. This is similar to Java package namespacing:

```text
com.acme.myproject.someserver.url = http://some.server/some/endpoint
com.acme.myproject.someserver.port = 9085
com.acme.myproject.someserver.active = true
com.acme.other_stuff.name = Karl
com.acme.myproject.notify.onerror=karl@mycompany,sue@mcompany
some.library.own.config=some value
```

**TIP** while the above example is in the java property file syntax the actual content could also e.g. be read from a database.
Config Usage Examples

An application can obtain its configuration programmatically via the `ConfigProvider`. In CDI enabled beans it can also get injected via `@Inject Config`. An application can then access its configured values via this `Config` instance.

Simple Programmatic Example

```java
public class ConfigUsageSample {
    public void useTheConfig() {
        // get access to the Config instance
        Config config = ConfigProvider.getConfig();

        String serverUrl = config.getValue("acme.myprj.some.url", String.class);
        callToServer(serverUrl);
    }
}
```

If you need to access a different server then you can e.g. change the configuration via a Java `-D` system property:

```
$> java -Dacme.myprj.some.url=http://other.server/other/endpoint -jar some.jar
```

Note that this is only one example how to possibly configure your application. Another example is to register Custom ConfigSources to e.g. pick up values from a database table, etc.

If a config value is a comma(,) separated string, this value can be automatically converted to a multiple element array with \ as the escape character. When specifying the property `myPets=dog,cat,dog\,cat` in a config source, the following code snippet can be used to obtain an array.

```java
String[] myPets = config.getValue("myPets", String[].class);
//myPets = {"dog", "cat", "dog,cat"}
```

Simple Dependency Injection Example

MicroProfile Config also provides ways to inject configured values into your beans using the `@Inject` and the `@ConfigProperty` qualifier. The `@Inject` annotation declares an injection point. When using this on a passivation capable bean, refer to CDI Specification for more details on how to make the injection point to be passivation capable.
`@ApplicationScoped
public class InjectedConfigUsageSample {

    @Inject
    private Config config;

    // The property myprj.some.url must exist in one of the config sources, otherwise a
    // DeploymentException will be thrown.
    @Inject
    @ConfigProperty(name = "myprj.some.url")
    private String someUrl;

    // The following code injects an Optional value of myprj.some.port property.
    // Contrary to natively injecting the configured value this will not lead to a
    // DeploymentException if the configured value is missing.
    @Inject
    @ConfigProperty(name = "myprj.some.port")
    private Optional<Integer> somePort;

    // Injects a Provider for the value of myprj.some.dynamic.timeout property to
    // resolve the property dynamically. Each invocation to Provider#get() will
    // resolve the latest value from underlying Config.
    // The existence of configured values will get checked during startup.
    // Instances of Provider<T> are guaranteed to be Serializable.
    @Inject
    @ConfigProperty(name = "myprj.some.dynamic.timeout", defaultValue = "100")
    private javax.inject.Provider<Long> timeout;

    // The following code injects an Array, List or Set for the 'myPets' property,
    // where its value is a comma separated value (myPets=dog,cat,dog\,cat)
    @Inject @ConfigProperty(name = "myPets")
    private String[] myArrayPets;
    @Inject @ConfigProperty(name = "myPets")
    private List<String> myListPets;
    @Inject @ConfigProperty(name = "myPets")
    private Set<String> mySetPets;
}
Accessing or Creating a certain Configuration

For using MicroProfile Config in a programmatic way the ConfigProvider class is the central point to access a configuration. It allows access to different configurations (represented by a Config instance) based on the application in which it is used. The ConfigProvider internally delegates through to the ConfigProviderResolver which contains more low-level functionality.

There are 4 different ways to create a Config instance:

- In CDI managed components, a user can use @Inject to access the current application configuration. The default and the auto discovered ConfigSources will be gathered to form a configuration. The default and the auto discovered Converters will be gathered to form a configuration. Injected instance of Config should behave the same as the one retrieved by ConfigProvider.getConfig(). Injected config property values should be the same as if retrieved from an injected Config instance via Config.getValue().

- A factory method ConfigProvider#getConfig() to create a Config object based on automatically picked up ConfigSources of the Application identified by the current Thread Context ClassLoader classpath. The default and the auto discovered Converters will be gathered to form a configuration. Subsequent calls to this method for a certain Application will return the same Config instance.

- A factory method ConfigProvider#getConfig(ClassLoader forClassLoader) to create a Config object based on automatically picked up ConfigSources of the Application identified by the given ClassLoader. The default and the auto discovered Converters will be gathered to form a configuration. This can be used if the Thread Context ClassLoader does not represent the correct layer. E.g. if you need the Config for a class in a shared EAR lib folder. Subsequent calls to this method for a certain Application will return the same Config instance.

- A factory method ConfigProviderResolver#getBuilder() to create a ConfigBuilder object. The builder has no config sources. Only the default converters are added. The ConfigBuilder object can be filled manually via methods like ConfigBuilder#withSources(ConfigSources… sources). This configuration instance will by default not be shared by the ConfigProvider. This method is intended be used if a IoC container or any other external Factory can be used to give access to a manually created shared Config.

  ◦ Create a builder:

  ```java
  ConfigProviderResolver resolver = ConfigProviderResolver.instance();
  ConfigBuilder builder = resolver.getBuilder();
  ```

  ◦ Add config sources and build:

  ```java
  Config config =
  builder.addDefaultSources().withSources(mySource).withConverters(myConverter).build;
  ```
Manage the lifecycle of the config

```java
resolver.registerConfig(config, classloader);
resolver.releaseConfig(config);
```

The `Config` object created via builder pattern can be managed as follows:

- A factory method `ConfigProviderResolver#registerConfig(Config config, ClassLoader classloader)` can be used to register a `Config` within the application. This configuration instance will be shared by `ConfigProvider#getConfig()`. Any subsequent call to `ConfigProvider#getConfig()` will return the registered `Config` instance for this application.

- A factory method `ConfigProviderResolver#releaseConfig(Config config)` to release the `Config` instance. This will unbind the current `Config` from the application. The `ConfigSources` that implement the `java.io.Closeable` interface will be properly destroyed. The `Converters` that implement the `java.io.Closeable` interface will be properly destroyed. Any subsequent call to `ConfigProvider#getConfig()` or `ConfigProvider#getConfig(ClassLoader forClassLoader)` will result in a new `Config` instance.

All methods in the `ConfigProvider`, `ConfigProviderResolver` and `Config` implementations are thread safe and reentrant.

The `Config` instances created via CDI are `Serializable`.

If a `Config` instance is created via `@Inject Config` or `ConfigProvider#getConfig()` or via the builder pattern but later called `ConfigProviderResolver#registerConfig(Config config, Classloader classloader)`, the `Config` instance will be released when the application is closed.
ConfigSources

A ConfigSource is exactly what its name says: a source for configured values. The Config uses all configured implementations of ConfigSource to look up the property in question.

ConfigSource Ordering

Each ConfigSource has a specified ordinal, which is used to determine the importance of the values taken from the associated ConfigSource. A higher ordinal means that the values taken from this ConfigSource will override values from lower-priority ConfigSources. This allows a configuration to be customized from outside a binary, assuming that external ConfigSource s have higher ordinal values than the ones whose values originate within the release binaries.

It can also be used to implement a drop-in configuration approach. Simply create a jar containing a ConfigSource with a higher ordinal and override configuration values in it. If the jar is present on the classpath then it will override configuration values from ConfigSources with lower ordinal values.

Manually defining the Ordinal of a built-in ConfigSource

Note that a special property config_ordinal can be set within any built-in ConfigSource implementation. The default implementation of getOrdinal() will attempt to read this value. If found and a valid integer, the value will be used. Otherwise the respective default value will be used.

```
config_ordinal = 120
com.acme.myproject.someserver.url = http://more_important.server/some/endpoint
```

Default ConfigSources

A MicroProfile Config implementation must provide ConfigSources for the following data out of the box:

- System properties (default ordinal=400).
- Environment variables (default ordinal=300).
- A ConfigSource for each property file META-INF/microprofile-config.properties found on the classpath. (default ordinal = 100).

Environment Variables Mapping Rules

Some operating systems allow only alphabetic characters or an underscore, _, in environment variables. Other characters such as ., /, etc may be disallowed. In order to set a value for a config property that has a name containing such disallowed characters from an environment variable, the
following rules are used.

The ConfigSource for the environment variables searches three environment variables for a given property name (e.g. com.ACME.size):

1. Exact match (i.e. com.ACME.size)
2. Replace each character that is neither alphanumeric nor _ with _ (i.e. com_ACME_size)
3. Replace each character that is neither alphanumeric nor _ with_; then convert the name to upper case (i.e. COM_ACME_SIZE)

The first environment variable that is found is returned by this ConfigSource.

Custom ConfigSources

ConfigSources are discovered using the java.util.ServiceLoader mechanism.

To add a custom ConfigSource, implement the interface org.eclipse.microprofile.config.spi.ConfigSource.

```java
public class CustomDbConfigSource implements ConfigSource {

    @Override
    public int getOrdinal() {
        return 112;
    }

    @Override
    public Set<String> getPropertyNames() {
        return readPropertyNames();
    }

    @Override
    public Map<String, String> getProperties() {
        return readPropertiesFromDb();
    }

    @Override
    public String getValue(String key) {
        return readPropertyFromDb(key);
    }

    @Override
    public String getName() {
        return "customDbConfig";
    }
}
```
Then register your implementation in a resource file `/META-INF/services/org.eclipse.microprofile.config.spi.ConfigSource` by including the fully-qualified class name of the custom implementation in the file.

**Custom ConfigSources via ConfigSourceProvider**

If you need dynamic ConfigSources you can also register a ConfigSourceProvider in a similar manner. This is useful if you need to dynamically pick up multiple ConfigSources of the same kind; for example, to pick up all `myproject.properties` resources from all the JARs in your classpath.

A custom ConfigSourceProvider must implement the interface `org.eclipse.microprofile.config.spi.ConfigSourceProvider`. Register your implementation in a resource file `/META-INF/services/org.eclipse.microprofile.config.spi.ConfigSourceProvider` by including the fully-qualified class name of the custom implementation/s in the file.

An example which registers all YAML files with the name `exampleconfig.yaml`:

```java
public class ExampleYamlConfigSourceProvider implements org.eclipse.microprofile.config.spi.ConfigSourceProvider {
    @Override
    public List<ConfigSource> getConfigSources(ClassLoader forClassLoader) {
        List<ConfigSource> configSources = new ArrayList<>();

        Enumeration<URL> yamlFiles = forClassLoader.getResources("sampleconfig.yaml");
        while (yamlFiles.hasMoreElements()) {
            configSources.add(new SampleYamlConfigSource(yamlFiles.nextElement()));
        }
        return configSources;
    }
}
```

Please note that a single ConfigSource should be either registered directly or via a ConfigSourceProvider, but never both ways.

**Cleaning up a ConfigSource**

If a ConfigSource implements the `java.lang.AutoCloseable` interface then the close() method will be called when the underlying Config is being released.

**ConfigSource and Mutable Data**

A Config instance provides no caching but iterates over all ConfigSources for each getValue(String) operation. A ConfigSource is allowed to cache the underlying values itself.
Converter

For providing type-safe configuration we need to convert from the configured Strings into target types. This happens by providing Converters in the Config.

Built-in Converters

The following Converters are provided by MicroProfile Config by default:

- boolean and java.lang.Boolean, values for true (case insensitive) "true", "1", "YES", "Y", "ON". Any other value will be interpreted as false
- byte and java.lang.Byte
- short and java.lang.Short
- int and java.lang.Integer
- long and java.lang.Long
- float and java.lang.Float, a dot '.' is used to separate the fractional digits
- double and java.lang.Double, a dot '.' is used to separate the fractional digits
- char and java.lang.Character
- java.lang.Class based on the result of Class.forName

All built-in Converters have the @Priority of 1.

Adding custom Converters

A custom Converter must implement the generic interface org.eclipse.microprofile.config.spi.Converter. The Type parameter of the interface is the target type the String is converted to. You have to register your implementation in a file /META-INF/services/org.eclipse.microprofile.config.spi.Converter with the fully qualified class name of the custom implementation.

A custom Converter can define a priority with the @javax.annotation.Priority annotation. If a Priority annotation isn't applied, a default priority of 100 is assumed. The Config will use the Converter with the highest Priority for each target type.

A custom Converter for a target type of any of the built-in Converters will overwrite the default Converter.

Converters can be added to the ConfigBuilder programmatically via ConfigBuilder#withConverters(Converter<?>... converters) where the type of the converters can be obtained via reflection. However, this is not possible for a lambda converter. In this case, use the method ConfigBuilder#withConverter(Class<T> type, int priority, Converter<T> converter).
Array Converters

For the built-in converters and custom converters, the corresponding Array converters are provided by default. The delimiter for the config value is ",". The escape character is "\". e.g. With this config `myPets=dog,cat,\dog,\cat`, the values as an array will be `{"dog", "cat", "\dog,\cat"}`.

Programmatic lookup

Array as a class type is supported in the programmatic lookup.

```java
String[] myPets = config.getValue("myPets", String[].class);
```

myPets will be "dog", "cat", "\dog,\cat" as an array

Injection model

For the property injection, Array, List and Set are supported.

```java
@Inject @ConfigProperty(name="myPets") String[] myPetsArray;
@Inject @ConfigProperty(name="myPets") List<String> myPetsList;
@Inject @ConfigProperty(name="myPets") Set<String> myPetsSet;
```

myPets will be "dog", "cat", "\dog,\cat" as an array, List or Set.

Automatic Converters

If no built-in nor custom Converter exists for a requested Type `T`, an implicit Converter is automatically provided if the following conditions are met:

- The target type `T` has a `public static T of(String)` method, or
- The target type `T` has a `public static T valueOf(String)` method, or
- The target type `T` has a `public static T parse(CharSequence)` method, or
- The target type `T` has a public Constructor with a `String` parameter

Cleaning up a Converter

If a Converter implements the `java.lang.AutoCloseable` interface then the `close()` method will be called when the underlying Config is being released.
Release Notes for MicroProfile Config 1.1

The following changes occurred in the 1.1 release, compared to 1.0

A full list of changes may be found on the MicroProfile Config 1.1 Milestone

API/SPI Changes

- The ConfigSource SPI has been extended with a default method that returns the property names for a given ConfigSource (#178)

Functional Changes

- Implementations must now include a URL Converter, of @Priority(1) (#181)
- The format of the default property name for an injection point using @ConfigProperty has been changed to no longer lower case the first letter of the class. Implementations may still support this behavior. Instead, MicroProfile Config 1.1 requires the actual class name to be used. (#233)
- Implementations must now support primitive types, in addition to the already specified primitive type wrappers (#204)

Specification Changes

- Clarified what it means for a value to be present (#216)
Release Notes for MicroProfile Config 1.2

The following changes occurred in the 1.2 release, compared to 1.1

A full list of changes may be found on the MicroProfile Config 1.2 Milestone

API/SPI Changes

- The ConfigBuilder SPI has been extended with a method that allows for a converter with the specified class type to be registered (#205). This change removes the limitation, which was unable to add a lambda converter, from the previous releases.

Functional Changes

- Implementations must now support the array converter (#259). For the array converter, the programmatic lookup of a property (e.g. `config.getValue(myProp, String[].class)`) must support the return type of the array. For the injection lookup, an Array, List or Set must be supported as well (e.g. `@Inject @ConfigProperty(name="myProp") private List<String> propValue;`).

- Implementations must also support the common sense converters (#269) where there is no corresponding type of converters provided for a given class. The implementation must use the class's constructor with a single string parameter, then try `valueOf(String)` followed by `parse(CharSequence)`.

- Implementations must also support Class converter (#267)

Specification Changes

- Specification changes to document (#205), (#259), (#269) (#267)

Other Changes

The API bundle can work with either CDI 1.2 or CDI 2.0 in OSGi environment (#249).

A tck test was added to ensure the search path of `microprofile-config.properties` for a war archive is `WEB-INF\classes\META-INF` (#268)
Release Notes for MicroProfile Config 1.3

The following changes occurred in the 1.3 release, compared to 1.2

A full list of changes may be found on the MicroProfile Config 1.3 Milestone

API/SPI Changes

No API/SPI changes.

Functional Changes

• The implicit (common sense) converters have been improved and some of the built-in converters are removed from the spec as they are covered by implicit converters. The method invocation sequence on implicit converters are further improved (#325).

• Implementations must also support the mapping of a config property to the corresponding environment variable (#264)

Specification Changes

• Specification changes to document (#348), (#325), (#264)

Other Changes

More CTS were added:

• Assert URI will be converted (#322)

• Testing injecting an Optional<String> that has no config value (#336).

• Built-in converters are automatically added to the injected config (#348)

Java2 security related change (#343)
Release Notes for MicroProfile Config 1.4

A full list of changes delivered in the 1.4 release can be found at MicroProfile Config 1.4 Milestone.

API/SPI Changes

- Prevent incorrect caching of ConfigProviderResolver (#265)
- ConfigProviderResolver classloading issues (#450) (#390)
- Converter extends Serializable (#473)

Spec Changes

- Change the priority of implicit converters (#383)
- Clarify if @ConfigProperty injected values are bean passivating enabled (#404)
- Add built-in converters for byte, short and char (#386)

Other Changes

- Exclude EL api transitive dependency (#440)
- Other minor spec wording or JavaDoc updates