Configuration for MicroProfile

Mark Struberg, Emily Jiang, John D. Ament, Roberto Cortez, Jan Bernitt

2.0-RC10, November 09, 2020

Table of Contents

Specification: Configuration for MicroProfile

Version: 2.0-RC10

Status: Draft

Release: November 09, 2020

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 or erase the property by simply specifying the property name without providing a value or an empty string as the value, 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:

```
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

```
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);

        // or
        ConfigValue configServerUrl = config.getConfigValue("acme.myprj.some.url");
        callToServer(configServerUrl.getValue());
    }
}
```

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.

```
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 with a non-empty value, otherwise a
    //DeploymentException will be thrown.
    @Inject
    @ConfigProperty(name="myprj.some.url")
    private String someUrl;

    // You can also inject a configuration using the ConfigValue metadata object. The
    // configured value will not lead to a DeploymentException if the value is
missing.
    // A default value can also be specified like any other configuration.
    @Inject
    @ConfigProperty(name="myprj.another.url")
    private ConfigValue anotherUrl;

    //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 value is missing.
    @Inject
    @ConfigProperty(name="myprj.some.port")
    private Optional<Integer> somePort;

    // You can also use the specialized Optional classes like OptionalInt,
    // OptionalDouble, or OptionalLong to perform the injection. The configured value
    // will not lead to a DeploymentException if the value is missing.
    @Inject
    @ConfigProperty(name="myprj.some.port")
    private OptionalInt 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 start-up.
    //Instances of Provider<T> are guaranteed to be Serializable.
    @Inject
    @ConfigProperty(name="myprj.some.dynamic.timeout", defaultValue="100")
    private javax.inject.Provider<Long> timeout;

    //Injects a Supplier for the value of myprj.some.supplier.timeout property to
    //resolve the property dynamically. Each invocation to Supplier#get() will
    //resolve the latest value from underlying Config.
    @Inject
    @ConfigProperty(name="myprj.some.supplier.timeout", defaultValue="100")
    private java.util.function.Supplier<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;
}
```

Config value conversion rules

The table below defines the conversion rules, including some special edge case scenarios.

Input String	Output type	Method	behaviour
"foo,bar"	String	getValue	"foo,bar"
"foo,bar"	String[]	getValue	{"foo", "bar"}
"foo,bar"	String	getOptionalValue	Optional.of("foo,bar")
"foo,bar"	String[]	getOptionalValue	Optional.of({"foo","bar"})
"foo,bar"	String	getOptionalValues	Optional.of("foo", "bar")
"foo,"	String	getValue	"foo,"
"foo,"	String[]	getValue	{"foo"}
"foo,"	String	getOptionalValue	Optional.of("foo,")
"foo,"	String[]	getOptionalValue	Optional.of({"foo"})
"foo,"	String	getOptionalValues	Optional.of("foo")
",bar"	String	getValue	",bar"
",bar"	String[]	getValue	{"bar"}
",bar"	String	getOptionalValue	Optional.of(",bar")
",bar"	String[]	getOptionalValue	Optional.of({"bar"})
",bar"	String	getOptionalValues	Optional.of("bar")
" " (space)	String	getValue	" "

Input String	Output type	Method	behaviour
" "(space)	String[]	getValue	{" "}
" "(space)	String	getOptionalValue	Optional.of(" ")
" "(space)	String[]	getOptionalValue	Optional.of({" "})
" "(space)	String	getOptionalValues	Optional.of(" ")
missing	String	getValue	throws `NoSuchElementException`
missing	String[]	getValue	throws `NoSuchElementException`
missing	String	getOptionalValue	Optional.empty()
missing	String[]	getOptionalValue	Optional.empty()
missing	String	getOptionalValues	Optional.empty()
""	String	getValue	throws `NoSuchElementException`
""	String[]	getValue	throws `NoSuchElementException`
""	String	getOptionalValue	Optional.empty()
""	String[]	getOptionalValue	Optional.empty()
""	String	getOptionalValues	Optional.empty()
","	String	getValue	","
","	String[]	getValue	throws `NoSuchElementException`
","	String	getOptionalValue	Optional.of(",")
","	String[]	getOptionalValue	Optional.empty()
","	String	getOptionalValues	Optional.empty()
"\,"	String	getValue	"\,"
"\,"	String[]	getValue	{","}

Input String	Output type	Method	behaviour
"\,"	String	getOptionalValue	Optional.of("\,")
"\,"	String[]	getOptionalValue	Optional.of({","})
"\,"	String	getOptionalValues	Optional.of(List.of(","))
",,"	String	getValue	",,"
",,"	String[]	getValue	throws `NoSuchElementException`
",,"	String	getOptionalValue	Optional.of(",,")
",,"	String[]	getOptionalValue	Optional.empty()
",,"	String	getOptionalValues	Optional.empty()

Remove config properties

Sometimes, there is a need to remove a property. This can be done by setting an empty value or a value causing the corresponding converter returning `null` in a config source. When injecting a property that has been deleted, `DeploymentException` will be thrown unless the return type is `Optional`.

Aggregate related properties into a CDI bean

When injecting a number of related configuration properties, it can be tedious to repeat the statement of `ConfigProperty` in scatter places. Since they are related, it makes more sense to aggregate them into a single property class.

MicroProfile Config provides a way to look up a number of configuration properties starting with the same prefix using the `@ConfigProperties` annotation, e.g. `ConfigProperties(prefix="myPrefix")`. When annotating a class with `@ConfigProperties` or `@ConfigProperties(prefix="myPrefix")`, any of its fields, regardless of the visibility, maps to a configuration property via the following mapping rules.

- If the `prefix` is present, the field `x` maps to the configuration property `<prefix>.x`.
- If the `prefix` is absent, the field `x` maps to the property name `x`.

If the field name `x` needs to be different from the config property name `y`, use `@ConfigProperty(name="y")` to perform the transformation. If the prefix is present, the field `x` maps to the configuration property `<prefix>.y`, otherwise `y`.

Considering the following config sources:

```
config_ordinal = 120
server.host = localhost
server.port=9080
server.endpoint=query
server.old.location=London
```

```
config_ordinal = 150
client.host = myHost
client.port=9081
client.endpoint=shelf
client.old.location=Dublin
host = anotherHost
port=9082
endpoint=book
old.location=Berlin
```

In order to retrieve the above properties in a single property class, you can use the `@ConfigProperties` annotation with a prefix.

```
@ConfigProperties(prefix="server")
@Dependent
public class Details {
    public String host; // the value of the configuration property server.host
    public int port;   // the value of the configuration property server.port
    private String endpoint; //the value of the configuration property server.endpoint
    public @ConfigProperty(name="old.location")
    String location; //the value of the configuration property server.old.location
    public String getEndpoint() {
        return endpoint;
    }
}
```

You can then use one of the following to retrieve the properties.

Programmatic lookup of the bean annotated with `@ConfigProperites`

Since the class with `@ConfigProperties` is a CDI bean, you can use the programmatic lookup provided by CDI, e.g.

```
Details details = CDI.current().select(Details.class, ConfigProperties.Literal.NO
:PREFIX).get();
```

Inject the bean annotated with `@ConfigProperties`

```
@Inject
@ConfigProperties
Details serverDetails;
```

The `serverDetails` will contain the following info, as the prefix is `server`:

```
serverDetails.host -> server.host -> localhost
serverDetails.port -> server.port -> 9080
serverDetails.endpoint -> server.endpoint -> query
serverDetails.getLocation() -> server.old.location -> London
```

Specify the prefix attribute on the annotation `@ConfigProperties` when injecting the bean.

In this case, the prefix associated with `@ConfigProperties` on this injection point overrides the prefix specified on the bean class.

```
@Inject
@ConfigProperties(prefix="client")
Details clientDetails;
```

The prefix `client` overrides the prefix `server` on the `ServerDetails` bean. Therefore, this will retrieve the following properties.

```
clientDetails.host -> client.host -> myHost
clientDetails.port -> client.port -> 9081
clientDetails.endpoint -> client.endpoint -> shelf
clientDetails.getLocation() -> client.old.location -> Dublin
```

If `@ConfigProperties` has no associated prefix at the injection point, it defaults to the prefix set in the `Details` class, `server`.

```
@Inject
@ConfigProperties
Details details;
```

Therefore, this will retrieve the following properties.

```
serverDetails.host -> server.host -> localhost
serverDetails.port -> server.port -> 9080
serverDetails.endpoint -> server.endpoint -> query
serverDetails.getLocation() -> server.old.location -> London
```

If `@ConfigProperties` specifies an empty prefix at the injection point:

```
@Inject
@ConfigProperties(prefix = "")
Details details;
```

It overrides the prefix set on the bean class `server` with an empty string `""`

```
details.host -> host -> anotherHost
details.port -> port -> 9082
details.endpoint -> endpoint -> book
details.getLocation() -> old.location -> Berlin
```

ConfigProperties bean class validation

The configuration properties class should contain a zero-arg constructor. Otherwise, the behaviour is unspecified. When performing property lookup, a `DeploymentException` will be thrown for the following scenarios:

1. The property is missing and neither default value nor the property return type is optional. Use one of the following to fix the problem.
    - Define a value for the property
    - Supply a default value when defining the field.
    - Use `@ConfigProperty` to provide a default value.
    - Use `Optional<T>` or `OptionalInt`, `OptionalDouble`, `OptionalLong` as the type.
2. The property value cannot be converted to the specified type

If any of the property cannot be found and there is neither default value nor property is not optional, `java.util.NoSuchElementException` will be thrown. In order to avoid this, you can supply a default value when defining the field. Alternatively, you can use `@ConfigProperty` to provide a default value. You can also use `Optional<T>` or OptionalInt, OptionalDouble, OptionalLong as the type. If any property values cannot be converted to the specified type, `java.lang.IllegalArgumentException` will be thrown.

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 an IoC container or any other external Factory can be used to give access to a manually created shared `Config`.
  - Create a builder:

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

  - Add config sources and build:

    ```
    Config config =
    builder.addDefaultSources().withSources(mySource).withConverters(myConverter).bu
    ild;
    ```

- (optional) Manage the lifecycle of the config

```
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. Dynamically added config sources after the `Config` object has been built would be ignored, which means `Config.getConfigSources` always returns the same collection of `ConfigSource`s. The same rule applies to `ConfigSourceProvider.getConfigSources`.

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. Specifying an empty string as the value effectively erases the property. 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`.

```
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`:

```
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.

Dynamic ConfigSource

As a `ConfigSource` is a view of configuration data, its data may be changing, or unchanging. If the data is changing, and a `ConfigSource` can represent its changes, we call that `ConfigSource` a dynamic `ConfigSource`, since at any two moments two operations on it may reflect two different sets of underlying configuration data. If instead the data is unchanging, we call the `ConfigSource` a static `ConfigSource`, since at any two moments two operations on it will reflect only one set of underlying (unchanging) configuration data. A caller cannot know whether a `ConfigSource` is dynamic or static.

For the property lookup, the method `config.getValue()` or `config.getOptionalValue()` retrieves the up-to-date value. Alternatively, for the injection style, the following lookup should be used to retrieve the up-to-date value.

```
    @Inject
    @ConfigProperty(name="myprj.some.dynamic.timeout", defaultValue="100")
    private javax.inject.Provider<Long> timeout;
```

Whether a `ConfigSource` supports this dynamic behavior or not depends on how it's implemented. For instance, the default `ConfigSource` microprofile-config.properties and Environment Variables are not dynamic while System Properties are dynamic by nature. MicroProfile Config Implementation can decide whether a `ConfigSource` can be dynamic or not.

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.

Converters

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

Built-in Converters

The following `Converter`s 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`, `java.lang.Integer`, and `java.util.OptionalInt`
- `long`, `java.lang.Long`, and `java.util.OptionalLong`
- `float` and `java.lang.Float`; a dot '.' is used to separate the fractional digits
- `double`, `java.lang.Double`, and `java.util.OptionalDouble`; 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 `Converter`s have the `@Priority` of `1`.

The converters for these types must throw an NPE if given a null value to convert.

Adding custom Converters

A custom `Converter` must implement the generic interface `org.eclipse.microprofile.config.spi.Converter` and conform to the API requirements of that interface. The Type parameter of the interface is the target type the String is converted to. If your converter targets a wrapper of a primitive type (e.g. `java.lang.Integer`), the converter applies to both the wrapper type and the primitive type (e.g. `int`) 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.

```
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.

```
@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

If a converter returns `null` for a given config value, the property will be treated as being deleted. If it is a required property, `NoSuchElementException` will be thrown. Even if `defaultValue` is specified on the property injection, the `defaultValue` will not be used.

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.

Config Profile

Config Profile indicates the project phase, such as dev, testing, live, etc.

Specify Config Profile

The config profile can be specified via the property `mp.config.profile`, which can be set in any of the configuration sources. The value of the property can contain only characters that are valid for config property names. This is because the name of the profile is directly stored in the name of the config property. It can be set when starting your application. e.g.

```
java -jar myapp.jar -Dmp.config.profile=testing
```

The value of the property `mp.config.profile` shouldn't be updated after the application is started. It's only read once and will not be updated once the `Config` object is constructed. If the property value of `mp.config.profile` is modified afterwards, the behavior is undefined and any changes to its value made later can be ignored by the implementation.

The value of the property `mp.config.profile` specifies a single active profile. Implementations are free to provide additional mechanisms to support multiple active profiles. If the property `mp.config.profile` is specified in multiple config sources, the value of the property is determined following the same rules as other configuration properties, which means the value in the config source with the highest ordinal wins.

How Config Profile works

On Property level

The configuration property that utilizes the Config Profile is called a "profile-specific" property. A "profile-specific" property name consists of the following sequence: `% <profile name>.<original property name>`.

Conforming implementations are required to search for a configuration source with the highest ordinal (priority) that provides either the property name or the "profile-specific" property name. If the configuration source provides the "profile-specific" name, the value of the "profile-specific" property will be returned. If it doesn't contain the "profile-specific" name, the value of the plain property will be returned.

For instance, a config source can be specified as follows.

```
%dev.vehicle.name=car
%live.vehicle.name=train
%testing.vehicle.name=bike
vehicle.name=lorry
```

A config property associated with the Config Profile can be looked up as shown below.

```
@Inject @ConfigProperty(name="vehicle.name") String vehicleName;
```

```
String vehicleName = ConfigProvider.getConfig().getValue("vehicle.name",
String.class);
```

If the property `mp.config.profile` is set to `dev`, the property `%dev.vehicle.name` is the Active Property. An active property overrides the properties in the same config source. In more details, if `mp.config.profile` is set to `dev`, the property `%dev.vehicle.name` overrides the property `vehicle.name`. The `vehicleName` will be set to `car`. The properties `%live.vehicle.name` and `%testing.vehicle.name` are inactive config properties and don't override the property `vehicle.name`.

If `mp.config.profile` is set to `live`, the property `%live.vehicle.name` is the active property. The `vehicleName` will be `train`. Similarly, `bike` will be the value of `vehicleName`, if the profile is `testing`.

On Config Source level

Config Profile also affects the default config source `microprofile-config.properties`. If multiple config sources exist under the `META-INF` folder on the classpath with the name like `microprofile-config-<profile_name>.properties`, the config source matching the active profile name will also be loaded on top of the default config source `microprofile-config.properties`. It means if the same property specified in both config sources, the value from the config source `microprofile-config-<profile_name>.properties` will be used instead. If the property `mp.config.profile` is specified in the `microprofile-config-<profile_name>.properties`, this property will be discarded.

For instance, there are following config sources provided in your application.

```
META-INF\microprofile-config.properties
META-INF\microprofile-config-dev.properties
META-INF\microprofile-config-prod.properties
META-INF\microprofile-config-testing.properties
```

If the property `mp.config.profile` is set to `dev`, the config source `microprofile-config-dev.properties` will be loaded onto the config source of `microprofile-config.properties`. Similarly, if `mp.config.profile` is set to `prod`, the config source `microprofile-config-prod.properties` will be loaded onto the config source of `microprofile-config.properties`. However, if `mp.config.profile` is set to `live`, no additional property file will be loaded on the top of `microprofile-config.properties` as the config source `microprofile-config-live.properties` does not exist.

Property Expressions

The value of a configuration property may contain an expression corresponding to another configuration property. An expression string is a mix of plain strings and expression segments, which are wrapped by the sequence `${ ⋯ }`.

Consider the following configuration properties file:

```
server.url=http://${server.host}/endpoint
server.host=example.org
```

When looking up the `server.url` property, the value will be resolved and expanded to `http://example.org/endpoint`. All MicroProfile Config rules still apply. The `Config` is able to resolve expressions from different ConfigSources.

Additionally, it is also possible to use the following syntax for property expressions:

- `${expression:value}` - Provides a default value after the `:` if the expression doesn't find a value.
- `${my.prop${compose}}` - Composed expressions. Inner expressions are resolved first.
- `${my.prop}${my.prop}` - Multiple expressions.

Consider the following configuration properties file:

```
server.url=http://${server.host:example.org}:${server.port}/${server.endpoint}
server.port=8080
server.endpoint=${server.endpoint.path.${server.endpoint.path.bar}}
server.endpoint.path.foo=foo
server.endpoint.path.bar=foo
```

The property `server.url` is expanded to `http://example.org:8080/foo`.

If an expression cannot be expanded and does not have a default value, a `NoSuchElementException` is thrown. In the Optional case, an empty Optional will be returned.

The number of recursion lookups is not infinite, but a limited number for composed expressions. Implementations are encouraged to limit the number to `5`, but they can use higher limits if they wish to. When the number of allowed lookups exceeds the limit, an `IllegalArgumentException` is thrown.

Property expressions applies to all the methods in `Config` that performs resolution of a configuration property, including `getValue`, `getValues`, `getConfigValue`, `getValues`, `getOptionalValue`, `getOptionalValues` and `getConfigProperties`. The methods `getValue` and `getProperties` in `ConfigSource`, may support property expressions as well, but it is not required by the specification.

Property expressions must also be applied to configuration properties injected via CDI. A default value defined via `org.eclipse.microprofile.config.inject.ConfigProperty#defaultValue` is not eligible to be expanded since multiple candidates may be available.

If a configuration property value or default value requires the raw value without expansion, the expression may be escaped with a backslash ("\", double "\\" for property file-based sources). For instance:

```
server.url=\\${server.host}
server.host=localhost
```

The value of `server.url` is `${server.host}`.

Backwards Compatibility

MicroProfile Config implementations MUST provide a way to disable variable evaluation to provide backwards compatibility. The property `mp.config.property.expressions.enabled` was introduced for this purpose. The value of the property determines whether the property expression is enabled or disabled. The value `false` means the property expression is disabled, while `true` means enabled.

If property expression expansion is not desirable for a specific case, the raw value on a configuration property may be retrieved by calling `getRawValue()` in `ConfigValue`.

Specific sources may already use a similar or identical syntax to the one described in this specification. To preserve this usage, `ConfigSource#getValue()` should perform the expression substitution and then return the resolved value. Should such a source return a value with an expression from `ConfigSource#getValue()`, usual expression substitution does occur as described by this spec.

Release Notes for MicroProfile Config 2.0

A full list of changes delivered in the 2.0 release can be found at MicroProfile Config 2.0 Milestone.

Incompatible Changes

- ConfigSource#getPropertyNames is no longer a default method. The implementation of a ConfigSource must implement this method. (431)
- Previous versions of the specification would not evaluate Property Expressions. As such, previous working configuration may behave differently (if the used configuration contains values with Property Expressions syntax). Check the Property Expressions section on how to go back to the previous behaviour.
- Empty value or other special characters are no longer a valid config value for a particular return type. Refer to the earlier section of this spec for more details. In the previous release, the empty value was returned as an empty value. From now on, the empty value will cause `NoSuchElementException` to be thrown. (446) (531) (532) (397) (633)

API/SPI Changes

- Convenience methods have been added to Config allowing for the retrieval of multi-valued properties as lists instead of arrays (#496)
- Enable bulk-extraction of config properties into a separate POJO by introducing `@ConfigProperties` (240)
- Enable users to determine the winning source for a configuration value (312) (43)
- Expose conversion mechanism in Config API (492)
- Add unwrap() methods to Config (84)

Functional Changes

- Support Configuration Profiles so that the corresponding properties associated with the active profile are used (#418)
- Provide built-in Converters: OptinalInt, OptionalLong and OptionalDouble (513)
- Clarifies that Converters for primitive wrappers apply to primitive types as well (520)
- Clarify that nulls cannot be passed in to Converters (542)
- Support Property Expressions: This provides a way to set and expand variables in property values (118)
- Specify the behaviour when a converter returns null (608)

Other Changes

- Update to Jakarta EE8 APIs for MP 4.0 (469)

- Enable MicroProfile Config repo to be built with Java 11 (555)
- TCK changes: (563) (319) (573)
- Spec clarification: (371)

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

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 1.2

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 1.1

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)