Configuration for Microprofile

Mark Struberg, Emily Jiang

0.3, 2017-01-18

Table of Contents

Specification: Configuration for Microprofile

Version: 0.3

Status: Proposal

Release: 2017-01-18

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.

Rational

Released binaries often contain functionality which need to behave slightly differently depending on the deployment. This might be different 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 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 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 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
```

Configuration Usage Examples

A configuration object can be obtained programmatically via the `ConfigProvider` or automatically via `@Inject Config`. An application can then access its configured values via a `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);
        Integer serverPort = config.getValue("acme.myprj.some.port", Integer.class);

        callToServer(serverUrl, serverPort);
    }
}
```

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

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

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.

Simple Dependency Injection Example

Microprofile-Config also provides ways to inject configured values into your beans using the `@Inject` and the `@ConfigProperty` qualifier.

```
@ApplicationScoped
public class InjectedConfigUsageSample {

    @Inject
    private Config config;

    //The property myprj.some.url must exist in one of the configsources, 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;
}
```

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 an `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.
- 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. 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. 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 but with only the default converters added. The `ConfigBuilder` object can be filled manually via `ConfigBuilder#withSources(ConfigSources… sources)`. This configuration instance will by default not be shared by the `ConfigProvider`. This method is intended be used if a IoT container or any other external Factory can be used to give access to a manually created shared `Config`.

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

- A factory method `ConfigProvider#registerConfig(Config config, ClassLader 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 `ConfigProvider#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` is bound to an Application, it gets properly removed if the Application gets destroyed. The Config system does not create any memory leaks in that case.

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.

The ordinal for property file based ConfigSources can be configured using the key `config_ordinal` inside the property file.

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

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 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.

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 typeseafe 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` , values for `true` (case insensitive) "true", "1", "YES", "Y" "ON". Any other value will be interpreted as `false`
- `Integer`
- `Long`
- `Float` , a dot '.' is used to separate the fractional digits
- `Double` , a dot '.' is used to separate the fractional digits
- `Duration`
- `LocalTime`
- `LocalDate`
- `LocalDateTime`
- `OffsetDateTime`
- `OffsetTime`
- `Instant`

All built-in `Converter` 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.