



Effortless Application Configuration with MicroProfile Config



The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

User Guide

Contents

| | |
|---|-----------|
| What is MicroProfile? | 1 |
| Getting Started with MicroProfile | 2 |
| Why MicroProfile Config? | 3 |
| Dynamic Config Values | 3 |
| Powerful Abstractions | 3 |
| Easy DevOps | 3 |
| Good Base and Powerful Extensions | 4 |
| The Config Source | 6 |
| Ordinal Values | 6 |
| The Payara Built in Config Sources | 7 |
| Domain | 7 |
| Application | 7 |
| Config | 8 |
| Module | 8 |
| Server | 8 |
| Directory | 9 |
| Password | 10 |
| JNDI | 11 |
| Cluster | 11 |
| Custom Config Sources | 12 |
| Converters | 15 |
| Config Value | 18 |
| Payara Specific Configuration Properties | 19 |
| Dynamic Config Values | 20 |
| Summary | 21 |

Every application will go through different stages on its way to production. From development, to testing to staging to production, all these stages will have different configurations just for those environments. For instance, an app that uses an external service like a fraud detection API will have different configurations for the service depending on the stage the application is being run in. Another application that needs to import data in a batch service, for instance, will also have different configuration for which data to import depending on which stage the application is in.

Applications need to behave differently depending on some configuration values provided. It would be tedious if these configuration values were bundled with the application. This would mean having to repackage and deploy the application for each stage. It would be ideal to have a mechanism that gives you, the developer, a way to define different configuration values for different application stages without having to repackage and re-deploy your application.

This is what the MicroProfile Config API is designed for. It is a specification that gives your application transparent access to different configuration sources without having to touch the application itself. So for instance, in the fraud detection example above, you can have a dummy URL for the development stage, a test URL from the service provider for testing and staging and finally the live URL for the production stage.

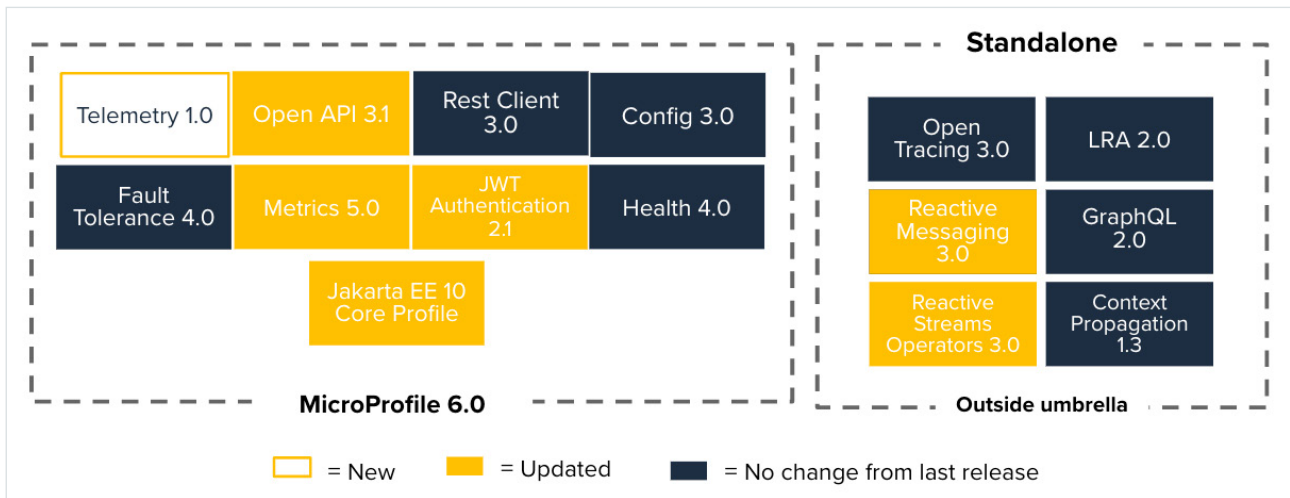
The third of the twelve-factor app deals with configuration and how it can help develop modern, cloud native applications. The tenet of this point is to have application configuration stored in the environment in which the application is running, rather than bundling it with the application itself.

The Config specification allows you to define these configuration values in a hierarchical way such that the right value is returned automatically depending on the stage. This chapter is about the MicroProfile Config specification. By the end of this guide, you will learn how to incorporate the Config API into your application and use it to externalize dynamic application configuration values.

What is MicroProfile?

MicroProfile is a community driven initiative, built on top of the Jakarta EE Core Profile, that is a collection of abstract specs that form a complete solution to developing cloud native, Jakarta EE microservices. The goal is to create a set of APIs that abstracts you from their implementations so that you can create highly portable microservices across vendors.

The current release is version 6.0 which is a major release that includes MicroProfile Config 3.0, MicroProfile Fault Tolerance 4.0, MicroProfile Health 4.0, MicroProfile Metrics 5.0, and MicroProfile Rest Client 3.0. MicroProfile 6.0 is built on top of the Core Profile of Jakarta EE, a slimmed down version of Jakarta EE “that contains a set of Jakarta EE Specifications targeting smaller runtimes suitable for microservices and ahead-of-time compilation.” The Core Profile of Jakarta EE was released as part of Jakarta EE 10.



As abstract specifications, the various implementations are free to implement the base specs and add custom features on top. Payara Server is one of the popular implementations of the MicroProfile spec and adds quite a number of custom features on top of the base specs. You can download a free trial of [Payara Enterprise here](#) to follow along with the rest of the guide.

Getting Started with MicroProfile

To get started with the MicroProfile API, you need to include it as a dependency in your project as shown below.

```
<dependency>
  <groupId>org.eclipse.microprofile</groupId>
  <artifactId>microprofile</artifactId>
  <version>6.0</version>
  <type>pom</type>
  <scope>provided</scope>
</dependency>
```

With the MicroProfile API dependency in place, you have access to all the APIs of the project. In our case, the Payara Server will provide the implementation for us.

What Is MicroProfile Config?

MicroProfile Config is an API that aims to simplify application configuration in different environments and from different sources without the need to redeploy or restart the application. An application might need different port numbers for different requests, or certain features need to be switched on or off based on certain configurations. The Config API aims at unifying and simplifying application configurations such that it provides an aggregated view of the various configuration options.

Why MicroProfile Config?

There are 4 compelling reasons to give the MP Config API a try. They are:

Dynamic Config Values

The benefit of abstracting config values from an application is the ability to change the config values without necessarily needing to package and redeploy the application. With MP Config, you can have your updated config values picked up in your application without having to redeploy.

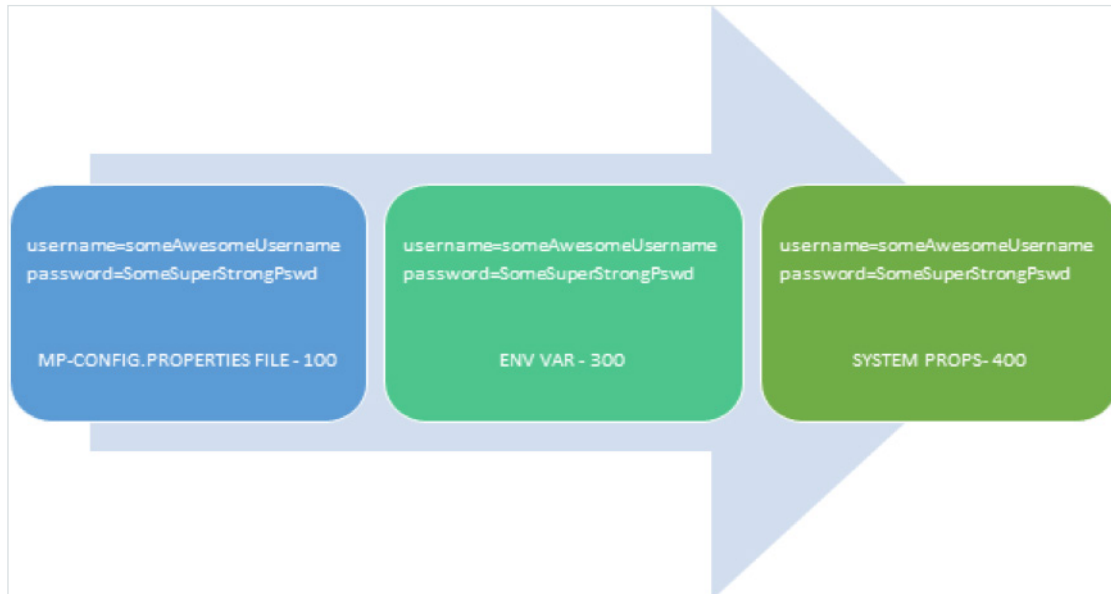
Powerful Abstractions

The core of the MP Config API is the ConfigSource object. It's an abstraction of any source of configuration values. Given the almost infinite options available for storing config values, it's good to have the Config API abstract all config sources away into a config source that can be queried. The developer can then get a set of well-defined methods to interact with the config source object.

Easy DevOps

For every application, different data is required for development, testing and production. For example, let's say we are developing an e-commerce app using microservices. One of these services - OrderService - needs to access data about clients from AccountService. AccountService is configured to give three kinds of data - dev, testing, production - depending on the credentials one authenticates with it. In developing OrderService, we'd want to use different credentials for the appropriate stage. We can easily do that with the Config API.

We set the three credentials in three different Config sources (more about this later) with the same keys. Because the Config sources are placed in a hierarchical order based on their ordinal values (don't worry if you're unfamiliar, we discuss this below), we can put dev credentials in the lowest priority source with production being in the highest priority source as shown below.



From the above image, we have the same values set in three different places. The source with the highest number takes the top of the hierarchy. So in development, we could only have the mp-config.properties file, in testing an ENVAR and in production, Systems Properties. Through it all, your code would stay the same, but the values you get from the Config runtime will be different depending on the config sources available. This can greatly enhance your software development operations.

Good Base and Powerful Extensions

The MP Config consists of a good base that alone can get you quite far. But the goal of the entire MP project is to encourage the community to innovate around it. And that is what you get with the MP Config API. The Payara Platform has some very intuitive extensions built on the Config API that completes it. For example, one of the config sources supported by Payara Server is Directory. In this case, you define a directory in the server Admin Console as ConfigSource and any file in the directory becomes a key value pair with the name of the file acting as key and content acting as value.

The simplest Config API use is shown below

```
@Inject
@ConfigProperty(name = "jakarta.version")
private String jakartaVersion;
```

The code snippet uses `@Inject` and `@ConfigProperty` annotations on the String `jakartaVersion`. The `@ConfigProperty` annotation is from the MicroProfile Config API. It takes two parameters - `name` and `defaultValue`. The `name` refers to the name or key to be used to identify the value from a given source. In this case, the `name` is `jakarta.version`, and the config runtime is going to iterate through all available config sources in search of a key value pair that has the key set to `jakarta.version`.

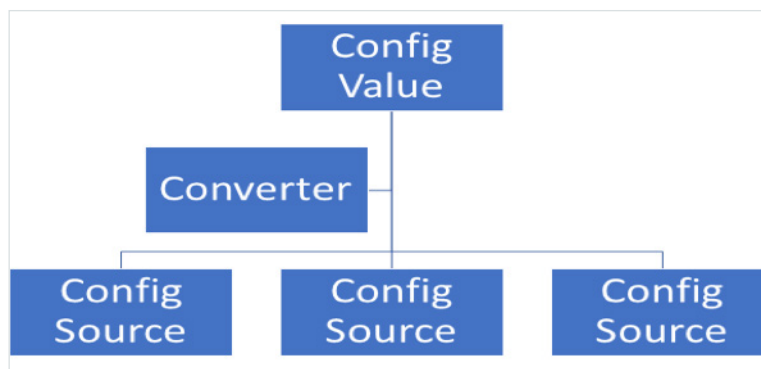
In this case, we have the value set in the default `microprofile-config.properties` file. This file is a normal text file that you put in the META-INF folder. Within it you create key value pairs as show below:

```
jakarta.version=10.0.0
application.server=payara-6-community
free.memory.limit=40
```

The above snippet shows a key-value pair config values. The code snippet shown earlier uses the `jakarta.version` key to get hold of the value in this file, in this example `10.0.0`.

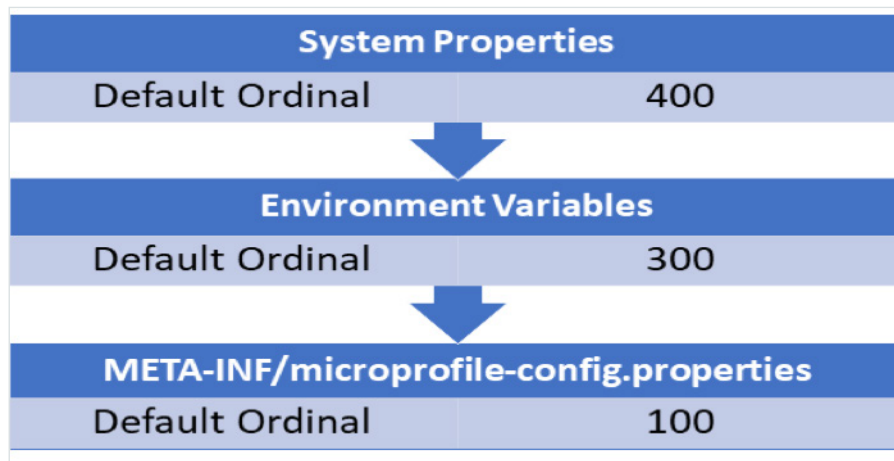
For the developer, you have all the config sources automatically aggregated and iterated on your behalf by the Config runtime. This is especially powerful in a microservices environment where you will need to configure different things for different purposes.

The Config API is made up of 3 parts - the config source, converters and config value as shown below.



The Config Source

The config source is an abstraction over the source of a configuration. This source of configuration could be a database, a properties file, system properties, environment variables and so on. The Config API abstracts any source from which configuration values might be read into a ConfigSource. There cannot be a configuration without a config source. The API defines 3 config sources out of the box from which it will search for configuration values by default. These 3 sources also have their default ordinal values defined.



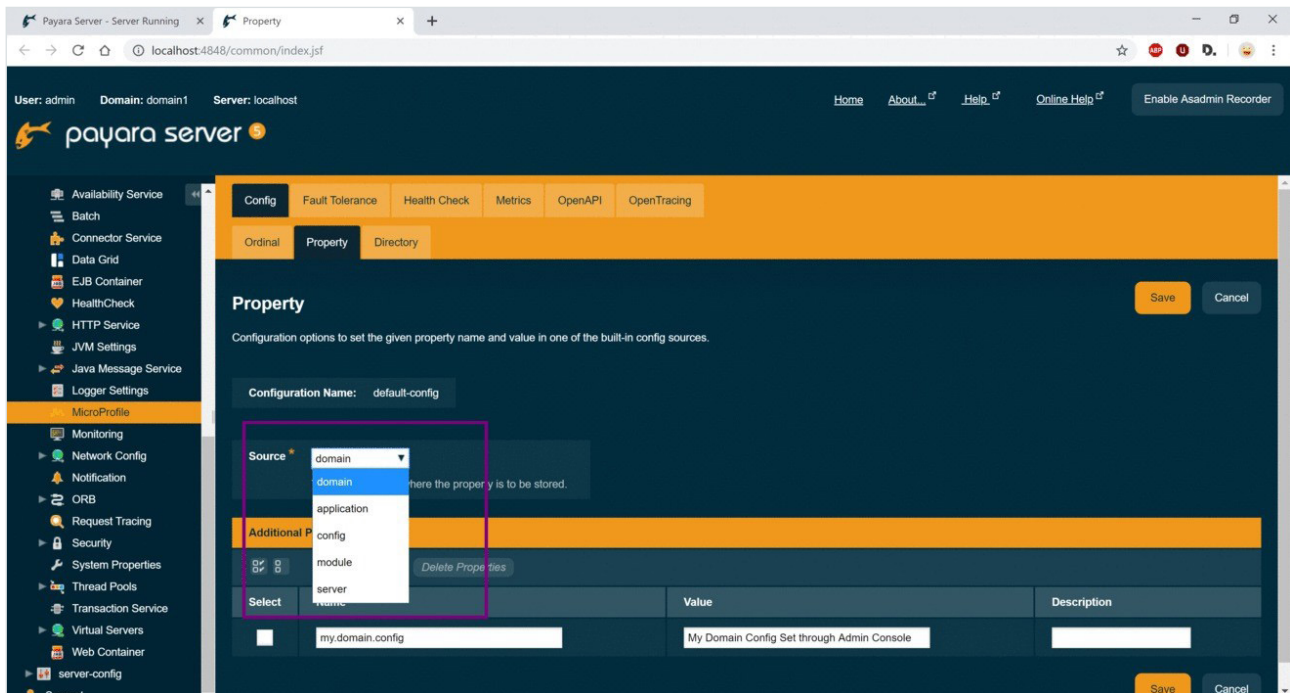
Ordinal Values

An ordinal value determines the position of a given config source in a hierarchy. It defines which config source to take precedence in the event of a conflict in terms of config values. The config source with a higher ordinal value has a higher precedence over a config source with a lower ordinal value.

As shown in the image above, System Properties has an ordinal value of 400, giving it the highest priority over environment variables with an ordinal of 300 and the microprofile-config.properties file with an ordinal of 100.

The Payara Built in Config Sources

Payara Server also comes with its own built-in config sources as shown in the image below:



From this image you can see some of the different config sources supported by Payara Server, namely Domain, Application, Config, Module and Server. You can conveniently set key value pairs to any of these sources right from the server admin interface as shown in the above image. Let's briefly examine what these sources are.

Domain

The Domain config source stores key value pairs in the domain.xml file and makes the same available for all applications deployed in that domain.

Application

This source constraints stored key value pairs to a selected application. Whatever key value pairs will only be visible to the chosen application of all server instances.

Config

This source stores key value pairs at the level of a named configuration in the domain.xml file. There are two named configurations that you can choose from the Payara Server admin - default config and server config.

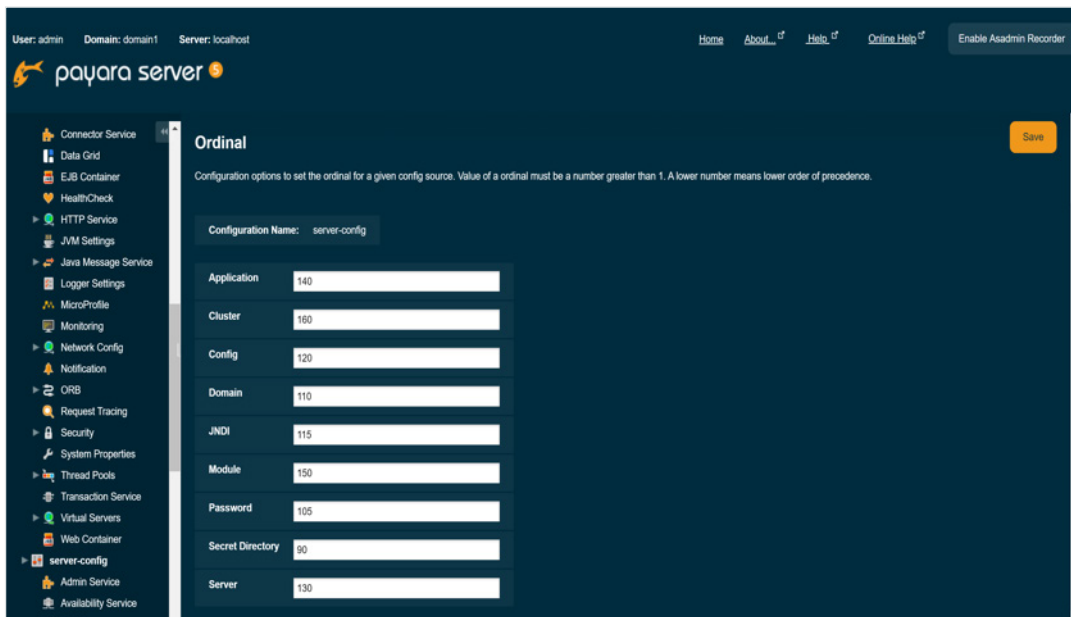
Module

This source stores key value pairs at the level of individual modules of an application. Whatever key value you store in a module will only be available to the module alone.

Server

This source stores key value pairs at the level of the named server in the domain.xml file. Key value pairs will only be available on the named server instance.

You can set the ordinal values for all these config sources as shown below:



The screenshot shows the Payara Server Admin console interface. The top navigation bar includes the user 'admin', domain 'domain1', and server 'localhost'. The main content area is titled 'Ordinal' and contains a form for setting ordinal values for the 'server-config' configuration source. The form includes a 'Configuration Name' dropdown set to 'server-config' and a list of configuration sources with their respective ordinal values.

| Configuration Source | Ordinal |
|----------------------|---------|
| Application | 140 |
| Cluster | 100 |
| Config | 120 |
| Domain | 110 |
| JNDI | 115 |
| Module | 150 |
| Password | 105 |
| Secret Directory | 90 |
| Server | 130 |

As you can see, the Payara Server Admin Console is a convenient yet powerful tool you can use to configure your MicroProfile Config key value pairs in a very fine grained manner. Aside from the built-in config sources that come with Payara Server, you can also create your own custom Config Sources quite easily.

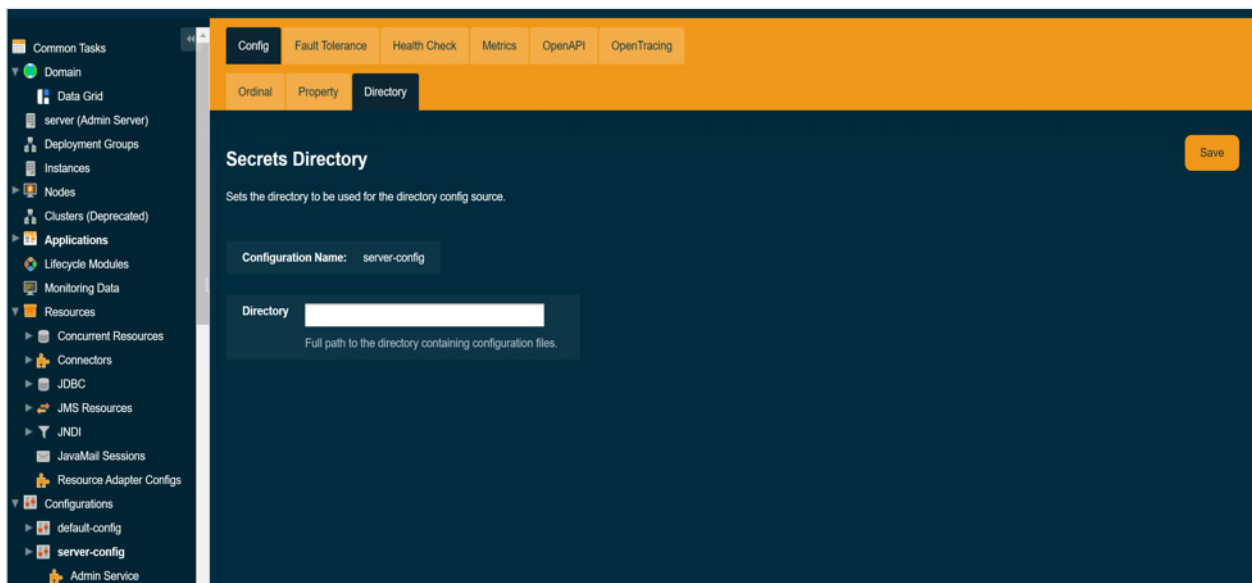
In addition to the above sources, the Payara Platform also supports the following config sources:

Directory

This source is perhaps the most powerful of all the sources. It treats file names as keys and file content as value. So let's say you configure a given directory such as `/opt/my/config/source`, you can get the content of any file in this directory as config value when you pass in the file name as key. For example `/opt/my/config/source/api-key.txt` is a file in the configured directory that contains the key to an external API. We could easily get the value thus:

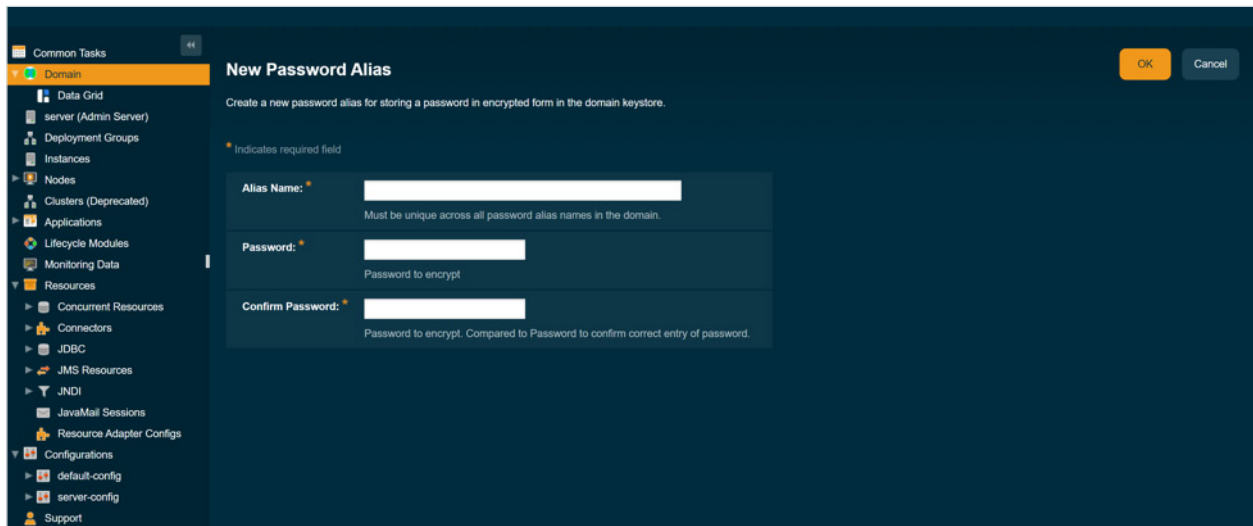
```
@Inject
@ConfigProperty(name = "api-key.txt", defaultValue = "noKey")
private String apiKey;
```

The code uses the name of the file as the name of the key for our config value retrieval. The value of the file will be injected into the String `apiKey`. The directory source can also be used to read secret files in Kubernetes. You can configure the path to the directory in the Payara Server Admin Console as shown below.



Password

This source stores config values in the Payara Server password alias store. The password alias you set becomes the key to the config runtime and the plain text password will be injected as the config value. You can create password aliases in the Payara Server Admin Console as shown below.



The screenshot shows the 'New Password Alias' form in the Payara Server Admin Console. The left sidebar contains a navigation menu with categories like Common Tasks, Domain, Data Grid, server (Admin Server), Deployment Groups, Instances, Nodes, Clusters (Deprecated), Applications, Lifecycle Modules, Monitoring Data, Resources, Concurrent Resources, Connectors, JDBC, JMS Resources, JNDI, JavaMail Sessions, Resource Adapter Configs, Configurations, default-config, server-config, and Support. The main panel is titled 'New Password Alias' and includes a sub-header 'Create a new password alias for storing a password in encrypted form in the domain keystore.' Below this, there are three required fields: 'Alias Name' (with a note 'Must be unique across all password alias names in the domain.'), 'Password' (with a note 'Password to encrypt'), and 'Confirm Password' (with a note 'Password to encrypt. Compared to Password to confirm correct entry of password.'). There are 'OK' and 'Cancel' buttons at the top right.

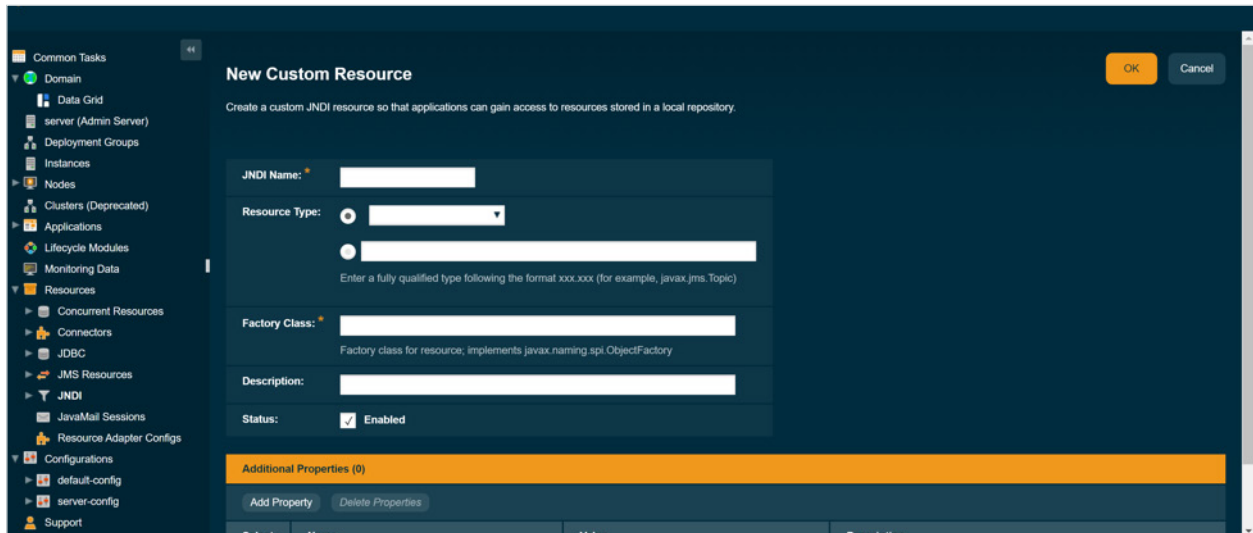
Once you create a password alias, you can use it as the key to `@ConfigProperty` annotation as shown below.

```
@Inject
@ConfigProperty(name = "api-key.txt", defaultValue = "noKey")
private String apiKey;
```

The above has the String 'my-alias' set as the name of the key to `@ConfigProperty`. That string is a password alias set using the Payara Server Admin Console. Running the above code, the config runtime will inject the plain text, decrypted password into the field `passwordAlias`. Using the password config source is another great way to deal with passwords and secrets on the Payara Platform through the Config API. Password alias is available to all server instances in a domain.

JNDI

This source stores config values in the JNDI tree and uses the dotted name path of the JNDI tree. You can create custom JNDI resources using the Payara Server Admin Console as shown below.



The screenshot shows the 'New Custom Resource' dialog in the Payara Server Admin Console. The dialog is titled 'New Custom Resource' and has a subtitle 'Create a custom JNDI resource so that applications can gain access to resources stored in a local repository.' The dialog contains the following fields:

- JNDI Name:** A text input field.
- Resource Type:** A dropdown menu with a radio button next to it. Below the dropdown is a text input field with the placeholder text 'Enter a fully qualified type following the format xxx.xxx (for example, javax.jms.Topic)'.
- Factory Class:** A text input field with the placeholder text 'Factory class for resource; implements javax.naming.spi.ObjectFactory'.
- Description:** A text input field.
- Status:** A checkbox labeled 'Enabled' which is checked.

Below the main fields is a section titled 'Additional Properties (0)' with a yellow background. It contains two buttons: 'Add Property' and 'Delete Properties'. At the bottom of the dialog is a table with columns 'Select', 'Name', 'Value', and 'Description'.

Cluster

This config source uses the Payara Server Domain Data Grid (DDG) to store config values as key value pairs in memory. The config values stored in the DDG are available to all server instances within a given Payara Domain.

Custom Config Sources

You can also create custom config sources by implementing the `org.eclipse.microprofile.config.spi.ConfigSource` interface as shown below.

```
public class MyCustomConfigSource implements ConfigSource {  
    @Override  
    public Set<String> getPropertyNames() {  
        return null;  
    }  
  
    @Override  
    public String getValue(String propertyName) {  
        return null;  
    }  
  
    @Override  
    public String getName() {  
        return null;  
    }  
}
```

The above code snippet declares a class called `MyCustomConfigSource` that implements `ConfigSource`. The `ConfigSource` interface has 5 methods, 2 of which have default implementations. The `getPropertyNames` method returns all property names known to this custom config source. The `getValue` method, which takes a `String` and returns a `String` is the method that will be queried on your behalf for a value when you pass a given key. The `getName` method returns the name of this `ConfigSource`.

One of the default methods in the `ConfigSource` interface is `getOrdinal`, which conditionally returns an ordinal of 100. Remember an ordinal value determines the place of a config source in a hierarchy? The Config API gives you the option of setting the ordinal of your custom config source - or even any of the built in ones - by including a key value pair with the key set to `config_ordinal` and the value set to an integer value.

You can implement your custom config source to fetch data from the database using JDBC as in the example below.

```
public class MyDBConfigSource implements ConfigSource {
    private DataSource dataSource;
    private final Map<String, String> PROPERTIES_MAP = new HashMap<>();

    public MyDBConfigSource() {
        try {
            dataSource = (DataSource) new InitialContext().lookup("jdbc/myDB");
        } catch (final NamingException e) {
            throw new IllegalStateException(e);
        }
    }

    @Override
    public Map<String, String> getProperties() {
        try {
            final Connection connection = dataSource.getConnection();
            final PreparedStatement query =
                connection.prepareStatement("SELECT CONFIG_KEY, CONFIG_
VALUE FROM MPCONFIG");
            final ResultSet names = query.executeQuery();
            while (names.next()) {
                PROPERTIES_MAP.put(names.getString(0), names.getString(1));
            }
            DbUtils.closeQuietly(names);
            DbUtils.closeQuietly(query);
            DbUtils.closeQuietly(connection);
        } catch (final SQLException e) {
            e.printStackTrace();
        }
        return PROPERTIES_MAP;
    }

    @Override
    public String getValue(String propertyName) {
        if (PROPERTIES_MAP.containsKey(propertyName)) {
            return PROPERTIES_MAP.get(propertyName);
        }
        return null;
    }
}
```

```
@Override
public int getOrdinal() {
    return 195;
}

@Override
public String getName() {
    return "MyDBConfig";
}

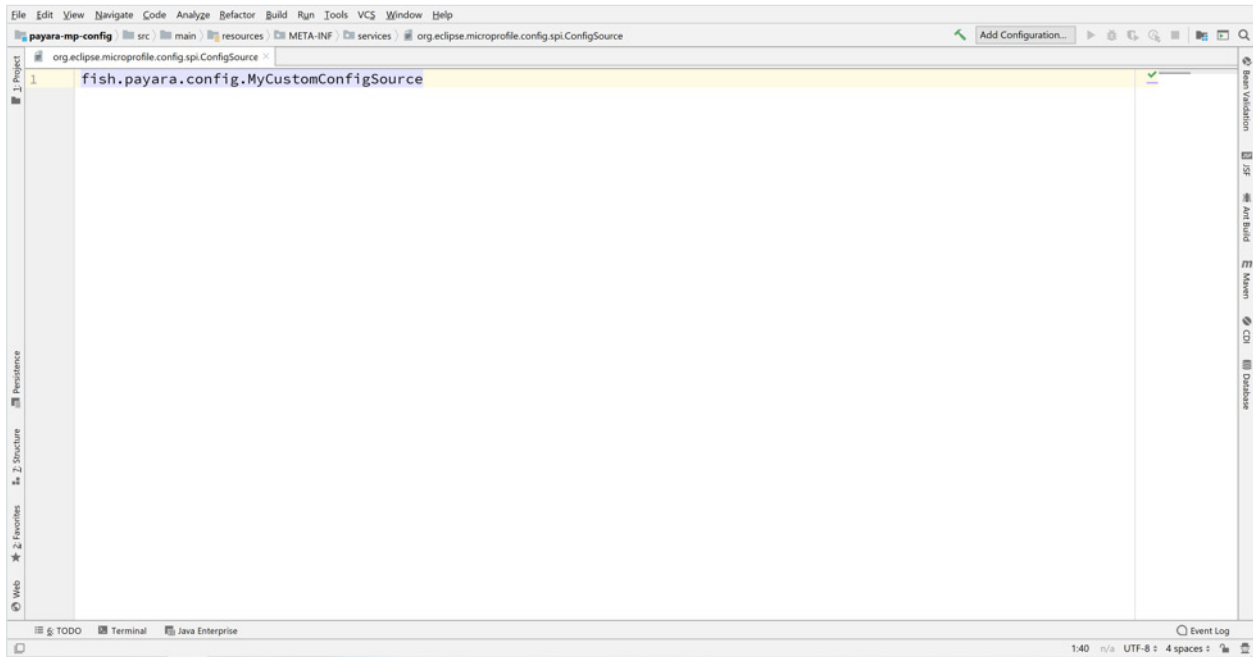
@Override
public Set<String> getPropertyNames() {
    return PROPERTIES_MAP.keySet();
}
}
```

The above code declares a class, `MyDBConfigSource` that implements `ConfigSource`. This time, however, we implement a simple database retrieval of config values into the map `PROPERTIES_MAP`. The `getPropertyNames` method uses the JDBC API to retrieve values from a table called `MPCONFIG`. These values are then stored into the `PROPERTIES_MAP` which is returned as the return value of the method.

The `getValue` method checks whether the `PROPERTIES_MAP` contains the key passed and returns the value or null if not. We want this config source to have a custom config ordinal of 195 (could be any number though), so the `getOrdinal` method returns 195. This is a basic example of implementing a database custom config source. Of course, the table would need to be populated with values from some other source. But in this example, we assume that has already been done.

You are free to implement the `ConfigSource` interface according to your business needs. What matters is that once it's registered as a `ConfigSource`, it should be capable of returning values as long as a key matches in any of the configured config sources.

Once you've implemented the `ConfigSource` interface, you will need to register it. You do so by registering it with the fully qualified class name in the file `/META-INF/services/org.eclipse.microprofile.config.spi.ConfigSource` as shown below:



In the above image, we have registered our custom `ConfigSource` implementation by including the fully qualified class name in the file. That is all you need to do to register a custom `ConfigSource`.

Converters

A converter is a mechanism for converting config values from the default String to their respective Java types. The Config API provides the following converters with default Priority of 1 out of the box.

- 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`

For situations where no default or custom converter is available for a given Java type, a dynamic converter is created automatically if:

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

Besides the above, you can also create custom converters by implementing the `org.eclipse.microprofile.config.spi.Converter` interface as shown below:

```
public class MyCustomConfigConverter implements Converter<Employee> {
    @Override
    public Employee convert(String value) {
        return JsonbBuilder.create().fromJson(value, Employee.class);
    }
}
```

The above code declares a Java class called `MyCustomConfigConverter` that implements the `Converter` interface. In the above example, we are converting from `String` to our custom Java type `Employee`, which we realize in the `convert` method of the interface. The `Employee` bean is reproduced below.

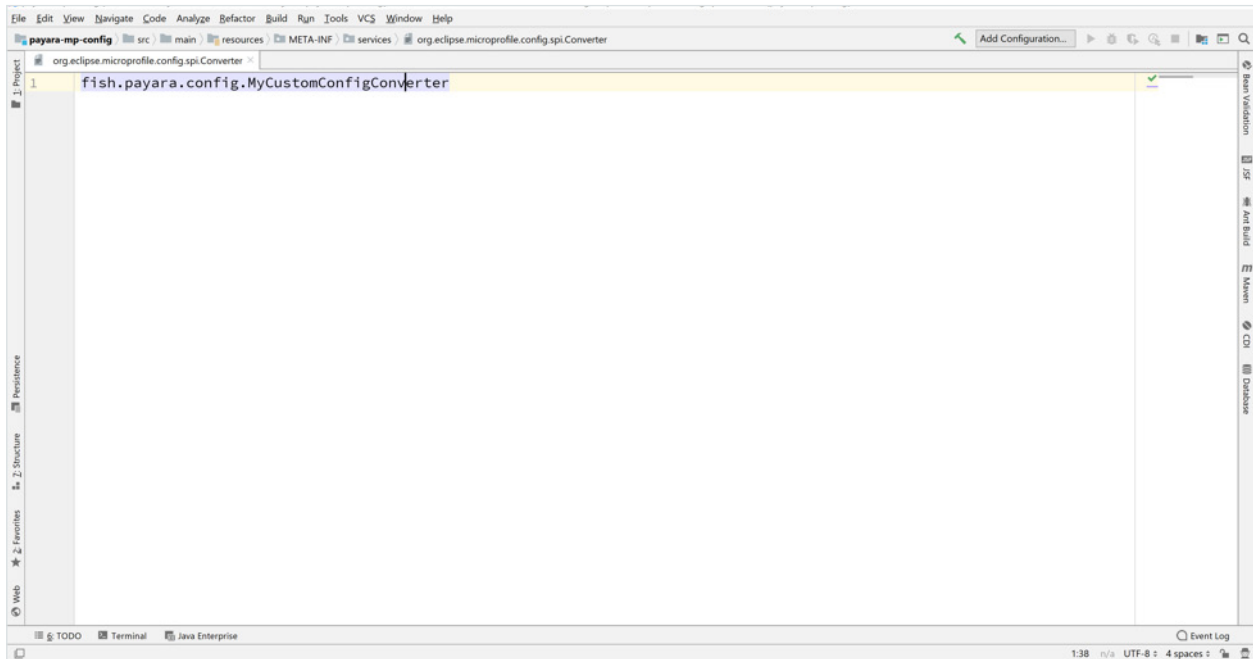
```
public class Employee {

    private String id;

    private LocalDateTime createdOn;
    private LocalDateTime updatedOn;
    private String firstName;
    private String middleName;
    private String lastName;
    private String businessKey;
    private LocalDate dateOfBirth;

}
```

After implementing the `Converter` interface, we need to register it. To do so, we include the fully qualified class name of the implementation in the file `/META-INF/services/org.eclipse.microprofile.config.spi.Converter` as shown below.



If your custom converter has a target type for which a default converter exists, your custom converter will take precedence. You can also set the Priority of your custom converter using the `@jakarta.annotation.Priority` annotation. A custom converter without an explicit `@jakarta.annotation.Priority` annotation defaults to a priority of 100.

The Payara Platform provides built-in custom converters for `URL` and `InetAddress` with priority of 1. This means you can directly request for `URL` based values without needing to write custom converters. For example:

```
@Inject
@ConfigProperty(name = "fish.payara.address", defaultValue = "payaraAddress")
private URL url;
```

The above code uses the `@Inject` annotation in addition to the `@ConfigProperty` annotation to inject a `URL` type into the field `url`. The value is derived from the properties file as shown below:

```
1 fish.payara.address=https://www.payara.fish
```

This conversion to the Java type URL is being carried out by the Payara Server implementation of the MicroProfile Config.

Config Value

The config value is the value returned by the Config API after querying the respective config sources using the key we pass. There are 2 primary ways of requesting config values. The first - and what we've been seeing so far - is through injection. You annotate a given field with `@jakarta.inject.Inject` and `@org.eclipse.microprofile.config.inject.ConfigProperty`.

The `ConfigProperty` annotation takes two parameters. The first is `name`, which is a `String` of the key to the value you want. The second parameter is `defaultValue`, which is a value you pass to be defaulted to in case the Config runtime fails to get a match using the key you pass to the `name` parameter.

The second means of getting a config value is through the programmatic method as shown below.

```
Config config = ConfigProvider.getConfig();
Optional<String> nonExistent = config.getOptionalValue("nonExistent",
String.class);
```

The above code snippet invokes the `getConfig()` method on the `ConfigProvider` class to get an instance of the `Config` object. This object has two twin methods for getting values from a config source. The second line invokes the `getOptionalValue()` method on the `Config` object returned in the previous line to return an `Optional` type. The type of the `Optional` object returned is determined from the type passed as the second parameter to the `getOptionalMethod()`. The first parameter is the name or key to fetch the value with. This comes in handy in situations where you are not sure of the existence of a given key value pair.

The converters are applied transparently to your config values. So for instance, I could have a field of type `BigDecimal` and request for a config value as shown below.

The converters are applied transparently on your behalf. For example, you could make a request as follows:

```
@Inject
@ConfigProperty(name = "my.very.big.number")
private BigDecimal bigDecimal;
```

The code snippet uses the `@Inject` and `@ConfigProperty` annotations to request for config value for the field type `BigDecimal`. Remember in the config source where we have this value, in this case the `mp-config.properties` file, the type is a simple `String` as shown below.

```
1 my.very.big.number=23000000000
```

As you can see, even though what we have in the properties file is a `String`, a dynamic converter was transparently created and applied to the value for us, such that we ended up getting a `BigDecimal` type.

Payara Specific Configuration Properties

The Payara Platform comes with some handy configuration properties that you can use directly in your applications to get information about the Payara Server instances you are running. These properties are exposed as keys or names you can pass to the `@ConfigProperty` to get the needed info. Some of the important ones are:

- `payara.instance.type` - This returns the type of the Payara Server runtime, whether a Server instance, Embedded or Micro.
- `payara.domain.name` - This returns the Payara domain name
- `payara.instance.starttime` - The start time of the server in `System.currentTimeMillis`
- `payara.domain.installroot` - The root directory in which the Payara Server is installed

As stated earlier, you can simply get the information using the above keys just as you would your own config sources. The Payara Platform will take care of the config source for the Payara special config properties.

Dynamic Config Values

The entire essence of using a config API is to be abstracted away from config sources and to not necessarily redeploy your application when a config value changes in any of your config sources. However, all the examples you've seen so far have just requested config values into static fields.

There is a very simple way to get changed config values picked up in your application at runtime without redeploying or rebooting the server, though. Let's see how.

```
@Inject
@ConfigProperty(name = "my.dynamic.value")
private Provider<String> myDynamicValue;
```

We declare the `@Inject` and `@ConfigProperty` annotations as usual. This time around, however, the field type annotated is the `jakarta.inject.Provider` with the type set to `String`. The `Provider`, as the name implies, provides instances of objects. So in this case, it will provide objects of type `String`. It has one method, `get()` which returns the typed instance. So any time you invoke the `get()` method, a new `String` object is returned.

With `myDynamicValue`, anytime there is a change in any of our config sources that contains the data with the key `my.dynamic.value`, we can invoke the `get()` method to get an instance of the updated value. It's that simple. Even though we have set the type of the `Provider` to `String` in this example, you can set the type to just about any valid Java type as long as there is a `Config` converter available.

Summary

MicroProfile is a very compelling community effort aimed at increasing the pace of innovation in enterprise Java software development. MicroProfile itself is an abstract spec, making it possible for various vendors to implement the base spec and add other custom features on top.

The Payara Platform is a full Jakarta EE implementation that also implements the MP API and adds quite a number of features on top of it. This guide took you through the reason for and use of the MP Config API. We saw the various Payara Server custom features available to you as a developer in addition to what the MP Config API gives you out of the box.

Should you need further support or need further info about using or transitioning your enterprise Java workload to the Payara Platform, please don't hesitate to [get in touch with us](#).

[Payara Platform Enterprise](#) is our fully supported and stable runtime designed for mission critical production environments.

Found this guide useful?

Try our other related resources:

- [Explaining Microservices: No Nonsense Guide for Decision Makers](#)
- [A Business Guide to Legacy Application Modernization For The Cloud Era](#)
- [A Business Guide to NoSQL on the Jakarta EE Platform](#)



sales@payara.fish



UK: +44 800 538 5490
Intl: +1 888 239 8941



www.payara.fish

Payara Services Ltd 2023 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ