How to Use MicroProfile Rest Client with Payara Server

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

Contents

Introduction

The goal of this guide is to help you as a Java developer make the most of the [MicroProfile](#) Rest Client API using Payara Server. The guide starts by looking at what MicroProfile is, the individual APIs that make up MicroProfile and finally takes a deeper look at the REST Client API. By the end of this guide, you will be able to integrate MP into your application and learn how to build reliable and well-functioning applications on Payara Server.
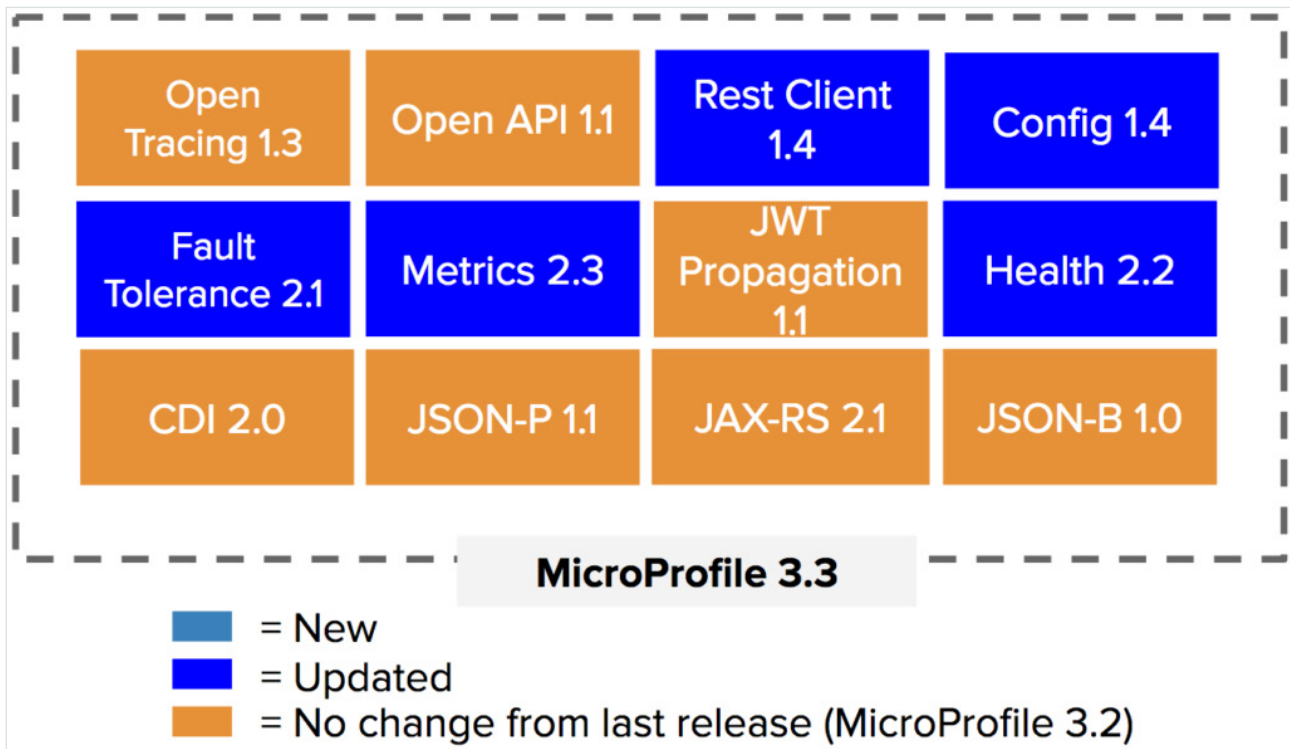
What Do I Need to Use This Guide

You will need a copy of the latest Payara Server release to follow along this guide. The sample code is built using Apache Maven and as such, any IDE that supports Maven can be used. You should also have Java SE 8 installed on your computer.

What is MicroProfile?

MicroProfile is a [community driven](#) initiative that is a collection of abstract specs that form a complete solution to developing cloud native, Java enterprise 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 3.3 which is an incremental release that includes an update to MicroProfile Config 1.4, MicroProfile Fault Tolerance 2.1, MicroProfile Health 2.2, MicroProfile Metrics 2.3, and MicroProfile Rest Client 1.4. Like its previous version, MicroProfile 3.3 continues to align itself with Java EE 8 as the foundational programming model for the development of Java microservices and consists of 12 different specifications as shown below.

Open Tracing 1.3	Open API 1.1	Rest Client 1.4	Config 1.4
Fault Tolerance 2.1	Metrics 2.3	JWT Propagation 1.1	Health 2.2
CDI 2.0	JSON-P 1.1	JAX-RS 2.1	JSON-B 1.0

MicroProfile 3.3

= New
= Updated
= No change from last release (MicroProfile 3.2)

As abstract specifications, the various implementations are free to implement the base specs and add custom features on top. The Payara Server Platform is one of the popular implementations of the MicroProfile spec and adds quite a number of custom features on top of the base specs. This guide will walk you through the Payara Server Platform implementation of the MicroProfile Rest Client API. By the end of this guide, you would have learned how to integrate the Rest Client API into your applications to aid in running in automated cluster management cloud environments.

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.

```
1    <dependency>
2        <groupId>org.eclipse.microprofile</groupId>
3        <artifactId>microprofile</artifactId>
4        <version>3.3</version>
5        <type>pom</type>
6        <scope>provided</scope>
7    </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.

MicroProfile REST Client

What is the MicroProfile Rest Client? The MP REST Client is an API that abstracts the use of the JAX-RS client in the fetching of HTTP resources across the network. To understand better, let us take a simple example of making a REST call the to the freely available REST Countries resource - https://restcountries.eu - to get information about a given country. The full resource path is https://restcountries.eu/api/v2/name/{country} where the country path param is the name of the country for which information we are interested in.

The image below shows a typical invocation using the JAX-RS client that ships with Jakarta EE JAX-RS.

```java
1 public Country fetchCountryInfo(final String countryName) {
2         Client client = ClientBuilder.newClient();
3         Response response = client.target("https://restcountries.eu")
4                 .path("rest/v2")
5                 .path("name")
6                 .path(countryName)
7                 .request()
8                 .accept(MediaType.APPLICATION_JSON)
9                 .get();
10
11         return response.readEntity(Country.class);
12     }
```

Line 1 declares method *fetchCountryInfo* that takes a String as the name of the country whose information we want to fetch from the REST Countries resource. Line 2 creates a new instance of the JAX-RS Client and line 3 uses it to target the requisite resource, passing in the country name as the path-param on line 6. Line 11 returns the Country as read from the entity field on the Response object.

The problem with the above snippet is that it is first, verbose. Secondly, it is not typesafe. The single invocation above would not be much of a problem in small applications. However, in a complex, microservices based application with a substantial number of services that have complex interdependencies, it becomes hard to see at a glance what method from which service is being invoked.

The goal of the MP REST Client API is to allow you to invoke REST resource methods directly. That is, you should be able to invoke resource methods directly on instances of the objects that declare them similar to how you would invoke a method on any Java object.

Your Rest Interface

So how do we convert the above JAX-RS Client invocation on the REST Countries API to a MP REST Client call? Let us start by creating a simple Java interface as shown below.

```java
1 @Path("rest/v2")
2 @Produces(MediaType.APPLICATION_JSON)
3 public interface RestCountriesClient {
4
5     @GET
6     @Path("name/{country}")
7     List<Country> getCountriesByName(@PathParam("country") @NotEmpty String country);
8
9 }
10
```

Line 3 declares interface *RestCountriesClient* with a single method - *getCountriesByName,* as declared on line 7. Line 2 marks this interface as a JAX-RS resource with the *@Path* annotation, giving it a value of "rest/v2." Line 2 declares this resource endpoint a producer of JSON media type. Line 5 uses the *@GET* annotation to declare method *getCountriesByName* as a GET method. Line 6 declares the method as hosted on the path *name/{country}*, where the path param is for setting the country dynamically at runtime.

The type of the List returned by method *getCountriesByName* - Country - is the Java representation of the JSON object returned by the remote REST Countries API. Its sole purpose is for us to have a typesafe representation of the data returned by the remote API.

The *@Produces* annotation in the above interface actually means that the implementation of this interface produces MIME type JSON. This is used by the REST Client runtime to set the Accept HTTP header. You can also declare the *@Consumes* annotation for a resource implementation that takes the JSON MIME type. In this case, the runtime will use that to set the Content-Type of the HTTP header. You can set any MIME type as long as there are respective entity providers available and registered.

By default, in the absence of an explicit MIME type declaration, the MP REST Client runtime defaults to "application/json" for both @Consumes and @Produces. The spec requires MP runtimes to use the JSON-B API as JSON entity provider if the runtime supports it, unless a resource method specifies a JSON-P return type, in which case, JSON- P should be used. Our method returns a Java type, so the JSON-B API will be used under the hood to convert the JSON object returned from the remote API to the Java type Country.

With interface *RestCountriesClient,* we have a Java interface that is a JAX-RS resource. With the MP REST Client, we can invoke the *getCountriesByName* method, passing in the requites arguments directly on an object of the same type as we would any other Java object. Let us see how in the code snippet below.

```java
@Test
void testGet() throws URISyntaxException {
    RestCountriesClient restCountries =
            RestClientBuilder.newBuilder().baseUri(new URI("https://restcountries.eu/"))
                    .build(RestCountriesClient.class);
    List<Country> united = restCountries.getCountriesByName("ghana");
    assertFalse(united.isEmpty());

}
```

The code snippet above uses a simple Junit test to assert our use of the MP REST Client. Line 3 declare a variable of our interface type *RestCountriesClient,* initializing it through static methods on the [RestClientBuilder](#) interface from the MP REST Client API. The implementation of the *RestClientBuilder* interface will be provided by the MP runtime, in our case the Payara server.

This is a basic use of the *RestClientBuilder* interface, where all we do is pass in the base URI and the class type of the JAX-RS resource interface we want to target. In this case, we pass in our *RestCountriesClient* interface.

With our *restCountries* Java instance, we can make normal Java invocations on it as done on line 6 by calling the *getCountriesByName*, passing in the name of "Ghana" as the country whose information we can to retrieve from the remote REST Countries API.

If you observe the use of the *RestCountriesClient* interface on line 6, you notice we passed in the String "Ghana" as the parameter to the method. Of course, the method is just a Java method. However, the resource we are targeting takes a path parameter argument. This is one of the "magic" of the MP REST Client API. The String we passed to the method will be used - automatically - as the path parameter in the resource path name/{country} as declared on the method *getCountriesByName* in the *RestCountriesClient* interface.

Another thing to note is that we have no concrete implementation of the interface anywhere in our code. The MP REST Client does not care where the implementation is. In our case, the implementation is actually the REST Countries API. The implementation could also be in our code or another microservice. It does not matter as long as the MP REST Client can find an implementation, it will be used. This use of the *RestClientBuilder* is the programmatic lookup approach. The other approach is the CDI approach that you will look at in a bit.

RestClientBuilder

The *RestClientBuilder* is the entry point to the creation of a typesafe REST Client. It extends the Configurable interface from JAX-RS, making it easy to register custom JAX-RS components while the REST Client is being built. The Payara Server platform implementation of the MP REST Client supports the following JAX-RS component types:

- ClientRequestFilter
- ClientResponseFilter
- MessageBodyReader
- MessageBodyWriter
- ParamConverter
- ReaderInterceptor
- WriterInterceptor
- ResponseExceptionMapper

To programmatically use the *RestClientBuilder*, you invoke the *newBuilder()* method, which is guaranteed to return a new, non-cached instance of the *RestClientBuilder*. The next method you will most invoke is either the *baseUrl()* or *baseUri()* method. These methods both take a URL or URI object respectively. In our example, we passed in a URI initialized to *https://restcountries.eu/*. This URI is going to be the base URI for making requests to the remote resource.

Specifying the URL/URI

In our case, we specified the base URI as *https://restcountries.eu/*, and our interface is hosted at path "rest/v2," meaning all resource method invocations on the interface will be made relative to the full URI *https://restcountries.eu/rest/v2*. For method *getCountriesByName*, the runtime will resolve the full path to *https://restcountries.eu/rest/v2/name/ghana*, where name is the path to the method and "ghana" is the path parameter substituted at runtime.

Specifying the connectTimeout

You can set the connection timeout by calling the *connectTimeout* method and passing in the maximum time out and the *TimeUnit*. This is similar to setting the JAX-RS *ClientBuilder's connectTimeout()* method. You can pass in any non-negative number for the timeout value. Passing zero means an infinite connect time out. In our example above, we did not specify any.

Specifying the readTimeout

Similar to the *connectTimeout*, you can specify the *readTimeout* on the *RestClientBuilder* during build time. This method *readTimeout* also takes a value for the duration of the read and a *TimeUnit*. A processing exception is thrown if the timeout is reached and the read is not completed yet.

Building the RestClientBuilder

The build method of the *RestClientBuilder* is the final step in the creation instances of REST interfaces. This method takes an interface type that has JAX-RS resource endpoint declarations. In our case, we pass in the *RestCountriesClient interface.* It returns in an instance of the interface that you can invoke methods on as you would any Java object. The build method throws a *RestClientDefinitionException* if the passed in interface is invalid, and an *IllegalStateException* if some prerequisite like the base URL/URI is not set.

There are other methods on the *RestClientBuilder* interface like *executorService()* for setting the *ExecutorService* to be used for async requests. There is also the *sslContext* method for setting the *SSLContext* on the *RestClientObject.*

Registering custom JAX-RS Components

You can register you custom JAX-RS components like *ClientRequest* and *ClientResponse* filters on the *RestClientBuilder* by calling the register method for each component. As stated earlier, the *RestClientBuilder* extends the Configurable interface from JAX-RS, and thus you can call any of the register methods of the Configurable interface to register your custom components.

As an example, let us create a simple JAX-RS *ClientResponseFilter* and register it on our Rest Client.

```java
public class CustomClientRequestFilter implements ClientResponseFilter {
    @Override
    public void filter(ClientRequestContext requestContext,
                       ClientResponseContext responseContext)
      throws IOException {
        System.out.println(requestContext.getProperty("org.eclipse.microprofile.rest.client.invokedMethod"));

    }
}
```

The above code snippet declares class *CustomClientRequestFilter,* which implements the *ClientResponseFilter* from the JAX-RS API. In our implementation of the filter method, we do a console printout of the value of property *org.eclipse.microprofile.rest.client.invokedMethod*. This value of this property is mandated by the REST Client spec to be the *java.lang.reflect.Method* object representing the Rest Client interface method currently being invoked.

Let us register our custom component on the *RestClientBuilder* interface.

```
1    @Test
2    void testGet() throws URISyntaxException {
3
4        RestCountriesClient restCountries =
5                RestClientBuilder.newBuilder().baseUri(new URI("https://restcountries.eu/"))
6                        .register(CustomClientRequestFilter.class)
7                        .build(RestCountriesClient.class);
8        List<Country> united = restCountries.getCountriesByName("united");
9        assertFalse(united.isEmpty());
10
11    }
```

The above code snippet is a modified version of what we looked at previously. The new change is on line 6, where we call the register method on the *RestClientBuilder*, passing in our *CustomClientFilter* class type. That is all we need to do to register our filter. A sample run of the test results in the following printed to the console by the response filter.

```
1 public abstract java.util.List
  fish.payara.boundary.rest.client.RestCountriesClient.getCountriesByName(java.lang.String)
```

The value of the property *invokedMethod* prints out the return type, the name and arguments of the method whose invocation resulted in the dispatch of the client response filter. This is how easy it is to register your custom JAX-RS components with the *RestClientBuilder*.

REST Client and Other MP APIs

REST Client with CDI

We have so far looked at programmatically initializing the *RestClientBuilder* interface, registering custom components. As a part of the MicroProfile group of APIs, the REST Client API has built in support for other APIs like CDI and Config. We can get an instance of our *RestCountriesClient* interface through the CDI runtime using the @Inject annotation.

The code snippet below shows our *RestCountriesClient* interface repurposed to be injected as a REST Client artefact.

```
1 @Path("rest/v2")
2 @Produces(MediaType.APPLICATION_JSON)
3 @RegisterRestClient(baseUri = "https://restcountries.eu/rest/v2")
4 public interface RestCountriesClient {
5
6     @GET
7     @Path("name/{country}")
8     List<Country> getCountriesByName(@PathParam("country") @NotEmpty String country);
9 }
```

The above code snippet is the *RestCountriesClient* we have seen to this point. The only new thing added here is line 3, *@RegisterRestClient*, an annotation from the REST Client API that registers this interface as a RestClient artefact. We passed in one argument - *baseUri* - to the annotation to identify which REST endpoint this particular interface will be targeting. All resource methods in this interface will be invoked relative to the URI passed to the *baseUri* parameter of the *@RegisterRestClient* annotation. In the above code snippet, an invocation of the *getCountriesByName* method with "ghana" as the parameter will result in a GET request being made to the endpoint *https://restcountries.eu/rest/v2/name/ghana*.

With our REST interface registered, we can inject it anywhere as we would any CDI component. For example, the code snippet below shows us injecting our REST interface into a CDI ApplicationScoped service class called Controller.

```
1  @ApplicationScoped
2  public class Controller {
3
4      @Inject
5      @RestClient
6      private RestCountriesClient restClient;
7  }
```

Line 6 of the above code snippet declares a field of type *RestCountriesClient,* annotated with the CDI @Inject annotation on line 4. The field is then Qualified with the *@RestClient* CDI qualifier to explicitly tell the REST Client to resolve this field. The use of the qualifier is optional if there is only one type of bean we are injecting. In situations where you have more than one type of bean being injected, then you would need to use the qualifier to make your intention to the runtime clear.

The field *restClient* as shown above can be used like any other Java object, because the REST Client API spec mandates implementations to provide concrete implementations of the remote service being invoked at runtime.

CDI Registration of components

When using the programmatic creation of the *RestClientBuilder,* we used the register method to register custom JAX-RS components that we wanted to be available at runtime. Using the REST Client API with CDI, we can equally register custom components using annotations as shown below.

```java
1 @Path("rest/v2")
2 @Produces(MediaType.APPLICATION_JSON)
3 @RegisterRestClient(baseUri = "https://restcountries.eu/rest/v2")
4 @RegisterProvider(value = CustomClientRequestFilter.class, priority = Priorities.USER)
5 public interface RestCountriesClient {
6
7     @GET
8     @Path("name/{country}")
9     List<Country> getCountriesByName(@PathParam("country") @NotEmpty String country);
10
11 }
12
```

The above code snippet shows our *RestCountriesClient* interface, annotated with *@RegisterProvider* annotation passing in two parameters. The first one is the class type of the custom component we wish to register, in our case a simple implementation of the *ClientResponseFilter* interface. The second optional parameter is the [priority](#). We can use this parameter to set the JAX-RS priority of this component. That is all that is needed to register custom components through annotations.

The default CDI scope of the RestCountriesClient will be the @Default scope because we haven't explicitly specified any CDI scope. You can specify your resource interfaces as having any valid CDI scope and that will be honored by runtime.

Overriding with MP Config

So far, we have seen how to define a resource interface and inject it for use as a REST Client artefact. The values set for the *@RegisterRestClient* and *@RegisterProvider* annotations are set at compile time. However, using the MicroProfile Config API, it is trivial to override the parameters of these annotations at runtime by setting certain properties in a valid MP [ConfigSource](#). For our examples, we are going to use the microprofile-config.properties file.

We can have a slimmed down version of our REST interface declaration and use the MP Config API to provide the values at runtime.

```
1 @Produces(MediaType.APPLICATION_JSON)
2 @RegisterRestClient
3 public interface RestCountriesClient {
4
5     @GET
6     @Path("name/{country}")
7     List<Country> getCountriesByName(@PathParam("country") @NotEmpty String country);
8 }
9
```

You can override the *baseUri* of the *@RegisterRestClient* annotation by setting the following property to the value of the *baseUri.*

```
1 fish.payara.boundary.rest.client.RestCountriesClient/mp-rest/uri=https://restcountries.eu/rest/v2
```

You could also have use */mp-rest-url property instead. In either case, the value provider should be something that can be parsed to valid object type. In case you provide both the uri and url, the uri takes precedence.

We can also set the CDI scope we want our resource interface to be in by setting the fully qualified class name of any CDI scope as the value to the following property.

```
1 fish.payara.boundary.rest.client.RestCountriesClient/mp-rest/scope=javax.enterprise.context.ApplicationScoped
```

We can register our custom JAX-RS providers by providing a comma separated list of FQN of our components. The property-value pair below registers our *ClientResponseFilter* implementation *CustomClientRequestFilter.*

```
1  fish.payara.boundary.rest.client.RestCountriesClient/mp-
   rest/providers=fish.payara.boundary.rest.boundary.CustomClientRequestFilter
```

We can also set the priority of our custom components by setting a value to the */priority property as shown below.

```
1  fish.payara.boundary.rest.client.RestCountriesClient/mp-
   rest/providers/fish.payara.boundary.rest.boundary.CustomClientRequestFilter/priority=1020
```

Priorities set through the MP Config API take precedence over those set on the component using the @RegisterProvider annotation.

The *connectTimeout* and *readTimout* can equally be set using the following properties, respectively. The default time unit is milliseconds.

```
1  fish.payara.boundary.rest.client.RestCountriesClient/mp-rest/mp-rest/connectTimeout=700
2  fish.payara.boundary.rest.client.RestCountriesClient/mp-rest/mp-rest/readTimeout=700
```

Using Configuration Keys

It is possible to simplify the MP Config keys used to configure REST clients by setting a config key in the *@RegisterClient* annotation that will be used. As an example, let us simplify our use of the MP config key-value pair we have seen so far by setting a common config key.

```
1 @Produces(MediaType.APPLICATION_JSON)
2 @RegisterRestClient(configKey = "restCountries")
3 public interface RestCountriesClient {
4
5     @GET
6     @Path("name/{country}")
7     List<Country> getCountriesByName(@PathParam("country") @NotEmpty String country);
8 }
```

Line 2 of the above code snippet sets the *configKey* parameter of the *@RegisterRestClient* annotation to "restCountries." What this means is that in the MP ConfigSource, we can use *restCountries/mp-rest/ connectTimout* for instance, as the property to set the connect timeout. Setting the *configKey* is a way to replace the use of the FQN of the REST interface as part of the keys. For instance, the snippet below sets the base URI using the config key as defined above.

```
1 restCountries/mp-rest/uri=https://restcountries.eu/rest/v2
```

Async REST Client

REST client methods can be declared to be asynchronous. Such methods be non-blocking and will return a CompletionStage object. Let us create an async method in our *RestCountriesClient*.

```java
1 @Produces(MediaType.APPLICATION_JSON)
2 @RegisterRestClient
3 public interface RestCountriesClient {
4
5     @GET
6     @Path("name/{country}")
7     List<Country> getCountriesByName(@PathParam("country") @NotEmpty String country);
8
9     @GET
10    @Path("region/{region}")
11    CompletionStage<List<Country>> getCountryByRegion(@PathParam("region") @NotEmpty String region);
12 }
```

Line 11 of the above code snippet declares method *getCountryByRegion*, taking a single parameter of type String and returning a List of Country objects, wrapped in a *CompletionStage* object. This return type makes the method async.

By default, the MicroProfile Rest Client implementation can determine how to implement the asynchronous request. The primary requirement for the implementation is that the response from the remote server should be handled asynchronously from the invoking method. Alternatively, you can provide your own *ExecutorService* by passing an instance of it to the *executorService* method on the *RestClientBuilder* instance during build time. The Payara Server platform uses the thread pool of the server by default.

REST Client and Fault Tolerance

The REST Client API spec requires runtimes to ensure that the behavior of most Fault Tolerance annotations are followed. The FT annotations *@Asynchronous, @Bulkhead, @CircuitBreaker, @Fallback*, and *@Retry* should work as expected when used to annotate REST interface methods.

Summary

The MicroProfile REST Client API is a very simple but powerful abstraction over the JAX-RS Client API. The goal, as we have seen in this guide, is to help you make typesafe REST calls through Java interfaces. We looked at how to programmatically build rest client interfaces using the *RestClientBuilder.*

We also looked at how to inject rest interfaces through the use of CDI annotations. We then looked at how to use the MP Config API to provide or override values for the configuration of a REST client interface. The REST Client API spec goes into more detail for all aspects of the API, including SSL configuration and hostname verification.

Conclusion

The MP REST Client is a great way to simply REST calls in your applications. With Payara Server fully implementing the latest MicroProfile specification, you are assured of a powerful platform on which to run your mission critical enterprise Java workload.

Should you need further support or info about using or transitioning your enterprise Java workload to Payara Server, please don't hesitate to get in touch with us. We would love to hear from you. You can also keep in touch with us on our social media platforms

sales@payara.fish   **+44 207 754 0481**   **www.payara.fish**