Typesafe REST Calls With MicroProfile REST Client

MicroProfile 6

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

User Guide

Contents

Guide Updated: **June 2023**

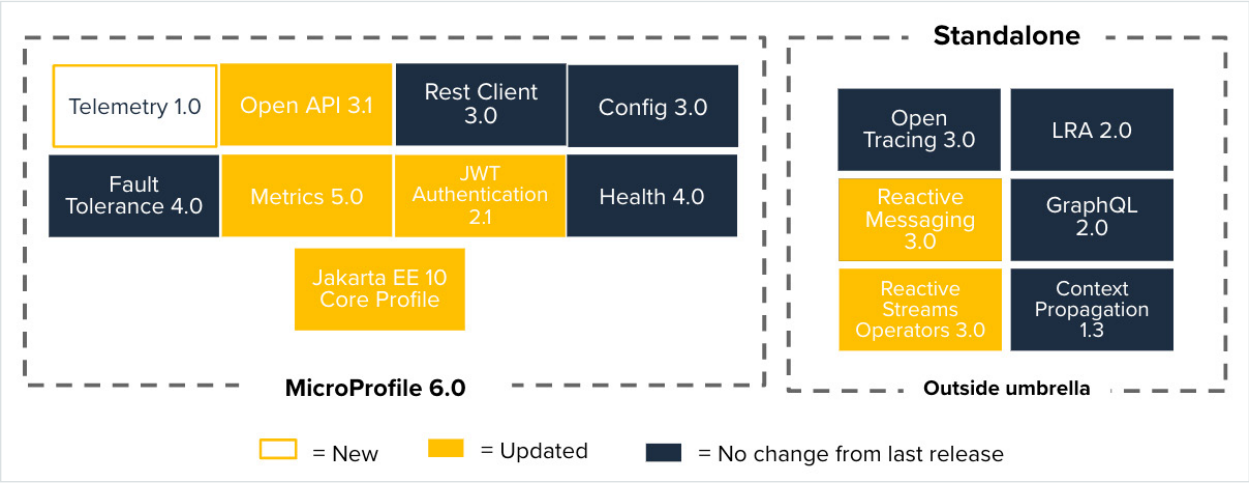
In a cloud native, microservices era, almost all applications need to interface with other services, both internal and external, through the standard HTTP protocol. There are different ways to consume REST services in a Jakarta EE application, from use of the HTTP client in Java SE, to Jakarta REST client to MicroProfile REST client. Almost all other options for consuming RESTful resources in an application can be cumbersome, verbose, and in some cases, require some plumbing.

The MicroProfile REST client, built on top of Jakarta REST, is a typesafe, easy to use client that abstracts you from the low level HTTP infrastructure. This guide will show you how to use this API in your Jakarta EE applications to create much more readable and maintainable applications that consume REST services in a typesafe way.

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. MicroProfile 6.0 is incompatible with versions of Jakarta EE below 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.

MicroProfile REST Client

What is the MicroProfile Rest Client? The MP REST Client is an API that abstracts the use of the Jakarta REST client in the fetching of HTTP resources across the network. To understand better, let us take a simple example of making a REST call to the freely available REST Countries resource - https://restcountries.com - to get information about a given country. The full resource path is https://restcountries.com/v3.1/name/{country} where the country path param is the name of the country for which information we are interested in.

The sample call below shows a typical invocation using the Jakarta REST client that ships with Jakarta EE 10.

```
    public Country fetchCountryInfo(final String countryName) {
        Client client = ClientBuilder.newClient();
        Response response = client.target("https://restcountries.com")
                .path("v3.1")
                .path("name")
                .path(countryName)
                .request()
                .accept(MediaType.APPLICATION_JSON)
                .get();

        return response.readEntity(Country.class);
    }
```

Method `fetchCountryInfo` takes a String as the name of the country whose information we want to fetch from the REST Countries resource. It then creates a new instance of the Jakarta REST Client and uses it to target the requisite resource, passing in the country name as the path-param. The method then 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 Jakarta REST 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.

```
@Path("v3.1")
@Produces(MediaType.APPLICATION_JSON)
public interface RestCountriesClient {

    @GET
    @Path("name/{country}")
    List<Country> getCountriesByName(@PathParam("country") @NotEmpty String
country);

}
```

In the above snippet, we declare interface `RestCountriesClient` with a single method - `getCountriesByName`. The interface is marked as a Jakarta REST resource with the @Path annotation, giving it a value of "rest/3.1", capable of producing JSON type. The interface has method `getCountriesByName`, which takes a path parameter of type string, as the name, or part of it, to search for. So, for instance, you can pass either "Ghana" or "gh". The first one will return data for just that country, and the second one will return data for all country names that have "gh" in them.

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 Jakarta REST resource. With the MP REST Client, we can invoke the `getCountriesByName` method, passing in the requisite 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.

```
    @Test
    void testGet() {
         RestCountriesClient restCountries = RestClientBuilder.newBuilder().
baseUri("https://restcountries.com/")
                .register(CustomClientRequestFilter.class)
                .build(RestCountriesClient.class);

        List<Country> united = restCountries.getCountriesByName("united");
        assertFalse(united.isEmpty());

    }
```

The code snippet above uses a simple JUnit test to assert our use of the MP REST Client. The test declares 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 Jakarta REST 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 above by calling the getCountriesByName, passing in the name of "Ghana" as the country whose information we want to retrieve from the remote REST Countries API.

If you observe the use of the RestCountriesClient interface, 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 part 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 we will look at in a bit.

RestClientBuilder

The `RestClientBuilder` is the entry point to the creation of a typesafe REST Client. It extends the `jakarta.ws.rs.core.Configurable` interface from Jakarta REST, making it easy to register custom Jakarta REST components while the REST Client is being built. The Payara Server platform implementation of the MP REST Client supports the following Jakarta REST 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 likely invoke is either the `baseUrl()` or `baseUri()` method. These methods take a URL or URI object respectively. In our example, we passed in a URI initialized to https://restcountries.com/. 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.com/, and our interface is hosted at path "v3.1," meaning all resource method invocations on the interface will be made relative to the full URI https://restcountries.com/v3.1. For method `getCountriesByName`, the runtime will resolve the full path to https://restcountries.com/v3.1/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 Jakarta REST ClientBuilder's `connectTimeout()` method. You can pass in any non-negative number for the timeout value. Passing zero means an infinite connection 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 of instances of REST interfaces. This method takes an interface type that has Jakarta REST resource endpoint declarations. In our case, we pass in the RestCountriesClient interface. It returns 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 Jakarta REST Components

You can register your custom Jakarta REST 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 Jakarta REST, 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 Jakarta REST `ClientResponseFilter` and register it on our Rest Client.

```
public class CustomClientResponseFilter 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 `CustomClientResponseFilter`, which implements the `ClientResponseFilter` from the Jakarta REST API. In our implementation of the filter method, we do a console printout of the value of property `org.eclipse.microprofile.rest.client.invokedMethod`. The 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.

```
   RestCountriesClient restCountries =
                RestClientBuilder.newBuilder().baseUri(apiUri)
                        .register(CustomClientResponseFilter.class)
                        .build(RestCountriesClient.class)
```

The above code snippet is a modified version of what we looked at previously. The new change is 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 Jakarta REST 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 and registering custom components. As a part of the MicroProfile group of APIs, the REST Client API has built in support for other APIs like Jakarta CDI and Config. We can get an instance of our RestCountriesClient interface through the use of `@Inject` annotation.

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

```
@Path("v3.1")
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient(baseUri = "https://restcountries.com/v3.1")
public interface RestCountriesClient {

    @GET
    @Path("name/{country}")
    List<Country> getCountriesByName(@PathParam("country") @NotEmpty String
country);
}
```

The above code snippet is the RestCountriesClient we have seen to this point. The only new thing added here is the line `@RegisterRestClient`, an annotation from the REST Client API that registers this interface as a RestClient artifact. 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.com/v3.1/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.

```
@ApplicationScoped
public class Controller {

    @Inject
    @RestClient
    private RestCountriesClient restClient;
}
```

The above code snippet declares a field of type RestCountriesClient, annotated with the CDI `@Inject` annotation. 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 Jakarta REST 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.

```
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient(baseUri = "https://restcountries.com/v3.1")
@RegisterProvider(value = CustomClientResponseFilter.class, priority =
Priorities.USER)
public interface RestCountriesClient {

    @GET
    @Path("name/{country}")
    List<Country> getCountriesByName(@PathParam("country") @NotEmpty String
country);
    }
```

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 Jakarta REST 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.

Overriding with MP Config

So far, we have seen how to define a resource interface and inject it for use as a REST Client artifact. 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.

```
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient
public interface RestCountriesClient {

@GET
@Path("name/{country}")
List<Country> getCountriesByName(@PathParam("country") @NotEmpty String
country);
}
```

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

```
fish.payara.boundary.rest.client.RestCountriesClient/mp-rest/uri=https://
restcountries.com/v3.1.
```

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.

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

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

```
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.

```
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.

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

```
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient(configKey= “restCountries”)
public interface RestCountriesClient {

    @GET
    @Path("name/{country}")
    List<Country> getCountriesByName(@PathParam("country") @NotEmpty String
country);
}
```

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.

```
restCountries/mp-rest/uri=https://restcountries.com/v3.1.
```

Async REST Client

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

```
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient
public interface RestCountriesClient {

    @GET
    @Path("name/{country}")
    List<Country> getCountriesByName(@PathParam("country") @NotEmpty String
country);

    @GET
    @Path("region/{region}")
    CompletionStage<List<Country>> getCountryByRegion(@PathParam("region")
@NotEmpty String region);
}
```

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 Jakarta REST 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 - Twitter, YouTube, GitHub.

If you found this guide useful, investigate the others in our MicroProfile series:

- Keeping Count in Jakarta EE Applications with MicroProfile Metrics
- Jakarta EE Application Health Check With MicroProfile Health
- Effortless Application Configuration with MicroProfile Config

sales@payara.fish

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

www.payara.fish