Rest Client for MicroProfile

John D. Ament, Andy McCright

1.1-RC1, April 17, 2018

Table of Contents

Specification: Rest Client for MicroProfile

Version: 1.1-RC1

Status: Draft

Release: April 17, 2018

Microprofile Rest Client

MicroProfile Rest Client Definition Examples

MicroProfile TypeSafe Rest Clients are defined as Java interfaces.

Sample Definitions

```
public interface MyServiceClient {
    @GET
    @Path("/greet")
    Response greet();
}
```

This simple API exposes one API call, located at `/greet` from the base URL of the client. Invoking this endpoint returns a `javax.ws.rs.core.Response` object that represents the raw response from invoking the API. Below is a more comprehensive example of a client.

```
@Path("/users")
@Produces("application/json")
@Consumes("application/json")
public interface UsersClient {
    @OPTIONS
    Response options();

    @HEAD
    Response head();

    @GET
    List<User> getUsers();

    @GET
    @Path("/{userId}")
    User getUser(@PathParam("userId") String userId);

    @HEAD
    @Path("/{userId}")
    Response headUser(@PathParam("userId") String userId);

    @POST
    Response createUser(@HeaderParam("Authorization") String authorization, User user
);

    @PUT
    @Path("/{userId}")
    Response updateUser(@BeanParam PutUser putUser, User user);

    @DELETE
    @Path("/{userId}")
    Response deleteUser(@CookieParam("AuthToken") String authorization, @PathParam(
"userId") String userId);
}

public class PutUser {
    @HeaderParam("Authorization")
    private String authorization;
    @PathParam("userId")
    private String userId;
    // getters, setters, constructors omitted
}
```

All built in HTTP methods are supported by the client API. Likewise, all base parameter types (query, cookie, matrix, path, form and bean) are supported. If you only need to inspect the body, you can provide a POJO can be processed by the underlying `MessageBodyReader` or `MessageBodyWriter`. Otherwise, you can receive the entire `Response` object for parsing the body and header information from the server invocation.

Invalid Client Interface Examples

Invalid client interfaces will result in a RestClientDefinitionException (which may be wrapped in a `DefinitionException` if using CDI). Invalid interfaces can include:

- Using multiple HTTP method annotations on the same method

A client interface method may contain, at most, one HTTP method annotation (such as `javax.ws.rs.GET`, `javax.ws.rs.PUT`, `javax.ws.rs.OPTIONS`, etc.). If a method is annotated with more than one HTTP method, the implementation must throw a `RestClientDefinitionException`.

```
public interface MultipleVerbsClient {
    @GET
    @DELETE
    Response ambiguousClientMethod()
}
```

- Invalid URI templates

A client interface that accepts parameters based on the URI path must ensure that the path parameter is defined correctly in the `@Path` annotation. For example:

```
@Path("/somePath/{someParam}")
public interface GoodInterfaceOne {
    @DELETE
    public Response deleteEntry(@PathParam("someParam") String entryNameToDelete);
}

@Path("/someOtherPath")
public interface GoodInterfaceTwo {
    @HEAD
    @Path("/{someOtherParam}")
    public Response quickCheck(@PathParam("someOtherParam") String entryNameToCheck);
}
```

Both of these interfaces show valid usage of the `@PathParam` annotation. In `GoodInterfaceOne`, the URI template is specified at the class-level `@Path` annotation; in `GoodInterfaceTwo`, the template is specified at the method-level.

Implementations must throw a RestClientDefinitionException if a `@Path` annotation specifies an unresolved URI template or if a `@PathParam` annotations specifies a template that is not specified in a `@Path` annotation on the enclosing method or interface. For example, the following three interfaces will result in a RestClientDefinitionException:

```
@Path("/somePath/{someParam}")
public interface BadInterfaceOne {
    @DELETE
    public Response deleteEntry();
}

@Path("/someOtherPath")
public interface BadInterfaceTwo {
    @HEAD
    @Path("/abc")
    public Response quickCheck(@PathParam("someOtherParam") String entryNameToCheck);
}

@Path("/yetAnotherPath")
public interface BadInterfaceThree {
    @GET
    @Path("/{someOtherParam}")
    public Response quickCheck(@PathParam("notTheSameParam") String entryNameToCheck);
}
```

`BadInterfaceOne` declares a URI template named "someParam" but the `deleteEntry` method does not specify a `@PathParam("someParam")` annotation. `BadInterfaceTwo` does not declare a URI template, but the `quickCheck` method specifies a `@PathParam` annotation on a parameter. `BadInterfaceThree` has a mismatch. The `@Path` annotation declares a URI template named "someOtherParam" but the `@PathParam` annotation specifies a template named "notTheSameParam". All three interfaces will result in a RestClientDefinitionException.

MicroProfile Rest Client Programmatic Lookup

Type Safe Rest Clients support both programmatic look up and CDI injection approaches for usage. An implementation of MicroProfile Rest Client is expected to support both use cases.

Sample Builder Usage

```
public class SomeService {
   public Response doWorkAgainstApi(URI apiUri, ApiModel apiModel) {
       RemoteApi remoteApi = RestClientBuilder.newBuilder()
            .baseUri(apiUri)
            .build(RemoteApi.class);
       return remoteApi.execute(apiModel);
   }
}
```

Specifying the `baseUri` is the URL to the remote service. The `build` method takes an interface that defines one or more API methods to be invoked, returning back an instance of that interface that can be used to perform API calls.

MicroProfile Rest Client Provider Registration

The RestClientBuilder interface extends the `Configurable` interface from JAX-RS, allowing a user to register custom providers while its being built. The behavior of the providers supported is defined by the JAX-RS Client API specification. Below is a list of provider types expected to be supported by an implementation:

ClientResponseFilter

Filters of type `ClientResponseFilter` are invoked in order when a response is received from a remote service.

ClientRequestFilter

Filters of type `ClientRequestFilter` are invoked in order when a request is made to a remote service.

MessageBodyReader

The `MessageBodyReader` interface defined by JAX-RS allows the entity to be read from the API response after invocation.

MessageBodyWriter

The `MessageBodyWriter` interface defined by JAX-RS allows a request body to be written in the request for `@POST`, `@PUT` operations, as well as other HTTP methods that support bodies.

ParamConverter

The `ParamConverter` interface defined by JAX-RS allows a parameter in a resource method to be converted to a format to be used in a request or a response.

ReaderInterceptor

The `ReaderInterceptor` interface is a listener for when a read occurs against the response received from a remote service call.

WriterInterceptor

The `WriterInterceptor` interface is a listener for when a write occurs to the stream to be sent on the remote service invocation.

ResponseExceptionMapper

The `ResponseExceptionMapper` is specific to MicroProfile Rest Client. This mapper will take a `Response` object retrieved via an invocation of a client and convert it to a `Throwable`, if applicable. The runtime should scan all of the registered mappers, sort them ascending based on `getPriority()`, find the ones that can handle the given status code and response headers, and invoke them. The first one discovered where `toThrowable` returns a non-null `Throwable` that can be thrown given the client method's signature will be thrown by the runtime.

How to Implement ResponseExceptionMapper

The specification provides default methods for `getPriority()` and `handles(int status, MultivaluedMap<String,Object> headers)` methods. Priority is meant to be derived via a `@Priority` annotation added to the `ResponseExceptionMapper` implementation. The runtime will sort ascending, taking the one with the lowest numeric value first to check if it can handle the `Response` object based on it's status code and headers. The usage of ascending sorting is done to be consistent with JAX-RS behavior.

Likewise, the `handles` method by default will handle any response status code >= 400. You may override this behavior if you so choose to handle other response codes (both a smaller ranger and a larger range are expected) or base the decision on the response headers.

The `toThrowable(Response)` method actually does the conversion work. This method should not raise any `Throwable`, instead just return a `Throwable` if it can. This method may return `null` if no throwable should be raised. If this method returns a non-null throwable that is a sub-class of RuntimeException or Error (i.e. unchecked throwables), then this exception will be thrown to the client. Otherwise, the (checked) exception will only be thrown to the client if the client method declares that it throws that type of exception (or a super-class). For example, assume there is a client interface like this:

```
@Path("/")
public interface SomeService {
   @GET
   public String get() throws SomeException;

   @PUT
   public String put(String someValue);
}
```

and assume that the following ResponseExceptionMapper has been registered:

```
public class MyResponseExceptionMapper implements ResponseExceptionMapper
<SomeException> {

   @Override
   public SomeException toThrowable(Response response) {
       return new SomeException();
   }
}
```

In this case, if the `get` method results in an exception (response status code of 400 or higher), SomeException will be thrown. If the `put` method results in an exception, SomeException will not be thrown because the method does not declare that it throws SomeException. If another ResponseExceptionMapper (such as the default mapper, see below) is registered that returns a subclass of RuntimeException or Error, then that exception will be thrown.

Any methods that read the response body as a stream must ensure that they reset the stream.

Provider Declaration

In addition to defining providers via the client definition, interfaces may use the `@RegisterProvider` annotation to define classes to be registered as providers in addition to providers registered via the `RestClientBuilder`

Providers may also be registered by implementing the `RestClientBuilderListener` interface. This interface is intended as an SPI to allow global provider registration. The implementation of this interface must be specified in a `META-INF/services/org.eclipse.microprofile.rest.client.spi.RestClientBuilderListener` file, following the service loader pattern.

Provider Priority

Providers may be registered via both annotations and the builder pattern. Providers registered via a builder will take precedence over the `@RegisterProvider` annotation. The `@RegisterProvider` annotation takes precedence over the `@Priority` annotation on the class.

Provider priorities can be overridden using the various `register` methods on `Configurable`, which can take a provider class, provider instance as well as priority and mappings of those priorities.

Feature Registration

If the type of provider registered is a `Feature`, then the priority set by that `Feature` will be a part of the builder as well. Implementations must maintain the overall priority of registered providers, regardless of how they are registered. A `Feature` will be used to register additional providers at runtime, and may be registered via `@RegisterProvider`, configuration or via `RestClientBuilder`. A `Feature` will be executed immediately, as a result its priority is not taken into account (features are always executed).

Automatic Provider Registration

Implementations may provide any number of providers registered automatically, but the following providers must be registered by the runtime.

JSON-P Provider

When an interface is registered that contains:

- `@Produces("*/json")` or
- `@Consumes("*/json")` or
- a method that declares input or output of type `javax.json.JsonValue` or any subclass therein

Then a JSON-P `MessageBodyReader` and `MessageBodyWriter` will be registered automatically by the implementation. This is in alignment with the JAX-RS 2.0 specification. The provider registered will have a priority of `Integer.MAX_VALUE`, allowing a user to register a custom provider to be used instead.

Default Message Body Readers and Writers

For the following types, and any media type, the runtime must support `MessageBodyReader`s and `MessageBodyWriter`s being automatically registered.

- `byte[]`
- `String`
- `InputStream`
- `Reader`
- `File`

Values supported with `text/plain`

The following types are supported for automatic conversion, only when the media type is `text/plain`.

- `Number`
- `Character` and `char`
- `Long` and `long`
- `Integer` and `int`
- `Double` and `double`
- `Float` and `float`
- `Boolean` and `boolean` (literal value of `true` and `false` only)

Default ResponseExceptionMapper

Each implementation will provide out of the box a `ResponseExceptionMapper` implementation that will map the response into a `WebApplicationException` whenever the response status code is >= 400. It has a priority of `Integer.MAX_VALUE`. It is meant to be used as a fall back whenever an error is encountered. This mapper will be registered by default to all client interfaces.

This behavior can be disabled by adding a configuration property `microprofile.rest.client.disable.default.mapper` with value `true` that will be resolved as a `boolean` via MicroProfile Config.

It can also be disabled on a per client basis by using the same property when building the client, `RestClientBuilder.newBuilder().property("microprofile.rest.client.disable.default.mapper",true)`

MicroProfile Rest Client CDI Support

Rest Client interfaces may be injected as CDI beans. The runtime must create a CDI bean for each interface annotated with `RegisterRestClient`. The bean created will include a qualifier `@RestClient` to differentiate the use as an API call against any other beans registered of the same type. Based on the rules of how CDI resolves bean, you are only required to use the qualifier if you have multiple beans of the same type. Any injection point or programmatic look up that uses the qualifier `RestClient` is expected to be resolved by the MicroProfile Rest Client runtime. Below is an example of said interface, with its matching injection point:

```
package com.mycompany.remoteServices;

@RegisterRestClient
public interface MyServiceClient {
    @GET
    @Path("/greet")
    Response greet();
}
```

```
@ApplicationScoped
public class MyService {
    @Inject
    @RestClient
    private MyServiceClient client;
}
```

Likewise, a user can perform programmatic look up of the interface. Here is one example, but any CDI look up should work:

```
@ApplicationScoped
public class MyService {
    public void execute() {
        MyServiceClient client = CDI.current().select(MyServiceClient.class,
RestClient.LITERAL).get();
    }
}
```

The qualifier is used to differentiate use cases of the interface that are managed by this runtime, versus use cases that may be managed by other runtimes.

Interfaces are assumed to have a scope of `@Dependent` unless there is another scope defined on the interface. Implementations are expected to support all of the built in scopes for a bean. Support for custom registered scopes should work, but is not guaranteed.

Support for MicroProfile Config

For CDI defined interfaces, it is possible to use MicroProfile Config properties to define additional behaviors of the rest interface. Assuming this interface:

```
package com.mycompany.remoteServices;

public interface MyServiceClient {
    @GET
    @Path("/greet")
    Response greet();
}
```

The values of the following properties will be provided via MicroProfile Config:

- `com.mycompany.remoteServices.MyServiceClient/mp-rest/url`: The base URL to use for this service, the equivalent of the `baseUrl` method. This property (or */mp-rest/uri) is considered required, however implementations may have other ways to define these URLs/URIs.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/uri`: The base URI to use for this service, the equivalent of the `baseUri` method. This property (or */mp-rest/url) is considered required, however implementations may have other ways to define these URLs/URIs.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/scope`: The fully qualified classname to a CDI scope to use for injection, defaults to `javax.enterprise.context.Dependent` as mentioned above.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/providers`: A comma separated list of fully-qualified provider classnames to include in the client, the equivalent of the `register` method or the `@RegisterProvider` annotation.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/providers/com.mycompany.MyProvider/priority` will override the priority of the provider for this interface.

Implementations may support other custom properties registered in similar fashions or other ways.

The `url` property must resolve to a value that can be parsed by the `URL` converter required by the MicroProfile Config spec. Likewise, the `uri` property must resolve to a value that can be parsed by the `URI` converter. If both the `url` and `uri` properties are declared, then the `uri` property will take precedence.

The `providers` property is not aggregated, the value will be read from the highest property `ConfigSource`

MicroProfile Rest Client Asynchronous Support

It is possible for Rest Client interface methods to be declared asynchronous. This allows the thread invoking the interface method to proceed while the RESTful request occurs on another thread.

Asynchronous Methods

A method is considered to be asynchronous if the method's return type is `java.util.concurrent.CompletionStage`.

For example, the following methods would be declared asynchronous:

```
public interface MyAsyncClient {
    @GET
    @Path("/one")
    CompletionStage<Response> get();

    @POST
    @Path("/two")
    CompletionStage<String> post(String entity);
}
```

ExecutorService

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 on a different thread than the thread invoking the asynchronous method. Providers on the outbound client request chain (such as `ClientRequestFilter`s, `MessageBodyWriter`s, `WriterInterceptor`s, etc.) may be executed on either thread.

Callers may override the default implementation by providing their own `ExecutorService` via the `RestClientBuilder.executorService(ExecutorService)` method. The implementation must use the `ExecutorService` provided for all asynchronous methods on any interface built via the `RestClientBuilder`.

AsyncInvocationInterceptors

There may be cases where it is necessary for client application code or runtime components to be notified when control of the client request/response has flowed from one thread to another. This can be accomplished by registering an implementation of the AsyncInvocationInterceptorFactory provider interface. MP Rest Client implementations must invoke the `newInterceptor` method of each registered factory provider prior to switching thread of execution on async method requests. That method will return an instance of `AsyncInvocationInterceptor` - the MP Rest Client implementation

must then invoke the `prepareContext` method while still executing on the thread that invoked the async method. After swapping threads, but before invoking further providers or returning control back to the async method caller, the MP Rest Client implementation must invoke the `applyContext` method on the new async thread that will complete the request/response.

The following example shows how the AsyncInvocationInterceptorFactory provider and associated AsyncInvocationInterceptor interface could be used to propagate a `ThreadLocal` value from the originating thread to async thread:

```
public class MyFactory implements AsyncInvocationInterceptorFactory {

    public AsyncInvocationInterceptor newInterceptor() {
         return new MyInterceptor();
    }
}

public class MyInterceptor implements AsyncInvocationInterceptor {
    // This field is temporary storage to facilitate copying a ThreadLocal value
    // from the originating thread to the new async thread.
    private volatile String someValue;

    public void prepareContext() {
        someValue = SomeClass.getValueFromThreadLocal();
    }
    public void applyContext() {
        SomeClass.setValueIntoThreadLocal(someValue);
    }
}

@RegisterProvider(MyFactory.class)
public interface MyAsyncClient {...}
```

Release Notes for MicroProfile Rest CLient 1.1

Changes since 1.0:

- New `baseUri` method on `RestClientBuilder`.

Release Notes for MicroProfile Rest Client 1.0

MicroProfile Rest Client Spec PDF MicroProfile Rest Client Spec HTML MicroProfile Rest Client Spec Javadocs

Key features:

- Built in alignment to other MicroProfile Specs - automatic registration of JSON provider, CDI support for injecting clients, fully configurable clients via MicroProfile Config
- Can map JAX-RS `Response` objects into `Exception`s to be handled by your client code
- Fully declarative annotation driven configuration, with supported builder patterns
- Closely aligned to JAX-RS with configuration and behavior based on the JAX-RS `Client` object

To get started, simply add this dependency to your project, assuming you have an implementation available:

```
<dependency>
    <groupId>org.eclipse.microprofile.rest.client</groupId>
    <artifactId>microprofile-rest-client-api</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
</dependency>
```

And then programmatically create an interface:

```
public interface SimpleGetApi {
    @GET
    Response executeGet();
}
// in your client code
SimpleGetApi simpleGetApi = RestClientBuilder.newBuilder()
            .baseUri(getApplicationUri())
            .build(SimpleGetApi.class);
```

or you can use CDI to inject it:

```
@Path("/")
@Dependent
@RegisterRestClient
public interface SimpleGetApi {
    @GET
    Response executeGet();
}
// in your client code
@Inject
private SimpleGetApi simpleGetApi
// in your config source
com.mycompany.myapp.client.SimpleGetApi/mp-rest/url=http://microprofile.io
```