Rest Client for MicroProfile

John D. Ament

1.0-T9, December 05, 2017

Table of Contents

Specification: Rest Client for MicroProfile

Version: 1.0-T9

Status: Draft

Release: December 05, 2017

Microprofile Rest Client

MicroProfile Rest Client Definition Examples

MicroProfile TypeSafe Rest Clients are defined as Java interfaces.

Sample Definitions

```java
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.

```java
@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.

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

```java
public class SomeService {
    public Response doWorkAgainstApi(URL apiUrl, ApiModel apiModel) {
        RemoteApi remoteApi = RestClientBuilder.newBuilder()
                .baseUrl(apiUrl)
                .build(RemoteApi.class);
        return remoteApi.execute(apiModel);
    }
}
```

Specifying the `baseUrl` is the URL to the remote service. The `proxy` 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:

```java
@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:

```java
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 @RegisterProviders annotation to define classes to be registered as providers in addition to providers registered via the RestClientBuilder

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:

```java
package com.mycompany.remoteServices;

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

```java
@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:

```java
@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 is considered required, however implementations may have other ways to define these URLs.

- `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 `@RegisterProviders` annotation.

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.

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

Release Notes for MicroProfile Rest Client 1.0