Rest Client for MicroProfile

John D. Ament, Andy McCright, Ken Finnigan, Michal Szynkiewicz

2.0-RC3, November 16, 2020

Table of Contents

Specification: Rest Client for MicroProfile

Version: 2.0-RC3

Status: Draft

Release: November 16, 2020

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.

Users may specify the media (MIME) type of the outbound request using the `@Consumes` annotation (this determine the `Content-Type` HTTP header), and the expected media type(s) of the response by using the `@Produces` annotation (the `Accept` HTTP header). This indicates that the client interface expects that the remote *service* consumes/produces the specified types. If no `@Consumes` or `@Produces` annotation is specified for a given request, it will default to `javax.ws.rs.core.MediaType.APPLICATION_JSON` ("application/json").

Specifying Additional Client Headers

While it is always possible to add a `@HeaderParam`-annotated method argument to specify headers, some times that does not make sense within the context of the application. For example, you may want to specify a username/password in the `Authorization` header to a secure remote service, but you may not want to have a `String authHeader` parameter in the client interface method. The `@ClientHeaderParam` annotation can allow users to specify HTTP headers that should be sent without altering the client interface method signature.

The annotation contains three attributes: `name`, `value`. and `required`. The `name` attribute is used to specify the header name. The `value` attribute is used to specify the value(s) of the header. The value can be specified explicitly or it can reference a method that would compute the value of the header - in this latter case, the compute method name must be surrounded by curly braces. The compute method must be either a default method on the interface or a public static method that is accessible to the interface, must return a `String` or `String[]` and must either contain no arguments or contain a single `String` argument - the implementation will use this `String` argument to pass the name of the header. When specifying a compute method as the value attribute, only one method may be specified - if more than one string is specified as the value attribute, and one of the strings is a compute method (surrounded by curly braces), then the implementation will throw a RestClientDefinitionException. The `required` attribute determines what should happen in the event that the compute method throws an exception. **Note: If the required attribute is set to true (default), then the client request will fail if the compute method throws an exception.** If it is set to false and the compute method throws an exception, then the client request will continue, but without sending the HTTP header.

Note that if a `@ClientHeaderParam` annotation on a method specifies the same header name as an annotation on the client interface, the annotation on the method will take precedence. Likewise, if the same header name is used in a `@HeaderParam` annotation on a client interface method parameter or in a bean class when a `@BeanParam` annotation is on a client interface method parameter, the value of the `@HeaderParam` annotation takes precedence over any value specified in the `@ClientHeaderParam`. It is invalid for the same header name to be specified in two different `@ClientHeaderParam` annotations on the same target - in this case, the implementation will throw a `RestClientDefinitionException.`

Here are a few examples:

```
@Path("/somePath")
public interface MyClient {

    @POST
    @ClientHeaderParam(name="X-Http-Method-Override", value="PUT")
    Response sentPUTviaPOST(MyEntity entity);

    @POST
    @ClientHeaderParam(name="X-Request-ID", value="{generateRequestId}")
    Response postWithRequestId(MyEntity entity);

    @GET
    @ClientHeaderParam(name="CustomHeader",
                       value="{some.pkg.MyHeaderGenerator.generateCustomHeader}",
                       required=false)
    Response getWithoutCustomHeader();

    default String generateRequestId() {
        return UUID.randomUUID().toString();
    }
}

public class MyHeaderGenerator {
    public static String generateCustomHeader(String headerName) {
        if ("CustomHeader".equals(headerName)) {
            throw UnsupportedOperationException();
        }
        return "SomeValue";
    }
}

@Path("/someOtherPath")
@ClientHeaderParam(name="CustomHeader", value="value1")
@ClientHeaderParam(name="CustomHeader", value="{generateCustomHeader}")
// will throw a RestClientDefinitionException at build time
public interface MyInvalidClient {
...
}
```

It is also possible to add or propagate headers en masse using a `ClientHeadersFactory`. This interface has a single method and takes two read-only `MultivaluedMap` parameters: The first map represents headers for the incoming request - if the client is executing in a JAX-RS environment then this map will contain headers from the inbound JAX-RS request. The second map represents the headers to be sent, and it contains headers that have been specified via `@ClientHeaderParam`, `@HeaderParam`, `@BeanParam`, etc. The method should return a `MultivaluedMap` containing the "final" map of headers to be sent to the outbound processing flow. Providers such as filters, interceptors, message body writers, etc. could still modify the final map of headers prior to sending the HTTP request.

By default, no `ClientHeadersFactory` implementation is used. To enable a `ClientHeadersFactory`, the

client interface must be annotated with the `@RegisterClientHeaders` annotation. If this annotation specifies a value, the client implementation must invoke an instance of the specified `ClientHeadersFactory` implementation class. If no value is specified, then the client implementation must invoke the `DefaultClientHeadersFactoryImpl`. This default factory will propagate specified headers from the inbound JAX-RS request to the outbound request - these headers are specified with a comma-separated list using the following MicroProfile Config property:

`org.eclipse.microprofile.rest.client.propagateHeaders`

Following Redirect Responses

By default, a Rest Client instance will not automatically follow redirect responses. Redirect responses are typically responses with status codes in the 300 range and include `Location` header that indicates the URL of the redirected resource.

To enable a client instance to automatically follow redirect responses, the builder must be configured using the `followRedirects(true)` method. For example:

```
RedirectClient client = RestClientBuilder.newBuilder()
                                         .baseUri(someUri)
                                         .followRedirects(true)
                                         .build(RedirectClient.class);
```

Alternatively, if the client is instantiated and injected using CDI, then it can be configured to follow redirect responses using the `<client_interface_name>/mp-rest/followRedirects` MP Config property. See Support for MicroProfile Config for more details.

Using HTTP Proxy Servers

In some environments it may be necessary to route requests through a HTTP proxy server to reach the REST endpoint. Users may configure the proxy server's address (hostname and port) using the `proxyAddress` method on `RestClientBuilder`. For example:

```
ProxiedClient client = RestClientBuilder.newBuilder()
                                        .baseUri(someUri)
                                        .proxyAddress("myproxy.mycompany.com", 8080)
                                        .build(ProxiedClient.class);
```

Alternatively, if the client is instantiated and injected using CDI, then the proxy address can be configured using the `<client_interface_name>/mp-rest/proxyAddress` MP Config property. See Support for MicroProfile Config for more details.

Specifying Query Parameter Style for multi-valued parameters

Different RESTful services may require different styles of query parameters when handling multiple values for the same query parameter. For example, some servers will require query parameters to be expanded into multiple key/value pairs such as `key=value1&key=value2&key=value3`. Others will require parameters to be separated by comma with a single key/value pair such as `key=value1,value2,value3`. Still others will require an array-like syntax using multiple key/value pairs such as `key[]=value1&key[]=value2&key[]=value3`.

The `queryParamStyle(…)` method in the `RestClientBuilder` can be used to specify the desired format of query parameters when multiple values are sent for the same parameter. This method uses the `QueryParamStyle` enum. Here is an example:

```
public interface QueryClient {
    Response sendMultiValues(@QueryParam("myParam") List<String> values);
}
```

```
QueryClient client = RestClientBuilder.newBuilder()
                                      .baseUri(someUri)
                                      .queryParamStyle(QueryParamStyle.
COMMA_SEPARATED)
                                      .build(QueryClient.class);
Response response = client.sendMultiValues(Collections.asList("abc", "mno", "xyz"));
```

This should send a request with a query segment of `myParam=abc,mno,xyz`.

Alternatively, if the client is instantiated and injected using CDI, then the query parameter style can be configured using the `<client_interface_name>/mp-rest/queryParamStyle` MP Config property. See Support for MicroProfile Config for more details.

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.

As previously mentioned, specifying the same header name in multiple `@ClientHeaderParam` annotations on the same target will result in a RestClientDefinitionException. Likewise, specifying multiple compute methods in the `@ClientHeaderParam` value attribute 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.

Both the `ClientRequestFilter` and `ClientResponseFilter` interfaces contains methods that pass an instance of `ClientRequestContext`. The Rest Client implementation must provide a property via that `ClientRequestContext` called `org.eclipse.microprofile.rest.client.invokedMethod` - the value of this property should be the `java.lang.reflect.Method` object representing the Rest Client interface method currently being invoked.

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` or `RestClientListener` interfaces. These interfaces are intended as SPIs to allow global provider registration. The implementation of these interface must be specified in a `META-INF/services/org.eclipse.microprofile.rest.client.spi.RestClientBuilderListener` or `META-INF/services/org.eclipse.microprofile.rest.client.spi.RestClientListener` file, respectively, following the `ServiceLoader` pattern.

CDI Managed Providers

If CDI is available in the implementation's runtime environment, and CDI is managing the lifecycle of a registered provider class, the implementation must use the CDI-managed instance of the provider. This does not apply when an instance is registered, nor does it apply when a class is registered but no instance of that class is managed by CDI.

The following example shows cases where a CDI-managed provider instance must be used:

```
@ApplicationScoped // makes it a CDI bean
public class MyFilter implements ClientRequestFilter {
    @Inject SomeOtherCdiObject obj;
    // ...
}

@RegisterRestClient
@RegisterProvider(MyFilter.class)
public interface MyRestClient1 { /* ... */ }

@RegisterRestClient
// MP Config property: com.mycompany.MyRestClient2/mp-
rest/providers=com.mycompany.MyFilter
public interface MyRestClient2 { /* ... */ }

public interface MyRestClient3 { /* ... */ }

public class Client3Builder {

    public MyClient3 createClient3() {
        return RestClientBuilder.baseUri(someUri).register(MyFilter.class).build
(MyClient3.class);
    }
}
```

When registering `Features`, it should not matter whether the feature itself is managed by CDI or not,

but the implementation should use CDI-managed instances of classes registered by the feature. For example:

```
public class MyFeature implements Feature {
    @Override
    public boolean configure(FeatureContext context) {
        context.register(MyFilter.class); // will be managed by CDI
        context.register(new MyOtherFilter()); // will not be managed by CDI
        return true;
    }
}

@RegisterRestClient
@RegisterProvider(MyFeature.class)
public interface MyRestClient4 { /* ... */ }
```

For more information on integration with CDI, see MicroProfile Rest Client CDI Support for more details.

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 and JSON-B Providers

Implementations of the MicroProfile Rest Client should behave similar to JAX-RS implementations with regard to built-in JSON-P and JSON-B providers. Implementations must provide a built-in JSON-P entity provider. If the implementation supports JSON-B, then it must also provide a built-in JSON-B entity provider. Note that the JSON-B provider should take precedence over the JSON-P provider unless the client interface method's entity parameter or return type is a JSON-P object type (`javax.json.JsonObject`, `javax.json.JsonArray`, etc.).

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 (JSON-P only) or
- no `@Produces` or `@Consumes`

Then a JSON-B or JSON-P `MessageBodyReader` and `MessageBodyWriter` will be registered automatically by the implementation. This is in alignment with the JAX-RS 2.1 specification. The provider registered will have a priority of `Integer.MAX_VALUE`, allowing a user to register a custom provider to be used instead.

Users may configure how JSON-B serializes a request entity or deserializes a response entity by registering a class or instance of `ContextResolver<Jsonb>`. For example, the following code would enable the JSON-B provider implementation to deserialize private fields (without needing getters/setters):

```
public class MyJsonbContextResolver implements ContextResolver<Jsonb> {

    @Override
    public Jsonb getContext(Class<?> type) {
        JsonbConfig config = new JsonbConfig().
                withPropertyVisibilityStrategy(new PropertyVisibilityStrategy(){
                    @Override
                    public boolean isVisible(Field f) {
                        return true;
                    }

                    @Override
                    public boolean isVisible(Method m) {
                        return false;
                    }
                });
        return JsonbBuilder.newBuilder().
                withConfig(config).
                build();
    }
}

@RegisterRestClient
@RegisterProvider(MyJsonbContextResolver.class)
public interface JsonBClient {
    //...
}
```

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(baseUri="http://someHost/someContextRoot")
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.

If the CDI implementation manages an instance of a registered provider class, the implementation must use that instance. See CDI Managed Providers for more details.

Support for MicroProfile Config

For CDI defined interfaces, it is possible to use MicroProfile Config properties to define additional behaviors or override values specified in the `@RegisterRestClient` annotation of the rest interface. Assuming this interface:

```
package com.mycompany.remoteServices;

@RegisterRestClient
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. This property will override any `baseUri` value specified in the `@RegisterRestClient` annotation.
- `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.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/connectTimeout`: Timeout specified in milliseconds to wait to connect to the remote endpoint.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/readTimeout`: Timeout specified in milliseconds to wait for a response from the remote endpoint.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/followRedirects`: A boolean value (Any value other than "true" will be interpreted as "false") used to determine whether the client should follow HTTP redirect responses.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/proxyAddress`: A string value in the form of `<proxyHost>:<proxyPort>` that specifies the HTTP proxy server hostname (or IP address) and port for requests of this client to use.
- `com.mycompany.remoteServices.MyServiceClient/mp-rest/queryParamStyle`: An enumerated type

string value with possible values of "MULTI_PAIRS" (default), "COMMA_SEPARATED", or "ARRAY_PAIRS" that specifies the format in which multiple values for the same query parameter is used.

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`.

Configuration Keys

It is possible to simplify configuration of client interfaces by using configuration keys. Config keys are specified in the `@RegisterRestClient` annotation and can be used in place of the fully-qualified classname in MP Config. For example, if we modify the previous example to be:

```
package com.mycompany.remoteServices;

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

Then config properties can be specified like: - `myClient/mp-rest/url` - `myClient/mp-rest/uri` - `myClient/mp-rest/scope` - `myClient/mp-rest/providers` - `myClient/mp-rest/providers/com.mycompany.MyProvider/priority` - `myClient/mp-rest/connectTimeout` - `myClient/mp-rest/readTimeout` - `myClient/mp-rest/followRedirects` - `myClient/mp-rest/proxyAddresses` - `myClient/mp-rest/queryParamStyle`

Multiple client interfaces may have the same configKey value, which would allow many interfaces to be configured with a single MP Config property.

If the same property exists for the same interface specified by both the configKey and the fully-qualified classname, the property specified by the fully-qualified classname takes precedence.

Lifecycle of Rest Clients

Instances of a MicroProfile Rest Client can have two states: open and closed. When open, a client instance is expected to invoke RESTful services as defined by the config and annotations described throughout this document. When closed, a client instance is expected to throw an `IllegalStateException` when a service method is invoked.

When a client instance is closed, the implementation is expected to clean up any underlying resources.

A client instance can be closed by casting the instance to a `Closeable` or `AutoCloseable` and invoking the the `close()` method (or auto-invoked if using in a try-with-resources block). For example:

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

...

MyServiceClient client = RestClientBuilder.newBuilder()
     .baseUri(apiUri)
     .build(MyServiceClient.class);
String response1 = client.greet(); // works
((Closeable)client).close();
String response2 = client.greet(); // throws IllegalStateException
```

Likewise, if the client interface extends `java.lang.AutoCloseable` or `java.io.Closeable`, the client can be closed by simply calling the inherited `close()` method. For example:

```
public interface MyServiceClient extends AutoCloseable {
    @GET
    @Path("/greet")
    String greet();
}

...

MyServiceClient client = RestClientBuilder.newBuilder()
     .baseUri(apiUri)
     .build(MyServiceClient.class);
String response1;
try (MyServiceClient c = client) {
    response1 = c.greet(); // works
} // client is auto-closed

String response2 = client.greet(); // throws IllegalStateException
```

MicroProfile Rest Client Asynchronous Support

It is possible for Rest Client interface methods to be declared asynchronous. This allows the implementation to utilize non blocking behavior in handling the request.

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 asynchronously from the invoking method.

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 is being invoked asynchronously. 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 commencing 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 commencing asynchronous processing, but before invoking further providers or returning control back to the async method caller, the MP Rest Client implementation must invoke

the `applyContext` method. The implementation must then invoke all inbound response providers (filters, interceptors, MessageBodyReaders, etc.) and then must invoke the `removeContext` method on the `AsyncInvocationInterceptor`. This allows the provider to remove any contexts before returning control back to the user.

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

```
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
    private volatile String someValue;

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

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

MicroProfile Rest Client SSL Support

MicroProfile Rest Client provides a uniform way to configure SSL for the client.

Trust store

By default, a MicroProfile Rest Client implementation uses the JVM trust store. MicroProfile Rest Client provides a way to specify a custom trust store.

For clients created programmatically, the trust store should be read to a `KeyStore` object and specified as follows:

```
KeyStore trustStore = readTrustStore();
RestClientBuilder.newBuilder()
    .trustStore(trustStore)
```

For CDI injected clients, the trust store can be specified with MicroProfile Config properties:

- `myClient/mp-rest/trustStore` to set the trust store location. Can point to either a classpath resource (e.g. `classpath:/client-truststore.jks`) or a file (e.g. `file:/home/user/client-truststore.jks`)
- `myClient/mp-rest/trustStorePassword` to set the password for the keystore
- `myClient/mp-rest/trustStoreType` to set the type of the trust store. Defaults to "JKS"

Hostname verification

A custom `HostnameVerifier` can be used to determine if an SSL connection that fails on a URL's hostname and a server's identification hostname mismatch should be allowed.

To specify a hostname verifier for a programmatically created client, use:

```
RestClientBuilder.newBuilder()
    .hostnameVerifier(verifier)
```

For CDI, the verifier can be specified by setting the `myClient/mp-rest/hostnameVerifier` MicroProfile Config property to the class name of the verifier. The class must have a public no-argument constructor.

Key store

Client key stores are useful for two-way SSL connections.

The programmatic API provides a `keystore` method for specifying the client key store. The method accepts a `KeyStore` object.

For the CDI usage, the keystore can be specified with MicroProfile Config properties similar to the trust store properties:

- `myClient/mp-rest/keyStore` to set the key store location. Can point to either a classpath resource (e.g. `classpath:/client-keystore.jks`) or a file (e.g. `file:/home/user/client-keystore.jks`)
- `myClient/mp-rest/keyStorePassword` to set the password for the keystore
- `myClient/mp-rest/keyStoreType` to set the type of the key store. Defaults to "JKS"

SSL Context

For the programmatically created client, it is also possible to configure SSL by setting the `SSLContext` using the `RestClientBuilder#sslContext` method.

MicroProfile Rest Client Server Sent Event Support

The HTTP 5 specification introduced Server Sent Events (SSE), allowing HTTP servers to push events to HTTP clients. MicroProfile Rest Client interfaces may consume events from servers that push SSEs by using the `@Produces(MediaType.SERVER_SENT_EVENTS)` annotation on the method or interface and by the interface method returning a `org.reactivestreams.Publisher<?>` type. The Publisher type is available from the Reactive Streams APIs used by the MicroProfile Reactive Streams Operators APIs.

A client interface's `Publisher` return type can include a type argument for `javax.ws.rs.sse.InboundSseEvent` allowing the client to obtain specific fields from the SSE including the name of the event, it's ID, comments, and the actual data. The data can be returned as a plain `String` or deserialized into a Java object using an applicable `MessageBodyReader` registered with the client.

Here is an example:

```
public interface SseClient {

    @GET
    @Path("ssePath")
    @Produces(MediaType.SERVER_SENT_EVENTS)
    Publisher<InboundSseEvent> getEvents();
}
```

```
void testSseClient() {
    SseClient client = RestClientBuilder.newBuilder().baseUri(someUri).build(
SseClient.class);
    Publisher<InboundSseEvent> publisher = client.getEvents();
    publisher.subscribe(new Subscriber<InboundSseEvent>(){
        int MAX_EVENTS = 3;
        int counter = 0;
        Subscription subscription;

        @Override
        public void onSubscribe(Subscription s) {
            subscription = s;
            s.request(MAX_EVENTS);
        }

        @Override
        public void onNext(InboundSseEvent event) {

            System.out.println("Received Event");
            System.out.println("  Name: " + event.getName());
            System.out.println("  ID: " + event.getId());
            System.out.println("  Comment: " + event.getComment());
            System.out.println("  Data: " + event.readData());
            if (++counter >= MAX_EVENTS) {
                subscription.cancel();
            }
        }

        @Override
        public void onError(Throwable t) {
            System.out.println("Error occurred while reading SSEs" + t);
        }

        @Override
        public void onComplete() {
            System.out.println("All done");
        }
    });
}
```

In this example, once the client instance is created and the `getEvents` method is called, the user has access to a `Publisher` instance. With that Publisher, the user can subscribe to events published by the Publisher - in this case, these events are instances of `InboundSseEvent`, which include all of the data from the SSEs. In this example, the subscriber has requested three events, and will close the connection to the server once it has received the third event. This is done with the `subscription.cancel()` call.

It is also possible to receive type-safe objects from SSEs. If the server always returns the same type of object in the SSE's data field, then the client can consume those events directly. For example,

suppose the server sends weather data in JSON format such as: `{"date":"2020-01-17", "description":"Blizzard"}` That data could be consumed into a `WeatherEvent` class directly like so:

```
public class WeatherEvent {
    private Date date;
    private String description;
    // ... getters and setters
}
```

```
public class WeatherEventProvider implements MessageBodyReader<WeatherEvent> {

    @Override
    public boolean isReadable(Class<?> type, Type genericType, Annotation[]
annotations, MediaType mediaType) {
        return WeatherEvent.class.isAssignableFrom(type);
    }

    @Override
    public WeatherEvent readFrom(Class<WeatherEvent> type, Type genericType,
Annotation[] annotations,
            MediaType mediaType, MultivaluedMap<String, String> httpHeaders,
InputStream entityStream)
            throws IOException, WebApplicationException {
        JsonReaderFactory factory = Json.createReaderFactory(null);
        JsonReader reader = factory.createReader(entityStream);
        JsonObject jsonObject = reader.readObject();
        String dateString = jsonObject.getString("date");
        String description = jsonObject.getString("description");
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        try {
            WeatherEvent event = new WeatherEvent(df.parse(dateString), description);
            return event;
        }
        catch (ParseException ex) {
            throw new IOException(ex);
        }
    }
}
```

```
@RegisterProvider(WeatherEventProvider.class)
public interface WeatherEventClient {

    @GET
    @Path("ssePath")
    @Produces(MediaType.SERVER_SENT_EVENTS)
    Publisher<WeatherEvent> getEvents();
}
```

This allows subscribers to consume the type-safe business objects (`WeatherEvent` in this example) directly without needing to manually deserialize them from the `InboundSseEvent`. Depending on the execution environment, the provider class may not be necessary.

Similar to JAX-RS, MicroProfile Rest Client implementations must use registered `MessageBodyReader` implementations to deserialize the data from the SSE into the business object. The SSE specification does not specify that a content type be sent with each SSE, so it is not always possible for Rest Client implementations to choose the correct `MessageBodyReader` for the specified business object. MessageBodyReader selection is documented in the JAX-RS specification. Users are advised to either use `Publisher<InboundSseEvent>` or create and register their own `MessageBodyReader` when type selection is difficult to determine. Users are always advised to use `Publisher<InboundSseEvent>` when a server pushes different types of objects from the endpoint.

Note that Java 9 and above provides the `java.util.concurrent.Flow` API, with enclosed interfaces that exactly match the `org.reactivestreams.` `interfaces.` `MicroProfile` `Rest` `Client` `2.0` `only` `requires` `Java` `8,` `but` `implementations` `may` `include` `support` `for` `Java` `9` `Flow` `APIs` `in` `addition` `to` `the` `org.reactivestreams.` APIs.

SSE processing is intended to be asynchronous. The `Publisher` instance returned by the client interface should fire events to any associated `Subscription` instance using the `ExecutorService` specified when the client instance was built.

Integration with other MicroProfile technologies

The MicroProfile Rest Client can be used as a standalone technology. That means that an implementation could work without CDI, MicroProfile Config, etc. This section documents how the MicroProfile Rest Client should interact when it is executed in an environment that provides other MicroProfile technologies.

CDI

Integration with CDI is already built-in to the Rest Client specification, and is documented in the MicroProfile Rest Client CDI Support section.

If CDI is available, the MP Rest Client implementation must ensure that CDI business method interceptors are invoked when the appropriate interceptor binding is applied to the client interface or method.

If a client interface specifies a custom `ClientHeadersFactory` via the `@RegisterClientHeaders` annotation, and if CDI manages a valid instance of that class, then the implementation must use that instance, allowing support of `@Inject` injection into the instance.

MicroProfile Config

MP Rest Client uses MP Config in order to declaratively configure the client behavior. The remote URI, client providers and priority, connect and read timeouts, etc. can all be configured using MP Config. See Support for MicroProfile Config for more details.

MicroProfile Fault Tolerance

MP Rest Client implementations must ensure that MP Fault Tolerance annotations on client interfaces are honored. In general, these annotations are treated as CDI interceptor bindings.

MP Rest Client should ensure that the behavior of most Fault Tolerance annotations should follow the behavior outlined in the MP Fault Tolerance specification. This includes the `@Asynchronous`, `@Bulkhead`, `@CircuitBreaker`, `@Fallback` and `@Retry` annotations.

The `@Timeout` annotation presents a problem since some parts of the MP Rest Client request are non-blocking and non-interruptible. Implementations should override the default connect and read timeouts and use the timeout value specified in the `@Timeout` annotation instead. This will ensure that the actual time spent in blocking/non-interruptible operations should be less than or equal to the time specified in the annotation, allowing the MP Fault Tolerance implementation to interrupt the request and the throw the appropriate `TimeoutException`.

JAX-RS

When a client interface is executed from within a JAX-RS context (resource or provider class), it is

possible to propagate HTTP headers using the `DefaultClientHeadersFactoryImpl` by adding the `@RegisterClientHeaders` annotation to the interface with no value. To specify which headers to propagate from the inbound JAX-RS request to the outbound MP Rest Client request, users must use a comma-separated list of headers in the following MicroProfile Config property:

`org.eclipse.microprofile.rest.client.propagateHeaders`.

If the client interface is used within a JAX-RS context, then the implementation may support injection of `@Context` fields and methods into custom `ClientHeadersFactory` instances. The injected objects are related to the JAX-RS context (i.e. an injected `UriInfo` will be specific to the JAX-RS resource's URI, not the URI of the MP Rest Client interface). This injection is optional for the implementation, so the only portable injection mechanism of `ClientHeadersFactory` instances is `@Inject` when the client is managed by CDI.

Other MicroProfile Technologies

Client requests can be automatically traced when using MP OpenTracing. Likewise, requests can be measured using MP Metrics. Configuration and usage of these technologies should be defined in their respective specification documents.

Release Notes

Release Notes for MicroProfile Rest Client 2.0

Changes since 1.4:

- Defined that CDI-managed providers should be used instead of creating a new instance, if applicable.
- Support different configurations for collections used in query parameters.
- Added proxy server configuration support.
- Added configuration for automatically following redirect responses.
- Added support for JSON-B configuration via `ContextResolver<Jsonb>`.
- Added support for Server Sent Events.
- Changed dependency scope for most dependencies to `provided`.
- Update to use Jakarta EE 8 dependencies.

Note: Prior to this release it may have been possible to use the MicroProfile Rest Client APIs with Java EE 7 APIs, which were added to the user application via the `compile` Maven scope. This is no longer possible as the APIs now depend on Jakarta EE 8 APIs and they must be provided by the implementation container (`provided` scope).

Release Notes for MicroProfile Rest Client 1.4

Changes since 1.3:

- Ensure CDI and optionally JAX-RS injection into `ClientHeadersFactory`.
- Specified `@Target` to `@RestClient` annotation.
- Removed recursive classloader check when resolving service loader for Rest Client SPI.
- Updated ParamConverter TCK test case to be more realistic (converting String-to-Widget rather than String-to-String).
- Fixed Javadoc warnings.

Release Notes for MicroProfile Rest Client 1.3

Changes since 1.2:

- Spec-defined SSL support via new `RestClientBuilder` methods and MP Config properties.
- Allow client proxies to be cast to `Closeable`/`AutoCloseable`.
- Simpler configuration using configKeys.
- Defined `application/json` to be the default MediaType if none is specified in `@Produces`/`@Consumes`.

Release Notes for MicroProfile Rest Client 1.2

Changes since 1.1:

- Generate headers en masse, including propagation of headers from inbound JAX-RS requests.
- New `@ClientHeaderParam` API for defining HTTP headers without modifying the client interface method signature.
- New section documenting the Integration with other MicroProfile technologies.
- Clarification on built-in JSON-B/JSON-P entity providers.
- New `baseUri` property added to `@RegisterRestClient` annotation.
- New `connectTimeout` and `readTimeout` methods on `RestClientBuilder` - and corresponding MP Config properties.
- `ClientRequestContext` should have a property named `org.eclipse.microprofile.rest.client.invokedMethod` containing the Rest Client `Method` currently being invoked.
- New SPI interface, `RestClientListener` interface for intercepting new client instances.
- New `removeContext` method for `AsyncInvocationInterceptor` interface.

Release Notes for MicroProfile Rest Client 1.1

Changes since 1.0:

- Asynchronous method support when Rest Client interfaces return `CompletionStage`.
- New SPI interface, `RestClientBuilderListener` for intercepting new client builders.
- `@RegisterRestClient` is now considered a bean-defining annotation.
- 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
```