

# Quarkus - Using the REST Client with Multipart

RESTEasy has rich support for the `multipart/*` and `multipart/form-data` mime types. The multipart mime format is used to pass lists of content bodies. Multiple content bodies are embedded in one message. `multipart/form-data` is often found in web application HTML Form documents and is generally used to upload files. The form-data format is the same as other multipart formats, except that each inlined piece of content has a name associated with it.

This guide explains how to use the MicroProfile REST Client with Multipart in order to interact with REST APIs requiring multipart/form-data content-type with very little effort.

## Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+

## Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `rest-client-multipart-quickstart` directory.

## Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.2.1.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=rest-client-multipart-quickstart \
  -DclassName="org.acme.restclient.multipart.MultipartClientResource" \
  -Dpath="/client" \
  -Dextensions="rest-client, resteasy"
cd rest-client-multipart-quickstart
```

This command generates the Maven project with a REST endpoint and imports the **rest-client** and **resteasy** extensions.

Then, we'll add the following dependency to support **multipart/form-data** requests:

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-multipart-provider</artifactId>
</dependency>
```

## Setting up the model

In this guide we will be demonstrating how to invoke a REST service accepting multipart/form-data input. We are assuming the payload is well-known before the request is sent, so we can model as a POJO.



If the payload is unknown, you can also use the RESTEasy custom API instead. If that's the case, see the RESTEasy Multipart Providers link in the end of the guide.

Our first order of business is to setup the model we will be using to define the **multipart/form-data** payload, in the form of a **MultipartBody** POJO.

Create a **src/main/java/org/acme/restclient/multipart/MultipartBody.java** file and set the following content:

```

package org.acme.restclient.multipart;

import java.io.InputStream;

import javax.ws.rs.FormParam;
import javax.ws.rs.core.MediaType;

import org.jboss.resteasy.annotations.providers.multipart.PartType;

public class MultipartBody {

    @FormParam("file")
    @PartType(MediaType.APPLICATION_OCTET_STREAM)
    public InputStream file;

    @FormParam("fileName")
    @PartType(MediaType.TEXT_PLAIN)
    public String fileName;
}

```

The purpose of the annotations in the code above is the following:

- **@FormParam** is a standard JAX-RS annotation used to define a form parameter contained within a request entity body
- **@PartType** is a RESTEasy specific annotation required when a client performs a multipart request and defines the content type for the part.

## Create the interface

Using the MicroProfile REST Client is as simple as creating an interface using the proper JAX-RS and MicroProfile annotations. In our case the interface should be created at `src/main/java/org/acme/restclient/multipart/MultipartService.java` and have the following content:

```

package org.acme.restclient.multipart;

import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import org.jboss.resteasy.annotations.providers.multipart.MultipartForm;

@Path("/echo")
@RegisterRestClient
public interface MultipartService {

    @POST
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    @Produces(MediaType.TEXT_PLAIN)
    String sendMultipartData(@MultipartForm MultipartBody data);

}

```

The `sendMultipartData` method gives our code the ability to POST a `multipart/form-data` request to our Echo service (running in the same server for demo purposes). Because in this demo we have the exact knowledge of the `multipart/form-data` packets, we can map them to the model class created in the previous section using the `@org.jboss.resteasy.annotations.providers.multipart.MultipartForm` annotation.

The client will handle all the networking and marshalling leaving our code clean of such technical details.

The purpose of the annotations in the code above is the following:

- `@RegisterRestClient` allows Quarkus to know that this interface is meant to be available for CDI injection as a REST Client
- `@Path`, `@GET` and `@PathParam` are the standard JAX-RS annotations used to define how to access the service
- `@MultipartForm` defines the parameter as a value object for incoming/outgoing request/responses of the multipart/form-data mime type.
- `@Consumes` defines the expected content-type consumed by this request (parameters)
- `@Produces` defines the expected content-type produced by this request (return type)



While `@Consumes` and `@Produces` are optional as auto-negotiation is supported, it is heavily recommended to annotate your endpoints with them to define precisely the expected content-types.

It will allow to narrow down the number of JAX-RS providers (which can be seen as converters) included in the native executable.

## Create the configuration

In order to determine the base URL to which REST calls will be made, the REST Client uses configuration from `application.properties`. The name of the property needs to follow a certain convention which is best displayed in the following code:

```
# Your configuration properties
org.acme.restclient.multipart.MultipartService/mp-
rest/url=http://localhost:8080/
```

Having this configuration means that all requests performed using `org.acme.restclient.multipart.MultipartService` will use `http://localhost:8080/` as the base URL.

Note that `org.acme.restclient.multipart.MultipartService` *must* match the fully qualified name of the `MultipartService` interface we created in the previous section.

## Update the JAX-RS resource

Open the `src/main/java/org/acme/restclient/multipart/MultipartClientResource.java` file and update it with the following content:

```

package org.acme.restclient.multipart;

import java.io.ByteArrayInputStream;

import javax.inject.Inject;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.eclipse.microprofile.rest.client.inject.RestClient;

@Path("/client")
public class MultipartClientResource {

    @Inject
    @RestClient
    MultipartService service;

    @POST
    @Path("/multipart")
    @Produces(MediaType.TEXT_PLAIN)
    public String sendFile() throws Exception {
        MultipartBody body = new MultipartBody();
        body.fileName = "greeting.txt";
        body.file = new ByteArrayInputStream("HELLO WORLD".
getBytes());
        return service.sendMultipartData(body);
    }
}

```

Note that in addition to the standard CDI `@Inject` annotation, we also need to use the MicroProfile `@RestClient` annotation to inject `MultipartService`.

## Creating the server

For demo purposes, let's create a simple Echo endpoint that will act as the server part.

Create the directory `src/main/java/org/acme/restclient/multipart/server` and include a `EchoService.java` file with the following content:

```
package org.acme.restclient.multipart.server;

import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/echo")
public class EchoService {

    @POST
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    @Produces(MediaType.TEXT_PLAIN)
    public String echo(String requestBody) throws Exception {
        return requestBody;
    }
}
```

This will just return the request body and it's useful for testing purposes.

## Update the test

We also need to update the functional test to reflect the changes made to the endpoint. Edit the `src/test/java/org/acme/restclient/multipart/MultipartResourceTest.java` file to:

```

package org.acme.restclient.multipart;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.containsString;

@QuarkusTest
public class MultipartResourceTest {

    @Test
    public void testMultipartDataIsSent() {
        given()
            .when().post("/client/multipart")
            .then()
            .statusCode(200)
            .body(containsString("Content-Disposition: form-
data; name=\"file\""),
                containsString("HELLO WORLD"),
                containsString("Content-Disposition: form-
data; name=\"fileName\""),
                containsString("greeting.txt"));
    }
}

```

The code above uses [REST Assured](#) to assert that the returned content from the echo service contains multipart elements

Because the test runs in a different port, we also need to include an `application.properties` in our `src/test/resources` with the following content:

```

# Your configuration properties
org.acme.restclient.multipart.MultipartService/mp-
rest/url=http://localhost:8081/

```

## Package and run the application

- Run the application with: `./mvnw compile quarkus:dev`.
- In a terminal, run `curl -X POST http://localhost:8080/client/multipart`

You should see an output similar to:



```
--89d288bd-960f-460c-b266-64c5b4d170fa
Content-Disposition: form-data; name="fileName"
Content-Type: text/plain

greeting.txt
--89d288bd-960f-460c-b266-64c5b4d170fa
Content-Disposition: form-data; name="file"
Content-Type: application/octet-stream

HELLO WORLD
--89d288bd-960f-460c-b266-64c5b4d170fa--
```

As usual, the application can be packaged using `./mvnw clean package` and executed using the `-runner.jar` file. You can also generate the native executable with `./mvnw clean package -Pnative`.

## Further reading

- [MicroProfile Rest Client specification](#)
- [RESTEasy Multipart Providers](#)