Java HTTP Adapter

Overview

Java adapters provide free reign over connectivity to a backend system. It is therefore the developer's responsibility to ensure best practices regarding performance and other implementation details. This tutorial covers an example of a Java adapter that connects to an RSS feed by using a Java `HttpClient`.

Prerequisite: Make sure to read the Java Adapters (../) tutorial first.

Initializing the adapter

In the supplied sample adapter, the `JavaHTTPApplication` class is used to extend `MFPJAXRSApplication` and is a good place to trigger any initialization required by your application.

```
@Override
protected void init() throws Exception {
    JavaHTTPResource.init();
    logger.info("Adapter initialized!");
}
```

Implementing the adapter Resource class

The adapter Resource class is where requests to the server are handled.
In the supplied sample adapter, the class name is `JavaHTTPResource`.

```
@Path("/")
public class JavaHTTPResource {

}
```

`@Path("/")` means that the resources will be available at the URL `http(s)://host:port/ProjectName/adapters/AdapterName/`.

HTTP Client

```
private static CloseableHttpClient client;
private static HttpHost host;

public static void init() {
  client = HttpClientBuilder.create().build();
  host = new HttpHost("mobilefirstplatform.ibmcloud.com");
}
```

Because every request to your resource will create a new instance of `JavaHTTPResource`, it is important to reuse objects that may impact performance. In this example we made the Http client a `static` object and initialized it in a static `init()` method, which gets called by the `init()` of `JavaHTTPApplication` as described above.

Procedure resource

```
@GET
@Produces("application/json")
public void get(@Context HttpServletResponse response, @QueryParam("tag") String tag)
    throws IOException, IllegalStateException, SAXException {
  if(tag!=null && !tag.isEmpty()){
    execute(new HttpGet("/blog/atom/"+ tag +".xml"), response);
  }
  else{
    execute(new HttpGet("/feed.xml"), response);
  }

}
```

The sample adapter exposes just one resource URL which allows to retrieve the RSS feed from the backend service.

- `@GET` means that this procedure only responds to `HTTP GET` requests.
- `@Produces("application/json")` specifies the Content Type of the response to send back. We chose to send the response as a `JSON` object to make it easier on the client-side.
- `@Context HttpServletResponse response` will be used to write to the response output stream. This enables us more granularity than returning a simple string.
- `@QueryParam("tag")` String tag enables the procedure to receive a parameter. The choice of `QueryParam` means the parameter is to be passed in the query (`/JavaHTTP/?tag=MobileFirst_Platform`). Other options include `@PathParam`, `@HeaderParam`, `@CookieParam`, `@FormParam`, etc.
- `throws IOException, ...` means we are forwarding any exception back to the client. The client code is responsible for handling potential exceptions which will be received as `HTTP 500` errors. Another solution (more likely in production code) is to handle exceptions in your server Java code and decide what to send to the client based on the exact error.
- `execute(new HttpGet("/feed.xml"), response)`. The actual HTTP request to the backend service is handled by another method defined later.

Depending if you pass a `tag` parameter, `execute` will retrieve a different build a different path and retrieve a different RSS file.

execute()

```
public void execute(HttpUriRequest req, HttpServletResponse resultResponse)
        throws ClientProtocolException, IOException,
        IllegalStateException, SAXException {
    HttpResponse RSSResponse = client.execute(host, req);
    ServletOutputStream os = resultResponse.getOutputStream();
    if (RSSResponse.getStatusLine().getStatusCode() == HttpStatus.SC_OK){
        resultResponse.addHeader("Content-Type", "application/json");
        String json = XML.toJson(RSSResponse.getEntity().getContent());
        os.write(json.getBytes(Charset.forName("UTF-8")));

    }else{
        resultResponse.setStatus(RSSResponse.getStatusLine().getStatusCode());
        RSSResponse.getEntity().getContent().close();
        os.write(RSSResponse.getStatusLine().getReasonPhrase().getBytes());
    }
    os.flush();
    os.close();
}
```

- `HttpResponse RSSResponse = client.execute(host, req)`. We use our static HTTP client to execute the HTTP request and store the response.
- `ServletOutputStream os = resultResponse.getOutputStream()`. This is the output stream to write a response to the client.
- `resultResponse.addHeader("Content-Type", "application/json")`. As mentioned before, we chose to send the response as JSON.
- `String json = XML.toJson(RSSResponse.getEntity().getContent())`. We used `org.apache.wink.json4j.utils.XML` to convert the XML RSS to a JSON string.
- `os.write(json.getBytes(Charset.forName("UTF-8")))` the resulting JSON string is written to the output stream.

The output stream is then `flush`ed and `close`d.

If `RSSResponse` is not `200 OK`, we write the status code and reason in the response instead.

Sample adapter

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/Adapters/tree/release80) the Adapters Maven project.

The Adapters Maven project includes the **JavaHTTP** adapter described above.

Sample usage

- Use either Maven or MobileFirst Developer CLI to build and deploy the JavaHTTP adapter (../../creating-adapters/).
- To test or debug an adapter, see the testing and debugging adapters (../../testing-and-debugging-adapters) tutorial.