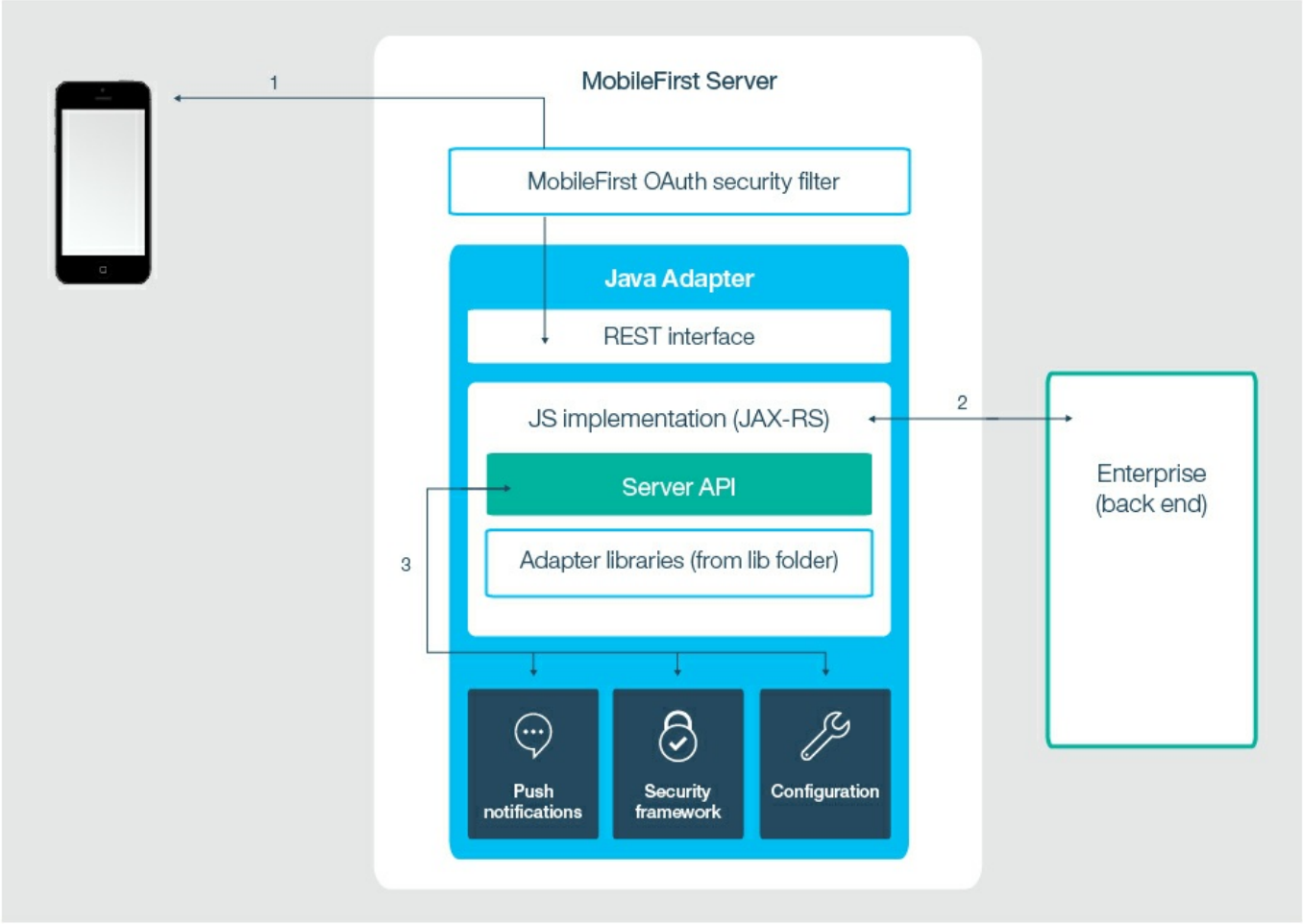
Java Adapters

Overview

Prerequisite: Make sure to read the Adapters Overview (../../server-side-development/adapter-framework-overview/) tutorial first.

Java adapters are based on the JAX-RS specification. In other words, a Java adapter is a JAX-RS service that can easily be deployed to a MobileFirst Server instance and has access to MobileFirst Server APIs.

In Java adapters, it is up to the developer to define the returned content and its format, as well as the URL structure of each resource. The only exception is if the client sending the request supports GZip, then the returned content encoding of the Java adapter is compressed by GZip. All operations on the returned content are done and owned by the developer.



Each Java adapter has its own isolated sandbox, in which all its classes run without knowing about or interrupting other adapter sandboxes. That said, adapters can still communicate with one another by calling API which makes "adapter mashup" possible.

It is possible to include 3rd-party libraries required by the adapter code in the `adapter/lib` directory. The libraries that you include in this directory override any libraries that are provided by the application server or included in the `server/lib` folder.

Agenda

- Benefits of Java adapters
- Creating a Java adapter
- File structure
- The JAX-RS application class
- Implementing a JAX-RS resource
- MobileFirst server-side API
- Testing MobileFirst Adapter
- HTTP & SQL best practices

> For more information about JAX-RS, see https://jsr311.java.net/nonav/releases/1.1/index.html (https://jsr311.java.net/nonav/releases/1.1/index.html)

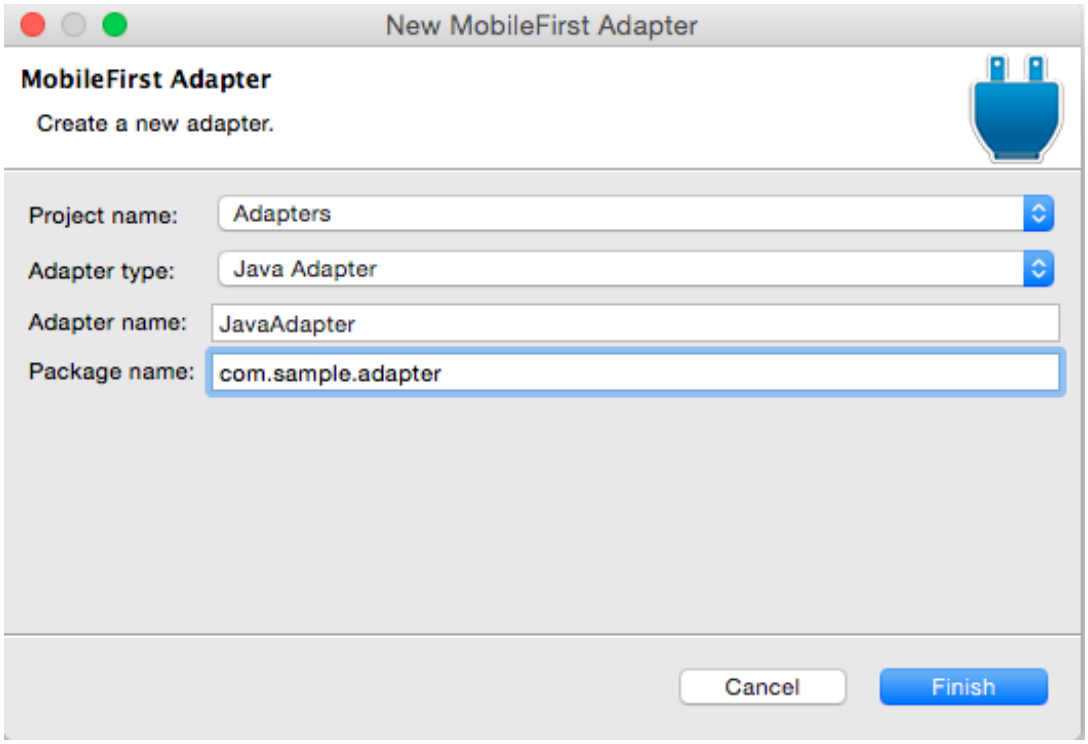
Benefits of Java adapters

- The ability to fully control the URL structure, the content types, the request and response headers, content and encoding
- Easy and fast development and testing by using MobileFirst Studio or the Command Line Interface (CLI)
- The ability to test the adapter without MobileFirst Studio or CLI by using a 3rd-party tool such as Postman
- Easy and fast deployment to a running MobileFirst Server instance with no compromise on performance and no downtime
- Security integration with the MobileFirst security model by using simple annotations in the source code.

Creating a Java adapter

By using MobileFirst Studio

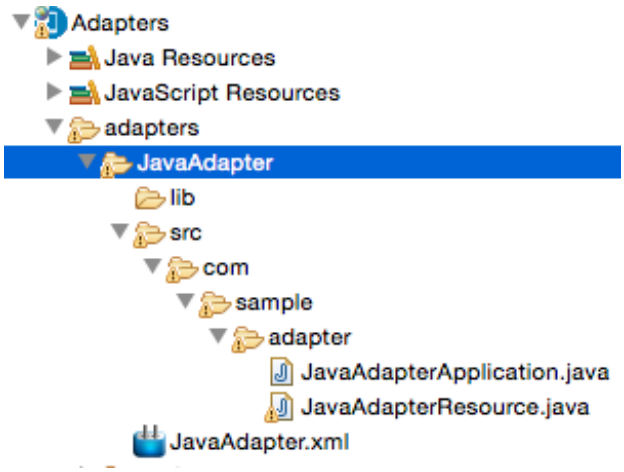
1. In MobileFirst Studio, right-click a MobileFirst project and select **New > MobileFirst Adapter**.
2. In the dialog, select **Java Adapter**.
3. Fill the adapter name and package name fields.
4. Click **Finish**.



By using MobileFirst CLI

1. Open the terminal.
2. Go to the MobileFirst project folder.
3. Enter `mfp add adapter {name} --type java --package {package name}`

File structure



XML File

Java adapters have an XML configuration file such as all the other types of adapters. In this configuration file, configure the class name of the JAX-RS application for this adapter.
In our case: `com.sample.adapter.JavaAdapterApplication`.

<?xml version="1.0" encoding="UTF-8"?>
<wl:adapter name="JavaAdapter"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:wl="http://www.ibm.com/mfp/integration"
 xmlns:http="http://www.ibm.com/mfp/integration/http">
 <displayName>JavaAdapter</displayName>
 <description>JavaAdapter</description>
 <connectivity>
  <connectionPolicy xsi:type="wl:NullConnectionPolicyType"></connectionPolicy>
 </connectivity>
 <JAXRSApplicationClass>com.sample.adapter.JavaAdapterApplication</JAXRSApplicationClass>

</wl:adapter>

Note that the connection policy type is `NullConnectionPolicy`, which means that the XML configuration file is not used to define the connectivity of the adapter. In Java adapters, this is the only allowed connectivity type. The MobileFirst Server leaves setting up the connectivity policy to the developer of the adapter. For example, an adapter can contain code that uses an Apache HTTP client to connect to a back-end system.

The `src` folder

In this folder, put the Java sources of the JAX-RS service. JAX-RS services are composed of an application class (which extends `com.worklight.wink.extensions.MFPJAXRSApplication`) and resources classes.

The JAX-RS application and resources classes define the Java methods and their mapping to URLs. `com.sample.JavaAdapterApplication` is the JAX-RS application class and `com.sample.JavaAdapterResource` is a JAX-RS resource included in the application.

The `lib` folder

In the `lib` folder you can put JAR library files that will be available in the adapter's class path at run time. Note that these JAR files are loaded in the private class loader of the adapter as `parent last`. This setting means that the JAR files override any libraries provided by the container.

JAX-RS application class

The JAX-RS application class tells the JAX-RS framework which resources are included in the application.

```
package com.sample.adapter;
import java.util.logging.Logger;
import com.worklight.wink.extensions.MFPJAXRSApplication;
public class JavaAdapterApplication extends MFPJAXRSApplication{
 static Logger logger = Logger.getLogger(JavaAdapterApplication.class.getName());
 @Override
 protected void init() throws Exception {
  logger.info("Adapter initialized!");
 }
 @Override
 protected String getPackageToScan() {
  //The package of this class will be scanned (recursively) to find JAX-RS resources.
  return getClass().getPackage().getName();
 }
}
```

The `MFPJAXRSApplication` class scans the package for JAX-RS resources and automatically creates a list. Additionally, its `init` method is called by MobileFirst Server as soon as the adapter is deployed (before it starts serving) and when the MobileFirst runtime starts up.

Implementing a JAX-RS resource

JAX-RS resource is a POJO (Plain Old Java Object) which is mapped to a root URL and has Java methods for serving requests to this root URL and its child URLs. Any resource can have a separate set of URLs.

```
package com.sample.adapter;
import java.util.logging.Logger;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import com.worklight.adapters.rest.api.WLServerAPI;
import com.worklight.adapters.rest.api.WLServerAPIProvider;
@Path("/")
public class JavaAdapterResource {
 //Define logger (Standard java.util.Logger)
 static Logger logger = Logger.getLogger(JavaAdapterResource.class.getName());
   //Define the server api to be able to perform server operations
   WLServerAPI api = WLServerAPIProvider.getWLServerAPI();
 /* Path for method: "<server address>/Adapters/adapters/JavaAdapter/{username}" */
 @GET
 @Path("/{username}")
 public String helloUser(@PathParam("username") String name){
  return "Hello " + name;
 }
}
```

`@Path("/")` before the class definition determines the root path of this resource. If you have multiple resource classes, you should set each resource a different path.

For example, if you have a `UserResource` with `@Path("/users")` to manage users of a blog, that resource is accessible via `http(s)://host:port/ProjectName/adapters/AdapterName/users/`.

That same adapter may contain another resource `PostResource` with `@Path("/posts")` to manage posts of a blog. It is accessible via the `http(s)://host:port/ProjectName/adapters/AdapterName/`*`posts`*`/` URL.

In the example above, because there it has only one resource class, it is set to `@Path("/")` so that it is accessible via `http(s)://host:port/Adapters/adapters/JavaAdapter/`.

Each method is preceded by one or more JAX-RS annotations, for example an annotation of type "HTTP request" such as `@GET`, `@PUT`, `@POST`, `@DELETE`, or `@HEAD`. Such annotations define how the method can be accessed.

Another example is `@Path("/{username}")`, which defines the path to access this procedure (in addition to the resource-level path). As you can see, this path can include a variable part. This variable is then used as a parameter of the method, as defined `@PathParam("username") String name`.

> You can use many other annotations. See **Annotation Types Summary** here: https://jsr311.java.net/nonav/releases/1.1/javax/ws/rs/package-summary.html (https://jsr311.java.net/nonav/releases/1.1/javax/ws/rs/package-summary.html)

HTTP Session

Depending on your infrastructure and configuration, your MobileFirst server may be running with `SessionIndependent` set to true, where each request can reach a different node and HTTP sessions are not used.
In such cases you should not rely on Java's HttpSession to persist data from one request to the next.

MobileFirst server-side API

Java adapters can use the MobileFirst server-side Java API to perform operations that are related to MobileFirst Server, such as calling other adapters, submitting push notifications, logging to the server log, getting values of configuration properties, reporting activities to Analytics and getting the identity of the request issuer.

To get the server API interface, use the following code:

```
WLServerAPI api = WLServerAPIProvider.getWLServerAPI();
```

The `WLServerAPI` interface is the parent of all the API categories: (Analytics, Configuration, Push, Adapters, Authentication).

If you want, for example, to use the push API, you can write:

```
PushAPI pushApi = serverApi.getPushAPI();
```

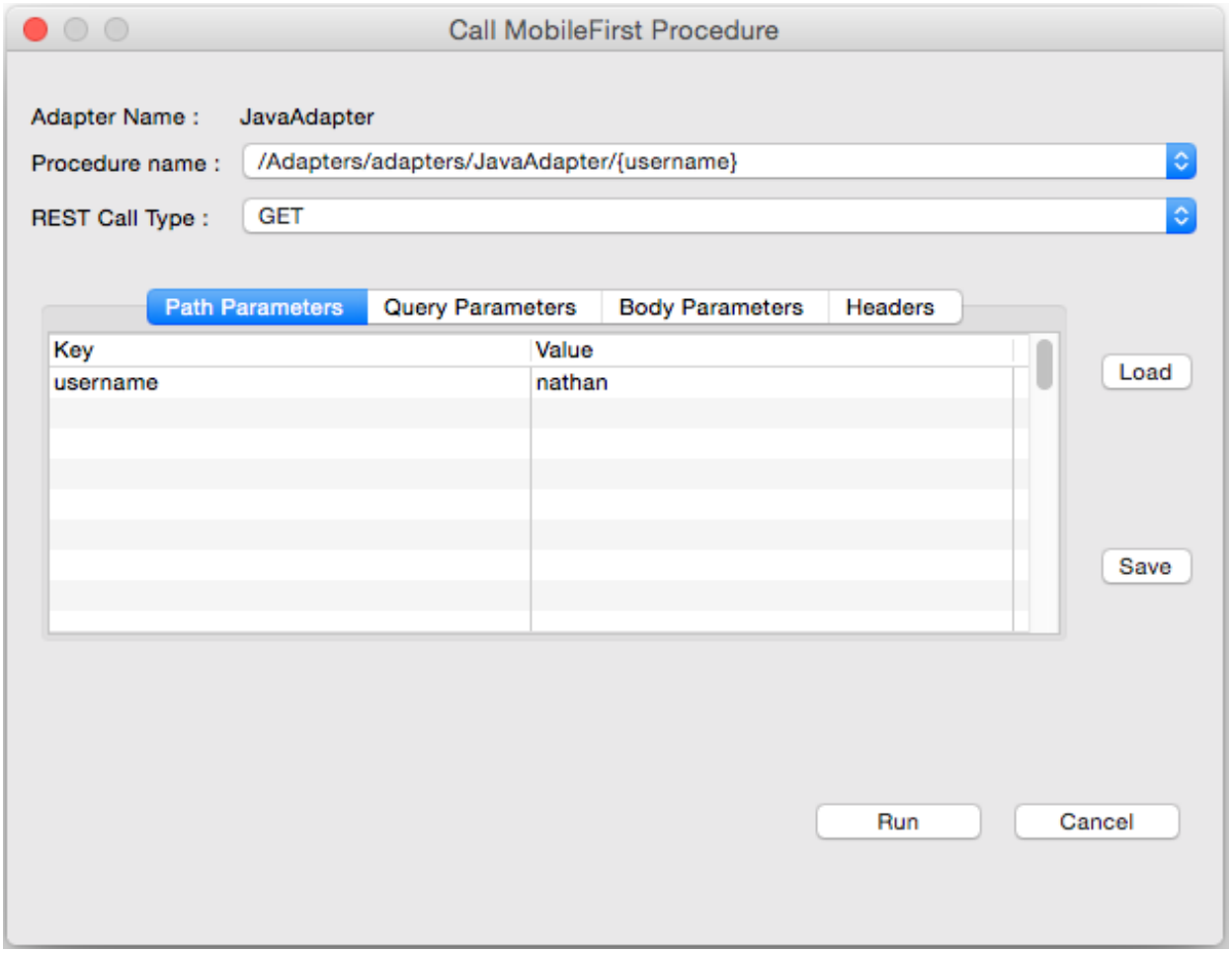
> For more information, review the Server-side API topics in the user documentation.

Testing MobileFirst adapters

In MobileFirst Studio

MobileFirst Studio provides a way to easily test Java adapters.

1. Right-click your adapter.
2. Select **Run As > Call MobileFirst Adapter**.
3. Enter any required parameters and click **Run**.



In the Command Line interface (CLI)

Enter `$ mfp adapter call`, then follow the prompts by using the arrow keys.

> For more information, see the Using CLI to create, build, and manage MobileFirst project artifacts (../../advanced-client-side-development/using-cli-to-create-build-and-manage-mobilefirst-project-artifacts/) tutorial.

In Postman

MobileFirst Java adapters are available via a REST interface. This means that if you know the URL of a resource/procedure, you can use HTTP tools such as Postman to test requests and pass URL parameters, path parameters, body parameters or headers as you see fit.

If your resource is protected by a security scope (see Protecting Java adapters (../../authentication-security/authentication-concepts/oauth-based-security-model/#protectAdapt)), the request prompts you to provide a valid authorization header. Note that by default, MobileFirst uses a simple security

scope even if you did not specify any. So unless you specifically disabled security, the endpoint is always protected.

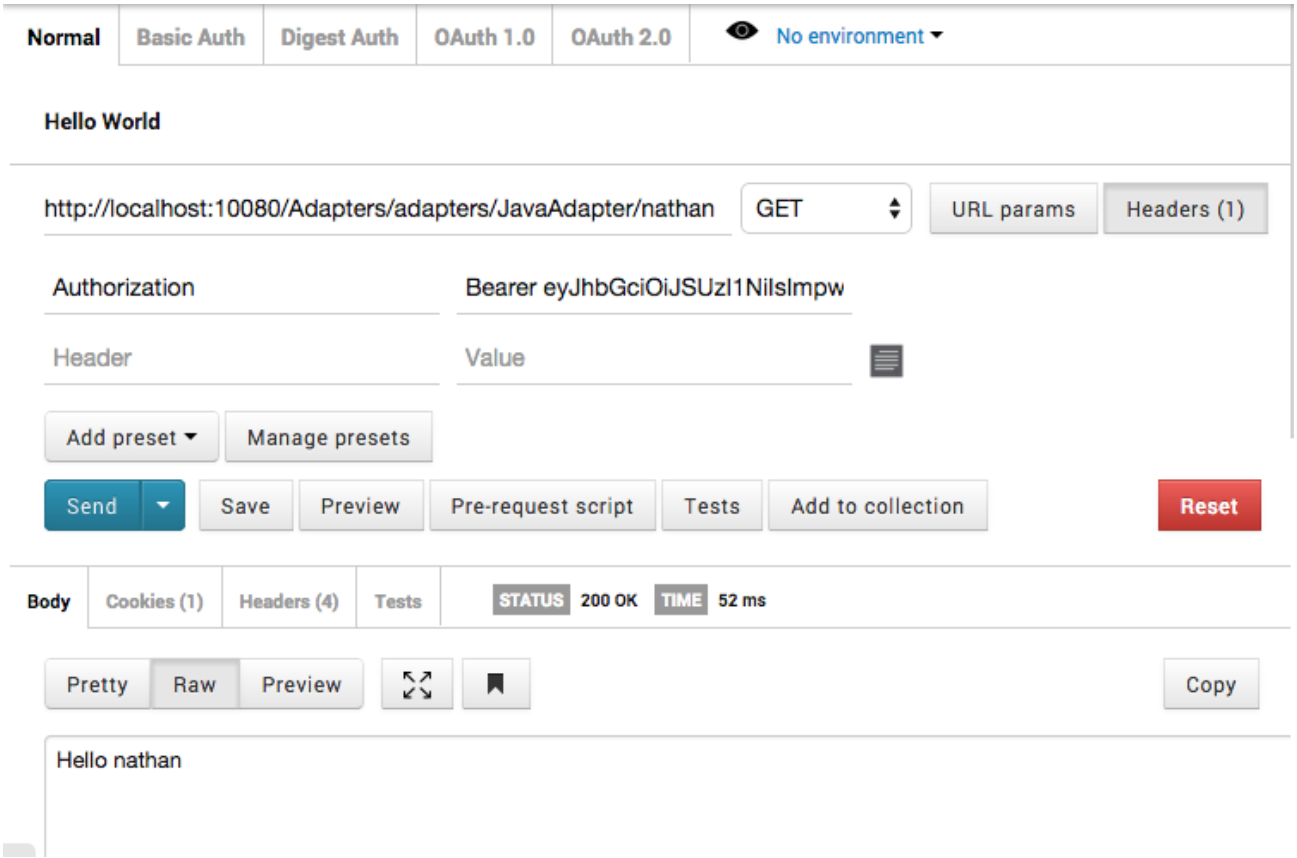
For you to work around this during your development stage, the development version of the MobileFirst server includes a test token endpoint.
To receive a Test Token, make an HTTP `POST` request to `http(s)://{server_ip}:{server_port}/{project_name}/authorization/v1/testtoken` with your HTTP client (Postman).

You receive a JSON object with a temporary valid authorization token:


```
{
"Authorization": "Bearer eyJhbGciOiJSUzI1NiIsImpwayI6eyJhbGciOiJSU0EiLCJleHAiOiJBUUFC
IiwibW9kIjoiQU0wRGQ3eEFkdjZILXInTDdyOHFDTGRFLTNJMm2FPZUlxb2UtckpBMHVadXcyc
khoWFozV1ZDZUtlelJWY0NPWXNRTi1tUUswbWZ6NV8zby1ldjBVWXdYa1NPd0JCbDFFaHFJ
d1ZEd09pZWcySk1HbDBFWHNQWmZrTlpJLUhVNG9NaWktVHJOTHp..."
}
```


In your next requests to your adapter endpoints, add an HTTP header with the name "`Authorization`" and the value you received previously (starting with `Bearer`).

The security framework now skips any security challenges protecting your resource.



> See it in action in this video blog entry: Getting familiar with IBM MobileFirst Platform Foundation Java Adapters (file:////home/travis/build/MFPSamples/DevCenter/_site/blog/2015/03/24/getting-familiar-ibm-mobilefirst-platform-foundation-java-adapters/)

HTTP & SQL best practices

For examples of Java adapters communicating with an HTTP or SQL back end, see: