REST tutorial

Table of Contents

The jPOS REST tutorial shows you how to use an out-of-the-box binary distribution of jPOS to build a REST server that responds to an /echo call.

This tutorial will show you how to do that by creating a Java project that will use jPOS as a dependency.

Create a project

Just as with the previous tutorials, we will start by cloning the jPOS-template project.

Visit the jPOS-template Github page and download or clone the jPOS-template. Let's call our project **rest-tutorial**. We use the `clone` command here, but downloading the ZIP is basically the same thing.

```
git clone https://github.com/jpos/jPOS-template.git rest-tutorial ①
cd rest-tutorial
rm -fr .git                                                       ②
```

① Visit jPOS-template and clone or download it.

② If you cloned it, you may want to get rid of the .git reference and `git init` your own.

Test run

```
gradle installApp                                    ①
build/install/rest-tutorial/bin/q2                   ②
```

① Build the project and install it in the `build/install` directory

② Run it in the foreground using the `bin/q2` start-up script

TIP If you don't have Gradle installed in your system, you can use the provided Gradle wrapper by calling `./gradlew` instead of your default `gradle`. It's a good idea to install a recent version of Gradle in your development environment, though.

The `dist` target creates a binary distribution in the `build/distributions` directory, that can be expanded in a staging directory and run from there. The `installApp` target is basically the same as calling `gradle dist` and then exploding the tarball in a working directory, in this case `build/install`.

TIP You can install a handy `q2` script like the following in your PATH:

```
#!/bin/bash
exec build/install/${PWD##*/}/bin/q2 "$*"
```

Then you can call `gradle iA && q2` to build and run instead of `gradle iA && build/install/rest-tutorial/bin/q2`. Please note `iA` is a shortcut for `installApp`.

Once you run q2, you'll see something like this:

```
<log realm="Q2.system" at="2017-05-06T20:58:57.419">
  <info>
    Q2 started, deployDir=/Users/spr/tutorials/rest-tutorial/build/install/rest-
tutorial/deploy
  </info>
</log>
...
...
```

Stop the system using `Ctrl-C`, you are now ready for the next step, but before that, we'd like you to understand why this works the way it does.

When we call `gradle dist` to build a binary distribution with a directory structure similar to the one you've seen in the previous tutorial, or call `gradle installApp` to build the distribution and then expand it in the `build/install` directory, the following takes place:

- The jPOS standard directory structure gets created in the `build/install/<projectname>/` directory (in this case `build/install/rest-tutorial`).
- The `src/dist/deploy` and `src/dist/cfg` directories get copied to their destination directory in

`build/install/rest-tutorial/deploy` and `build/install/rest-tutorial/cfg` directories. Same goes for the `src/dist/bin` directory that contains the `q2`, `start` and `stop` scripts (and `.BAT` for Windows users)

- And finally, the main jar in the `build/install/rest-tutorial` directory named after the project name and version (defined in the main `build.gradle` file).

So if we go to the `build/install/rest-tutorial` directory, we'll see a jar like this:

```
rest-tutorial-2.1.0.jar
```

If we analyze its content (using `jar tvf rest-tutorial-2.1.0.jar`) we'll see something like this:

```
    0 Sun May 21 21:06:00 ART 2017 META-INF/
  498 Sun May 21 21:06:00 ART 2017 META-INF/MANIFEST.MF
  126 Sun May 21 21:06:00 ART 2017 buildinfo.properties
   83 Sun May 21 21:06:00 ART 2017 revision.properties
```

If we expand it and take a look at the `META-INF/MANIFEST.MF` file we'll see something like this:

```
Manifest-Version: 1.0
Implementation-Title: rest-tutorial
Implementation-Version: 2.1.0
Class-Path: lib/jpos-2.1.0-SNAPSHOT.jar lib/jdom2-2.0.6.jar lib/jdbm-1    ①
 .0.jar lib/je-7.0.6.jar lib/commons-cli-1.3.1.jar lib/jline-3.2.0.jar
  lib/bsh-2.0b6.jar lib/javatuples-1.2.jar lib/org.osgi.core-6.0.0.jar
  lib/bcprov-jdk15on-1.56.jar lib/bcpg-jdk15on-1.56.jar lib/sshd-core-
 1.3.0.jar lib/slf4j-api-1.7.22.jar lib/javassist-3.21.0-GA.jar lib/Hd
 rHistogram-2.1.9.jar
Main-Class: org.jpos.q2.Q2                                               ②
```

① The main jar contains a reference to its dependencies which are available in the `lib` directory.

② We designate `org.jpos.q2.Q2` as our main class.

That little jar, which for now has no compiled classes is the reason we can launch Q2 just by calling:

```
java -jar rest-tutorial-2.1.0.jar
```

which is basically what the `bin/q2` script does (it just adds a few switches and defaults).

Add required dependencies

Open the `build.gradle` file. We will need to apply the `war` plugin and then the required dependencies. In the plugins area we add the following line:

```
apply plugin: 'war'
```

And at the end of the file we add:

```
installApp.dependsOn('war')
dist.dependsOn('war')
```

Now we need to add the dependencies that we will use in this example:

- jPOS-EE EERest module
- jPOS-EE Jetty module
- Java API for RESTful Web Services (JAX-RS)
- Jersey
- Swagger (optional)

```
providedCompile "org.jpos.ee:jposee-eerest:2.2.5-SNAPSHOT"
providedCompile "org.jpos.ee:jposee-jetty:2.2.5-SNAPSHOT"
providedCompile "javax.ws.rs:javax.ws.rs-api:2.0.1"
providedCompile "org.glassfish.jersey.media:jersey-media-json-jackson:2.22.1"
providedCompile "org.glassfish.jersey.core:jersey-server:2.22.1"
providedCompile "org.glassfish.jersey.containers:jersey-container-servlet:2.22.1"
compile 'io.swagger:swagger-jersey2-jaxrs:1.5.3'                ①
```

① Optional.

The complete `build.gradle` file should end looking like this:

```
apply plugin: 'java'
apply plugin: 'maven'
apply plugin: 'idea'
apply plugin: 'eclipse'
apply plugin: 'war'

buildscript {
    repositories { jcenter() }
}

group = 'org.jpos.template'
version = '2.1.0'
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
    maven { url 'http://jpos.org/maven' }
    maven { url 'http://download.oracle.com/maven' }
    mavenLocal()
}

dependencies {
    compile ('org.jpos:jpos:2.1.+') {
        exclude(module: 'junit')
        exclude(module: 'hamcrest-core')
    }
    testCompile 'junit:junit:4.8.2'
    providedCompile "org.jpos.ee:jposee-eerest:2.2.5-SNAPSHOT"
    providedCompile "org.jpos.ee:jposee-jetty:2.2.5-SNAPSHOT"
    providedCompile "javax.ws.rs:javax.ws.rs-api:2.0.1"
    providedCompile "org.glassfish.jersey.media:jersey-media-json-jackson:2.22.1"
    providedCompile "org.glassfish.jersey.core:jersey-server:2.22.1"
    providedCompile "org.glassfish.jersey.containers:jersey-container-servlet:2.22.1"
    compile 'io.swagger:swagger-jersey2-jaxrs:1.5.3'
}

apply from: 'jpos-app.gradle'
installApp.dependsOn('war')
dist.dependsOn('war')
```

Jetty configuration

The next step is to run `gradle installResources` that will extract everything needed from jPOS-EE's Jetty module.

```
~/rest-tutorial$ gradle installResources
:compileJava
:createBuildTimestampPropertyFile
:createRevisionPropertyFile
:processResources NO-SOURCE
:classes
:installResources
src/dist/deploy/99_sysmon.xml exists, use --force to override
src/dist/deploy/00_logger.xml exists, use --force to override
Created /Users/spr/rest-tutorial/src/dist/cfg
extracting src/dist/cfg/jetty.xml
extracting src/dist/cfg/keystore.jks
extracting src/dist/cfg/webdefault.xml
Created /Users/spr/rest-tutorial/src/dist/webapps/root
extracting src/dist/webapps/root/index.html
extracting src/dist/deploy/90_jetty.xml
src/dist/log/q2.log exists, use --force to override
src/dist/deploy/99_sysmon.xml exists, use --force to override
src/dist/deploy/00_logger.xml exists, use --force to override
extracting src/dist/deploy/01_db.sample
```

Inside `/src` we then need to create a `main/webapp/WEB-INF` directory.

```
~/rest-tutorial$ cd src/
~/rest-tutorial/src$ mkdir -p main/webapp/WEB-INF
```

In this new `WEB-INF` directory you need to add the following two files:

- `jetty-web.xml`

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN"
"http://www.eclipse.org/jetty/configure_9_0.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
    <Set name="contextPath">/api</Set>
    <Set name="war"><SystemProperty name="jetty.home" default="."/>
/webapps/@warname@</Set>
    <Set name="extractWAR">false</Set>
</Configure>
```

As you may notice, we set our REST API under `/api`. This is completely arbitrary and could be

defined to your preferred path.

- web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">
    <servlet>
        <display-name>jPOS</display-name>
        <servlet-name>Sample-REST-API</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>org.jpos.rest.App</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Sample-REST-API</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>Jersey2Config</servlet-name>
        <servlet-class>io.swagger.jersey.config.JerseyJaxrsConfig</servlet-class>
        <init-param>
            <param-name>api.version</param-name>
            <param-value>1.0.0</param-value>
        </init-param>
        <init-param>
            <param-name>swagger.api.basepath</param-name>
            <param-value>http://localhost:8080</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
</web-app>
```

Implement Echo

We will now create the App class that registers the packages that will contain our API and an implementation of an `/echo` call.

First, inside the `/main` directory we recently created we will add a `/java` directory.

Inside this directory we will add the package `org.jpos.rest` with two classes:

- `App.java`

```
package org.jpos.rest;

import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import org.glassfish.jersey.jackson.JacksonFeature;
import org.glassfish.jersey.server.ResourceConfig;
import javax.ws.rs.ext.ContextResolver;

public class App extends ResourceConfig {
    public App() {
        super();
        register(JacksonFeature.class);
        register(new App.Resolver());
        register(io.swagger.jaxrs.listing.ApiListingResource.class); //Optional
        register(io.swagger.jaxrs.listing.SwaggerSerializers.class); //Optional
        packages("org.jpos.rest");

    }
    class Resolver implements ContextResolver<ObjectMapper> {
        final ObjectMapper defaultObjectMapper = createDefaultMapper();

        @Override
        public ObjectMapper getContext(Class<?> type) {
            return defaultObjectMapper;
        }
        private ObjectMapper createDefaultMapper() {
            final ObjectMapper mapper = new ObjectMapper();
            mapper.configure(JsonGenerator.Feature.WRITE_BIGDECIMAL_AS_PLAIN, true);
            mapper.configure(JsonGenerator.Feature.ESCAPE_NON_ASCII, true);
            mapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
            mapper.enable(SerializationFeature.INDENT_OUTPUT);
            return mapper;
        }
    }
}
```

- `Echo.java`

```
package org.jpos.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import java.util.HashMap;
import java.util.Map;

@Path("/echo")
public class Echo {
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public Response echoGet(){
        Map<String, Object> resp = new HashMap<>();
        resp.put("success", "true");
        Response.ResponseBuilder rb = Response.ok(resp, MediaType.APPLICATION_JSON
).status(Response.Status.OK);
        return rb.build();
    }
}
```

After doing this we can now proceed to test our Echo call.

Test Echo call

As we did before we run the following:

```
~/rest-tutorial$ gradle installApp
~/rest-tutorial$ build/install/rest-tutorial/bin/q2
```

Leave this running and open a new terminal window to try the `/echo` call.

```
curl -i -H "Accept: application/json" -H "Content-Type: application/json"
http://localhost:8080/api/echo
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 24
Server: Jetty(9.3.19.v20170502)

{
  "success" : "true"
}
```

Using the Transaction Manager

As you may have read in our Blog post Eating our own dogfood we like to use the Transaction Manager for almost everything! In this post you can learn how to integrate a REST API with the Transaction Manager. As an example, this is how the Echo class we created in this tutorial could look like if we sent the request to the Transaction Manager to manage for us.

```
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public Response echoGet(
            @HeaderParam("version") String version,
            @HeaderParam("consumer-id") String consumerId,
            @HeaderParam("timestamp") long timestamp,
            @HeaderParam("hash") String hash,
            @Context UriInfo uriInfo,
            @Context SecurityContext sc
    ) throws IOException {
        org.jpos.transaction.Context ctx = createContext(getClass().getSimpleName
().toLowerCase() + "::get");
        ctx.put (Constants.RESTAPI_CREDENTIALS,
          APICredential.builder().version(version)
            .consumerId(consumerId)
            .timestamp(timestamp)
            .hash(hash)
            .nonce(nonce)
            .securityContext(sc).build()
        );
        ctx.put(Constants.URI_INFO,uriInfo);
        Response.ResponseBuilder rb = execute(ctx, Response.Status.OK, (ctx1, resp) ->
{
            put(resp, "version", Q2.getVersion());
            put(resp, "revision", Q2.getRevision());
            put(resp, "timestamp", ctx1.get(Constants.TIMESTAMP));
            put(resp, "security-context", sc);
            put(resp, "principal", sc.getUserPrincipal());
        });
        return rb.build();
    }
```

We use a `RestSupport` helper that implements methods that are used almost on every call such as `createContext` and `execute`.

These methods look like this:

```
    protected Context createContext(String txnName) {
      Context ctx = new Context();
      ctx.put(Constants.TXNNAME, txnName);
      return ctx;
    }

    protected Response.ResponseBuilder execute(Context ctx,
          Response.Status defaultOK, RestAction action, long timeout) {
      int result = queryTxnMgr(ctx, timeout);
      Map<String,Object> resp = new HashMap<>();
      put(resp, "success", result == TransactionManager.PREPARED);
      String errors = "";
      if (result != TransactionManager.PREPARED) {
          errors += ctx.getString(TxnConstants.RC)
                  + (ctx.getString(TxnConstants.EXTRC) != null ? " "
                  + ctx.getString(TxnConstants.EXTRC) : "");
          put(resp, "errors", errors);
      } else if (action!= null) {
          action.execute(ctx,resp);
      }
      return  Response.ok(resp, MediaType.APPLICATION_JSON)
              .status(getStatus(defaultOK, (String) resp.get("errors")))
              .location(((UriInfo) ctx.get(Constants.URI_INFO)).getAbsolutePath());
    }

    protected int queryTxnMgr (Context ctx, long timeout) {
      SpaceFactory.getSpace().out(RESTAPI_TXN_QUEUE, ctx, timeout);
      Integer result = (Integer) ctx.get(TXNRESULT, timeout);
      return result != null ? result : -1;
    }
```

How to get help

If you have questions while trying this tutorial, feel free to contact support@jpos.org, we'll be happy to help.

If you want online assistance, you can join the jPOS Slack, please request an invite.