Customizing your Gateway

Table of Contents

IMPORTANT	This tutorial assumes that you have successfully completed the Implementing a jPOS Gateway tutorial.

The jPOS Gateway tutorial shows you how to use an out-of-the-box binary distribution of jPOS to build an ISO-8583 gateway, but real world implementations usually require adding custom business logic, for validation, routing, logging, etc.

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

Create a project

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

```
git clone https://github.com/jpos/jPOS-template.git gateway ①
cd gateway
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/gateway/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/gateway/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/apr/tutorials/gateway/build/install/gateway/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 work the way it works.

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/gateway`).
- The `src/dist/deploy` and `src/dist/cfg` directories get copied to their destination directory in `build/install/gateway/deploy` and `build/install/gateway/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/gateway` directory named after the project name and version (defined in the main `build.gradle` file).

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

```
gateway-2.1.0.jar
```

If we analyze its content (using `jar tvf gateway-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: gateway
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 gateway-2.1.0.jar
```

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

Gateway Configuration

Now go back to the Implementing a jPOS Gateway tutorial and place the QServer, ChannelAdaptor, MUX and TransactionManager configurations presented there but instead of using the `deploy` directory, you should use the `src/dist/deploy` directory.

For your convenience, you can follow the following script:

```
cd src/dist/deploy
wget http://jpos.org/downloads/tutorials/gateway/10_channel_jpos.xml
wget http://jpos.org/downloads/tutorials/gateway/20_mux_jpos.xml
wget http://jpos.org/downloads/tutorials/gateway/30_txnmgr.xml
wget http://jpos.org/downloads/tutorials/gateway/50_xml_server.xml
cd ../..
```

Now, when you call `gradle installApp && build/install/gateway/bin/q2` you'd be able to run exactly the same configuration shown in the Gateway tutorial, with a nice difference, you're building from the sources, and you can add your custom code to the classpath.

At this point, we suggest you fire a message following the instructions from the gateway tutorial, just to make sure everything works alright before moving to the next section.

Adding your custom code

Adding your custom code is simple now, just create a directory `src/main/java` and place them there.

You probably want to use an IDE, so you can try something like this:

```
mkdir -p src/main/java
gradle idea # (or gradle eclipse if you wish)
```

and you'd be ready to `open gateway.ipr` project.

Let's add for example a TransactionParticipant.

Right now, our the TM configuration (`src/dist/deploy/30_txnmgr.xml`) looks like this:

```
<txnmgr class="org.jpos.transaction.TransactionManager" logger="Q2">
    <property name="queue" value="TXNMGR"/>
    <property name="sessions" value="2"/>
    <property name="max-sessions" value="128"/>
    <property name="debug" value="true"/>

    <participant class="org.jpos.transaction.participant.QueryHost"/>
    <participant class="org.jpos.transaction.participant.SendResponse"/>
</txnmgr>
```

Just the `QueryHost` and `SendResponse` participants. Let's add a `SelectDestination` participant that would allow you to route a transaction to different endpoints.

Create a directory `src/main/java/org/jpos/tutorial` where your `SelectDestination.java` code will sit:

```
mkdir -p src/main/java/org/jpos/tutorial
```

Now edit a file `SelectDestination.java` with code like this:

```
package org.jpos.tutorial;

import org.jpos.core.*;
import org.jpos.iso.ISOMsg;
import org.jpos.transaction.Context;
import org.jpos.transaction.ContextConstants;
import org.jpos.transaction.TransactionParticipant;

import java.io.Serializable;

public class SelectDestination implements TransactionParticipant, Configurable {
    Configuration cfg;

    @Override
    public int prepare(long id, Serializable context) {
        Context ctx = (Context) context;

        ISOMsg m = (ISOMsg) ctx.get(ContextConstants.REQUEST.toString());
        if (m != null && (m.hasField(2) || m.hasField(35))) {
            try {
                Card card = Card.builder().isomsg(m).build();
                String s = cfg.get("bin." + card.getBin(), null);
                if (s != null) {
                    ctx.put(ContextConstants.DESTINATION.toString(), s);
                }
            } catch (InvalidCardException ignore) {
                // use default destination
            }
        }
        return PREPARED | NO_JOIN | READONLY;
    }
    public void setConfiguration (Configuration cfg) {
        this.cfg = cfg;
    }
}
```

Now go back to `30_txnmgr.xml` and add a new participant just before `QueryHost`:

```
    <participant class="org.jpos.tutorial.SelectDestination">
       <property name="bin.411111" value="MYMUX" />
    </participant>
```

Now if you send a regular message to port 8000 as instructed in the Gateway tutorial, the message would go to the configured `jPOS-AUTORESPONDER` MUX, but if you care to add a field 2 (Primary Account Number) with a value of `4111111111111111` (valid LUHN, configured BIN 411111), then you won't get a response, but if you check the `log/q2.log` you'll see something like this:

```
      RESULT:
       <result>
         <fail>
           [MISCONFIGURED_ENDPOINT] o.j.t.p.QueryHost.prepare:66 MUX 'mux.MYMUX' not
found
         </fail>
       </result>
```

The system tried to route the transaction to a non existent MUX called `MYMUX`.

If you add it (`20_mymux_mux.xml` pointed to a channel), you should be able to get the router going.

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.