**Demystifying JINI: Dynamic Code Download**

In the 1st part of the [Demystifying JINI series](http://palashray.com/java/jini/), we had explored how to write a bare bones non secured client/server: <http://palashray.com/demystifying-jini-non-secure-server-part-1/>

In the 2nd part, we had shown how to Spring-ify it: <http://palashray.com/demystifying-jini-non-secure-server-part-2/>

In the latest increment, we will explore how to handle dynamic code download.

I have been working with JINI for the past couple of years. And the more I explore its features, its versatility, the more I come across new ones. This project was built with the famous “[**Eight Fallacies of Distributed Computing**](http://en.wikipedia.org/wiki/Fallacies_of_distributed_computing)”. So, needless to say, it does not disappoint in terms of scalability or reliability.

**Why need dynamic code download?**

At the heart of JINI lies its flexibility. It decouples the interface from the implementation (of the Proxy), and makes it independent of the transport protocol as well. All these details are transparent to the end user.

However, there needs to be a way in which the Stubs are downloaded. These stubs will actually help the remote client to connect to the server, and make calls on the remote JVM. This is where dynamic code download is needed. And this comes out of the box with JINI.

**Why should I care about dynamic code download?**

The dynamic code download is a very powerful feature. In normal client/server distributed architecture, the client and the server agrees on a common API. This is commonly known as the **shared module**, be it JEE or RMI. And it is mandatory for us to bundle the shared module with the server as well as the client modules. Otherwise, the application will blow up with hosts of NoClassDefFoundError-s.

But, with this interesting feature in JINI, the client does not even have to depend on this shared module at all. The reason is, it can be downloaded when doing the lookup. Of course, every feature comes with its costs, and in this case, the only downside to this is, we need to use reflection to query the remote service. Either that, or use a scripting language like Groovy.

**How does it work?**

To enable dynamic code download, we need to specify the code base property (similar to the Applet's) on the JINI server. When the client connects to the server via lookup, this will be available to the JINI client. In its simplest form, it can be as below:

System.*setProperty*("java.rmi.server.codebase", "http://localhost:8080/rmi-service-api-1.0.jar");

This should be called while starting the JINI server.

**Getting started**

Download the folder: <https://github.com/paawak/blog/tree/master/code/jini/jini-services>

cd into it and first start up the reggie by using the **./start-reggie.sh**

Then, again, we need to start a http server to serve the jars for dynamic code download. Use the **./httpd.sh** to start that server. This server starts at port 8080 and makes any jar placed inside of the **service-interface** directory, available over [http://localhost:8080](http://localhost:8080/). This is a small utility provided y JINI, in production, we would need a more secured web server.

**The Service API**

**public** **interface** BankDetailService **extends** Remote {

Map<String, List<BankDetail>> getBankDetails(BankDetailGroups group) **throws** RemoteException;

}

The API jar is built and put in the **service-interface** directory above to make it available for dynamic downloads. Note that it is bundled with the Server, but **not** with the Client.

**The Server**

We have built this on top of the Spring-ified server that we saw in [part 2](http://palashray.com/demystifying-jini-non-secure-server-part-2/).

**public** **class** SpringNonSecureRmiServer {

**private** **static** **final** String ***CODE\_BASE\_URL*** = "http://localhost:8080/";

**private** **static** **final** String[] ***JARS\_DOWNLOADED\_DYNAMICALLY*** = **new** String[] { "rmi-service-api-1.0.jar" };

**private** **static** String getRmiServerCodebase() {

StringBuilder urlBuilder = **new** StringBuilder(200);

// the urls are separated by space

**for** (String jar : ***JARS\_DOWNLOADED\_DYNAMICALLY***) {

urlBuilder.append(***CODE\_BASE\_URL***).append(jar).append(" ");

}

urlBuilder.setLength(urlBuilder.length() - 1);

**return** urlBuilder.toString();

}

**public** **static** **void** main(String[] args) {

String codebase = *getRmiServerCodebase*();

// set the below property to enable dynamic code download

System.*setProperty*("java.rmi.server.codebase", codebase);

**try** (ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext("server-application.xml")) {

System.***out***.println("SpringNonSecureRmiServer.main(): " + "The server is ready");

}

}

}

Note that its is almost similar, just has the extra line setting the **rmi-codebase property**.

**The Client**

Since the client does not have the API as its dependency, we have to use reflection. But remember to use the RMIClassLoader to load classes from the API jar. Let us demo a Swing app which uses the remote BankDetailService to fetch data to be displayed on the JtreeTable.

**The Main Class**

**public** **class** SpringNonSecureRmiClient {

**public** **static** **void** main(String[] args) **throws** Exception {

System.*setProperty*("java.security.policy", System.*getProperty*("user.home") + "/jini/policy.all");

// the below line is put only for debugging purposes, its not needed, as

// the default class loader is good enough

System.*setProperty*(RMIClassLoaderSpi.**class**.getName(), DelegatingRMIClassLoader.**class**.getName());

**if** (System.*getSecurityManager*() == **null**) {

System.*setSecurityManager*(**new** SecurityManager());

}

**try** (ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext("client-application.xml")) {

Remote bankDetailService = context.getBean("bankDetailService", Remote.**class**);

*showFrame*(bankDetailService);

}

}

**private** **static** **void** showFrame(Remote bankDetailService) {

**for** (LookAndFeelInfo info : UIManager.*getInstalledLookAndFeels*()) {

**if** ("Nimbus".equals(info.getName())) {

**try** {

UIManager.*setLookAndFeel*(info.getClassName());

} **catch** (ClassNotFoundException | InstantiationException | IllegalAccessException | UnsupportedLookAndFeelException e) {

**throw** **new** RuntimeException(e);

}

**break**;

}

}

EventQueue.*invokeLater*(() -> {

**new** GroupingDemoFrame(bankDetailService).setVisible(**true**);

});

}

}

**The Method to Query the Remote Service**

**private** Map<String, List<?>> queryRemoteService(String bankDetailGroupsEnumString) {

Class<?> bankDetailGroupsEnumClass;

**try** {

bankDetailGroupsEnumClass = RMIClassLoader.*loadClass*("http://localhost:8080/rmi-service-api-1.0.jar", "com.swayam.demo.rmi.dto.BankDetailGroups");

} **catch** (MalformedURLException | ClassNotFoundException e) {

**throw** **new** RuntimeException(e);

}

Field field;

**try** {

field = bankDetailGroupsEnumClass.getDeclaredField(bankDetailGroupsEnumString);

} **catch** (NoSuchFieldException | SecurityException e) {

**throw** **new** RuntimeException(e);

}

Object bankDetailGroupsEnumValue;

**try** {

bankDetailGroupsEnumValue = field.get(bankDetailGroupsEnumClass);

} **catch** (IllegalArgumentException | IllegalAccessException e) {

**throw** **new** RuntimeException(e);

}

Method getBankDetails;

**try** {

getBankDetails = bankDetailService.getClass().getDeclaredMethod("getBankDetails", bankDetailGroupsEnumClass);

} **catch** (NoSuchMethodException | SecurityException e) {

**throw** **new** RuntimeException(e);

}

**try** {

@SuppressWarnings("unchecked")

Map<String, List<?>> groupedBankDetails = (Map<String, List<?>>) getBankDetails.invoke(bankDetailService, bankDetailGroupsEnumValue);

**return** groupedBankDetails;

} **catch** (IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {

**throw** **new** RuntimeException(e);

}

}

**Sources**

The sources for the **server** be found here:

<https://github.com/paawak/blog/tree/master/code/jini/unsecure/plain/spring-rmi-server>

The sources for the **client** be found here:

<https://github.com/paawak/blog/tree/master/code/jini/unsecure/plain/rmi-client-dynamic-code-download>

You can download the sources and then run:

**mvn clean install -P create\_dist**

This will create the distribution inside the **target/appassembler** directory. You can go inside the **appassembler/bin** directory and run the **./launch\_swing\_app** to launch the application.

**Reference**

<http://www.softwarematters.org/jini-intro.html>