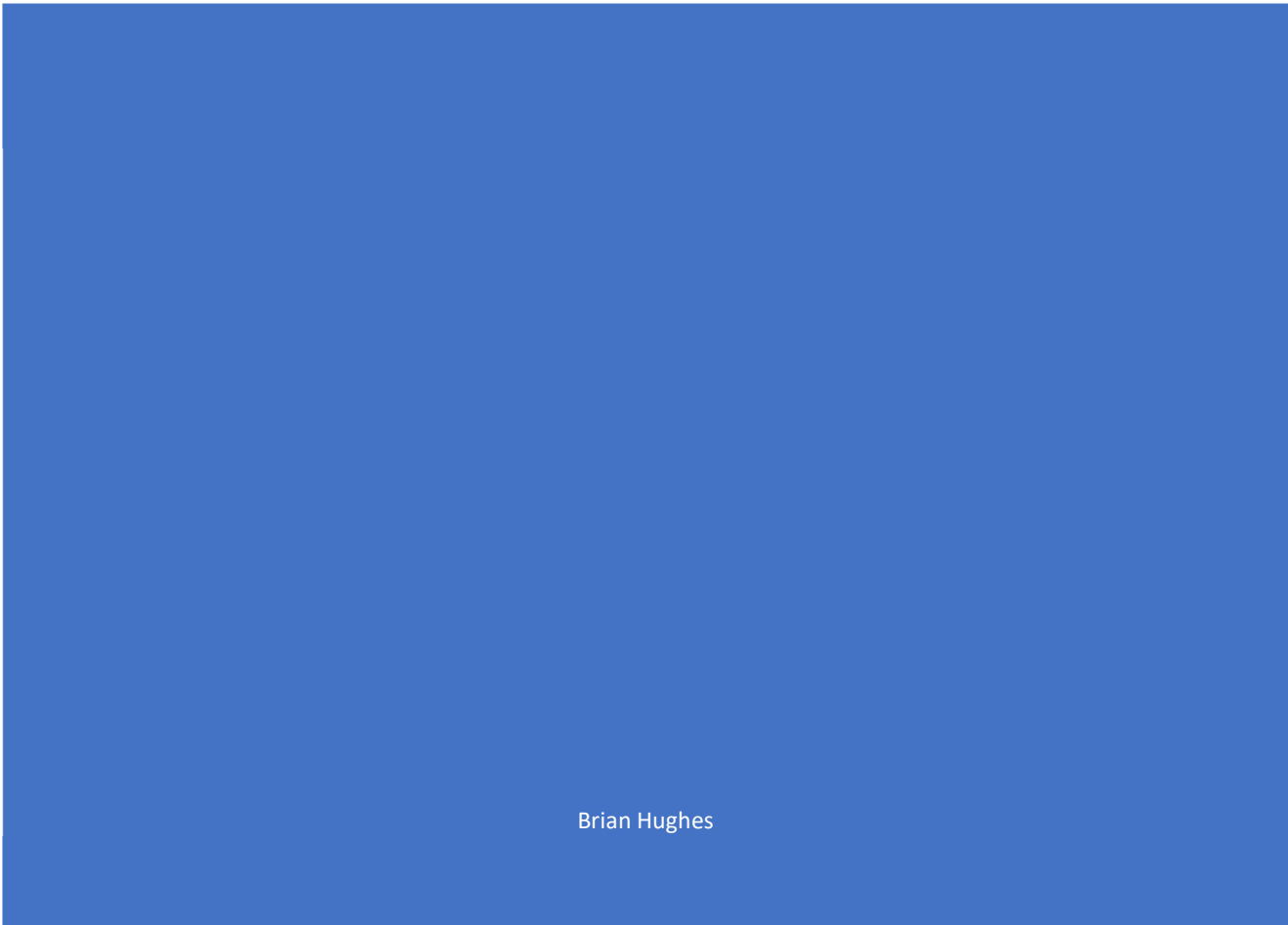




USER DEFINED ROUTINE DEVELOPMENT WITH J/FOUNDATION



Brian Hughes

Table of Contents

Contents

Prerequisites.....	3
What is J/Foundation?.....	3
How does J/Foundation work?.....	3
How to read this material.....	3
Introduction to technology used in this lab.....	4
Virtual Machine	4
Command Prompt	4
Java 8	4
Visual Studio code	4
Docker.....	5
Getting terminals started	7
Downloading the sample project	7
Getting your Informix docker container running.....	8
Stores demo.....	8
Setting up for J/Foundation.....	11
Registering a user defined routine that already exists in the server	12
Building our first Simple JUDR using system UDRs	17
System UDRs.....	17
Getting started with the sample project:	17
Conclusion	23

Prerequisites

You should have knowledge of the following:

- How to start/stop and basic Informix server operations
- Basic command prompt/terminal usage (executing commands, copying files, reading terminal output)
- Using an editor (vi/vim) to edit files

What is J/Foundation?

J/Foundation is a means of writing server-side extensions to Informix using the Java programming language. This is similar to writing extensions using the C programming language.

J/Foundation is a small performance tradeoff (verses C code) to allow you to write richer, faster extensions to Informix using a language that is more powerful than pure SPL.

NOTE: This Lab shows several ways to build and execute Java UDRs using 14.10.XC1 or higher servers. The exercises shown here might not be backwards compatible with prior server versions.

How does J/Foundation work?

Informix ships with its own Java runtime (Java 8 for 14.10). Informix has a set of C and Java libraries that provide interaction and data exchange between the core server and the Java Virtual Machine

Static methods written in Java that have corresponding mappable data types (think char to String) can be declared and executed inside of a SQL statement as a function or procedure

The Java virtual Machine runs in its own VPCLASS named JVP which must be running on the server

How to read this material

There are a couple conventions used in this document.

Bounded text boxes indicate commands to be issued in a terminal session.

```
Commands and expected output will be shown here.
```

Each command is prefixed by a prompt. The prompt will give an indication of what terminal/session you should execute a command in. This is important as to distinguish between commands ran on the virtual machine and commands to be executed in a docker container. Instructions later will tell you how to set up your terminals for docker.

```
$(VM)> This is the prompt to execute a command inside the terminal for the  
virtual machine itself
```

`$(DOCKER)>` This is the prompt for a command to be executed inside of a docker container.

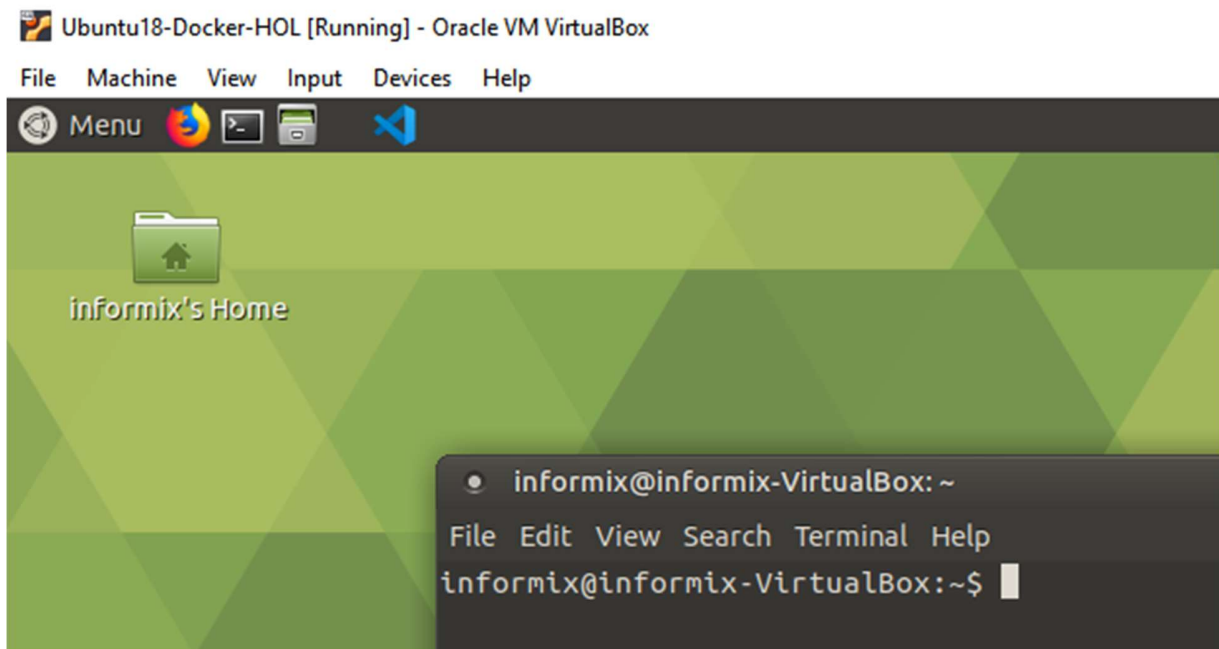
Introduction to technology used in this lab

Virtual Machine

For this tutorial we will be writing and running software inside of a virtual machine. This virtual machine is a Linux OS which is already installed on your laptop. All work will be done inside of this virtual machine (VM).

Command Prompt

We will use a command prompt/terminal to perform file system actions and execute commands. You can open a terminal by clicking the terminal icon on the top left of the screen



Java 8

Java 8 development kit is installed on the virtual machine and the Java 8 runtime is installed on both the virtual machine as well as inside the Informix Docker containers

Visual Studio code

Although you can choose to use VIM (terminal based editor) you might find it useful to do editing and file viewing through the installed Visual Studio Code editor. You can open the editor from the terminal or from the same top left bar.

To start Visual Studio Code from a directory in a terminal window, execute the following command

```
$ [VM]> code .
```

Ellipses (. . .) You might see 3 dots together in some output. This means some content was abbreviated for clarity or to hide unimportant output.

Docker

Your virtual machine comes with a pre-installed Docker runtime and the Informix developer edition docker image downloaded. If you are not familiar with docker consider it a small virtual machine with its own environment that you will access to perform work on the server.

Setting up your environment

Logging into the VM

If your virtual machine is not already running start the Virtual Box application. Start the VM named "Ubuntu18-Docker-HOL", by starting the Oracle virtual box program and double clicking the VM in the list.

The user and password for the VM is

User: informix

Password: informix

Getting terminals started

You need 2 terminal sessions for this project. One for the sample project on the virtual machine and one for the Informix server running inside of a docker container.

When you start your terminal sessions, we are going to run them as root. Execute this command in both your terminal sessions

```
$[VM]> sudo bash
```

Cleaning the environment

In case another lab was run before you, you can reset the environment by executing the following commands

```
$[VM]> cd lab-darint/compose-hq-demo
```

```
$[VM]> docker-compose down -v
```

Downloading the sample project

You can download the sample project code from github. Then you can go into the j-foundation directory. This is where you will start for one of your terminal sessions for this lab. The other terminal will be configured for the Informix server.

```
$[VM]> cd $HOME
$[VM]> git clone https://github.com/informix/informix-db-examples.git
$[VM]> cd informix-db-examples/j-foundation
$[VM]> chmod +x gradlew
```

Getting your Informix docker container running

To run Informix inside of docker you need to run the following commands in a **new Terminal session**. **Leave the old session running. It's easiest to have two terminal windows and switch between them.**

```
$[VM]> sudo bash
$[VM]> run.de
$[VM]> dbash server
$[DOCKER]> onstat -

IBM Informix Dynamic Server Version 14.10.FC2DE -- Initialization -- Up
00:00:05 -- 65480 Kbytes

$[DOCKER]> onstat -

IBM Informix Dynamic Server Version 14.10.FC2DE -- On-Line -- Up 00:00:43 -
- 90056 Kbytes
```

It can take a few minutes for Informix to be in 'On-line' mode keep checking periodically with onstat - until the server says it is 'On-line'

From now on, when you see the prompts say `$(DOCKER)>` then run that command in this docker enabled terminal. If you see `$(VM)>` then use the original terminal we setup for the VM

Stores demo

We will be using the built in Stores Demo database for these exercises.


```
$(DOCKER)> dbaccessdemo
```

```
. . .
```

The creation of the demonstration database is now complete. The remainder of this script copies the examples into your current directory.

Press "Y" to continue, or "N" to abort.

```
> N
```

Setting up for J/Foundation

Setting up for J/Foundation

Take a look at the J/Fondations file structure

```
[$[DOCKER]> cd $INFORMIXDIR/extend/krakatoa
[$[DOCKER]> ls -al
-rw-r--r-- 1 informix informix      1688 Aug 20 21:39 .jvpprops.template
drwxr-xr-x 2 informix informix     4096 Sep 11 15:33 examples
-rw-r--r-- 1 informix informix 1585162 Aug 20 21:39 ifxlang.jar
-rw-r--r-- 1 informix informix      526 Aug 20 21:39 informix.policy.std
lrwxrwxrwx 1 root      root         25 Sep 11 15:33 jre ->
/opt/ibm/informix/jvm/jre
-rw-r--r-- 1 informix informix 3046475 Aug 20 21:25 krakatoa.jar
-rwxr-xr-x 1 informix informix   84192 Aug 20 21:39 libjvp.so
-rwxr-xr-x 1 informix informix  706688 Aug 20 21:39 libjvp_g.so
-rwxr-xr-x 1 informix informix  180712 Aug 20 21:39 lmjava.so
-rw-r--r-- 1 informix informix    1673 Aug 20 21:39 logback.xml
-rw-r--r-- 1 informix informix     498 Aug 20 21:39 update_jars.sql
```

There are some important files to note.

krakatoa.jar	The main java library. Bridges between the server code and your java extensions
.jvpprops.template	A set of JVP properties you can set to change the behavior of the JVP process
jre	Link to a Java runtime environment. Can actually be swapped out for different java (with lots of caveats and some support concerns)
*.so	C programming shared libraries that work with krakatoa.jar to bridge functionality between the server and your java extensions
logback.xml	The main logging description file. Use this to change where and how the Java logs messages and errors

Please open and view the .jvpprops.template and the logback.xml files. Note the template file is a key/value properties file while the logback.xml file is XML based.

Looking at the logback.xml see where it is preconfigured to log to.

```
[$[DOCKER]> cd $INFORMIXDIR/extend/krakatoa
[$[DOCKER]> cat .jvpprops.template
[$[DOCKER]> cat logback.xml
. . .
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
  <append>true</append>
  <file>${INFORMIXDIR}/tmp/jvp.log</file>
. . .
```

To see more into what J/Foundation does on the Java side we will enable DEBUG logging for this exercise. Open up the logback.xml file and edit this line and restart Informix for it to take effect.

```
[$DOCKER]> vi $INFORMIXDIR/extend/krakatoa/logback.xml
Line 37  <root level="INFO">
Change "INFO" to "DEBUG"

Save the file.

[$DOCKER]> onmode -yuk
[$DOCKER]> oninit
```

Creating the Java virtual processor for Informix (JVP)

We need to create a new virtual processor for Java to run in. With the Informix server running execute the following command.

```
[$DOCKER]> onmode -p +1 JVP
```

Registering a user defined routine that already exists in the server

J/Foundation comes with a set of already coded Java routines. As with any user defined routine, even though it is coded it must be defined as a function in each database you wish to invoke the function.

This allows you some flexibility in naming the function at the SQL level as well as picking (within reason) the SQL type you want to see returned. For example you might want to name a function to avoid a clash with an existing name and you might want to use a char(128) rather than a VARCHAR(128). Both correspond to the String data type in Java so such alterations can be made at the function declaration.

Let's define our first function. Remember this code is already done. It's stored in the \$INFORMIXDIR/extend/krakatoa/krakatoa.jar and. We are just defining the name for it so we can reference it in a SQL statement. Later on, we will add our own Java code.

```
[$DOCKER]> dbaccess stores_demo -
> create function UUID() returning char(36) external name
'com.informix.judrs.IfStrings.getUUID()' language java;
Routine created.
```

Now we have the Java code (it's already installed for us) and we have registered the function. But J/Foundation is not running yet. It will only start when we execute the function.

Let's try this now. Using the same dbaccess session you have running

```
$[DOCKER]> dbaccess stores_demo -  
> execute function UUID();  
(expression)  
  
5455f8ba-2121-494d-b6a1-2f472487ce1f  
  
1 row(s) retrieved.
```

It might take a minute to return the first time. Congratulations! You just executed your first Java user defined routine. Now let us check to see what the server did.

```

$[DOCKER]> onstat -m

IBM Informix Dynamic Server Version 14.10.FC2DE -- On-Line -- Up 00:02:06 -
- 106440 Kbytes

Message Log File: /opt/ibm/data/logs/online.log

20:51:37 Booting Language <java> from module
<${INFORMIXDIR}/extend/krakatoa/lmjava.so>

20:51:37 Loading Module <${INFORMIXDIR}/extend/krakatoa/lmjava.so>

20:51:37 The C Language Module
</opt/ibm/informix/extend/krakatoa/lmjava.so> loaded

20:51:37 Loading Module <com.informix.judrs.IfStrings>

20:51:37 Got the mutex

20:51:37 LD_LIBRARY_PATH=/opt/ibm/informix/extend/krakatoa/jre/bin/j9vm

20:51:37 VM args[0]= -Xss512k

20:51:37 VM args[1]= -
Djava.security.policy=/opt/ibm/informix/tmp/JVM_security

20:51:37 VM args[2]= -Xms16m

20:51:37 VM args[3]= -Xmx16m

20:51:37 VM args[4]= exit

20:51:37 VM args[5]= abort

20:51:37 VM args[6]= -
Djava.class.path=/opt/ibm/informix/extend/krakatoa/krakatoa.jar:/opt/ibm/in
formix/extend/krakatoa

20:51:37 Successfully created Java VM.

```

Here we see the server starting up the Java virtual machine for the first time. Once this is started it will stay running until you restart the instance or forcefully shutdown the JVP process running in Informix.

We can check that the jvp process has done some work. Run the onstat command below and note the process 'jvp' which is hosting our Java virtual machine.

```
[$[DOCKER]> onstat -g glo
```

```
IBM Informix Dynamic Server Version 14.10.FC2DE -- On-Line -- Up 00:56:48 -  
- 106440 Kbytes
```

```
...
```

```
Virtual processor summary:
```

class	vps	usercpu	syscpu	total
cpu	1	4.94	0.77	5.71
aio	1	0.14	0.40	0.54
lio	1	0.03	0.03	0.06
pio	1	0.00	0.06	0.06
adm	1	0.09	0.33	0.42
msc	1	0.00	0.00	0.00
jvp	1	6.54	2.15	8.69
fifo	1	0.02	0.04	0.06
total	8	11.76	3.78	15.54

Building our own Java User Defined Routine

Building our first Simple JUDR using system UDRs

There are 2 ways to get Java code into the server, each with advantages over the other. For this exercise we will be using System UDRs and adjusting the server's CLASSPATH configuration to add new functionality.

System UDRs

System UDRs are routines written in Java that ANY database can declare and access. The code is compiled and referenced at the start of the Java virtual machine. It is the easiest to understand but requires server downtime to add or upgrade functionality.

Getting started with the sample project:

You have been provided a sample project on your virtual machine. You will not need to write any Java code for this example, it has been provided

This project exists on your virtual machine. You can choose to use the terminal to navigate the files and directories or use Visual Studio Code to navigate the project.

You should see a few directories and files in the folder. The ones of importance are

- `src/` -- source files are here
- `libs/` -- Library files we might use when building a Java UDR
- `build.gradle` – build script to make building java libraries easy/easier

```
# if you are not already in the directory with the project
$[VM]> cd informix-db-examples/j-foundation
$[VM]> ls -l
-rw-r--r-- 1 BRIAN.HUGHES 1049089 303 Sep 18 17:17 build.gradle
drwxr-xr-x 1 BRIAN.HUGHES 1049089 0 Sep 16 17:11 gradle/
-rwxr-xr-x 1 BRIAN.HUGHES 1049089 5960 Sep 16 17:11 gradlew
-rw-r--r-- 1 BRIAN.HUGHES 1049089 2942 Sep 18 17:17 gradlew.bat
drwxr-xr-x 1 BRIAN.HUGHES 1049089 0 Sep 18 17:17 libs/
-rw-r--r-- 1 BRIAN.HUGHES 1049089 281 Sep 17 14:53 README.md
-rw-r--r-- 1 BRIAN.HUGHES 1049089 38 Sep 17 14:57 settings.gradle
drwxr-xr-x 1 BRIAN.HUGHES 1049089 0 Sep 17 14:54 src/
```

Examine the primary build file (build.gradle)

Note the inclusion of the judr.properties and the dependency on the krakatoa.jar file which was copied from \$INFORMIXDIR/extend/krakatoa/krakatoa.jar. Many JUDRs won't need to use any functionality in krakatoa.jar, but if you do create connections to the database inside your JUDR, then you would want it.

```
$[VM]> cat build.gradle

plugins {
    id 'java-library'
}

dependencies {
    file('libs/krakatoa.jar')
}

jar {
    from('configuration') {
        include 'judr.properties'
    }
}
```

Digging further into this project we can see some Java classes and methods that look like good candidates for a user defined routine in the server. Below is a view of the first routines we are going to install.

```
$[VM]> cat src/main/java/com/informix/judrs/DateTime.java
...
    public static String getLongDateTime() {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new
Date());
    }

    public static long getUnixTimestamp() {
        return new Date().getTime();
    }
...

```

We can build this project by executing gradlew clean jar to build the project. Then you can look in the build/libs directory to see your new jar file.

```

$[VM]>./gradlew clean jar

BUILD SUCCESSFUL in 3s
3 actionable tasks: 3 executed

$> ls -l build/libs
total 4
-rw-r--r-- 1 BRIAN.HUGHES 1049089 1993 Sep 20 13:37 judr-examples.jar

```

Once the jar is built you can copy it into the docker container

```

$[VM]> docker cp build/lib/judr-examples.jar server:/tmp

```

Now we need to go back inside the docker container and install the udrs. To start with we need to include our new jar file in the CLASSPATH that the server has set in its ONCONFIG file. We will modify that file.

```

$[DOCKER]> vi $INFORMIXDIR/etc/$ONCONFIG
#Edit this line. Type /JVPCLASSPATH <enter> to have VI search for the start
of the JVP settings.

JVPCLASSPATH  $INFORMIXDIR/extend/krakatoa/krakatoa.jar

Add on the path to our new jar file so it looks like this
JVPCLASSPATH  $INFORMIXDIR/extend/krakatoa/krakatoa.jar:/tmp/judr-
examples.jar

Save the file

$[DOCKER]> onmode -yuk;
$[DOCKER]> oninit;
$[DOCKER]> onmode -p +1 JVP
# we need to put back the JVP if we restart, you are free to set it in the
$ONCONFIG file permanently if you wish

```

Notice we must restart the server for this to take effect. This is the downside to system level UDRs. Any changes to the source code will require a server restart. But this is very good for stable UDRs that don't need many changes.

Now that we have added our code, we can register our UDR

```

$[DOCKER]> dbaccess stores_demo -
> CREATE FUNCTION javadate() returning varchar(255) external name
'com.informix.judrs.DateTime.getLongDateTime()' language java;

> execute function javadate();

(expression)  2019-09-19 17:25:50.841

> CREATE TABLE datetest( d datetime year to fraction(3));

Table created.

> INSERT INTO datetest values(javadate());

1 row(s) inserted.

> select * from datetest;

d
2019-09-19 17:26:48.768

1 row(s) retrieved.

```

You can use the keyboard combo ctrl-c to quick the dbaccess program.

Did you know. Informix does not have a version of a "current date/time" that goes to the millisecond? It stops at seconds. Now you have a function that gives milliseconds!

Congratulations, you just ran your own custom Java UDR. You might start to see that from this point, you can add more logic and complexity to the Java code, include additional libraries and quickly expand the capabilities of the server.

Next, we will see how to install your UDRs into individual databases without restarting the server.

Installing a Java UDR into a
specific database

So far, we have installed our Java UDRs into the entire system. Which make them available to all databases, but requires a restart if we need to do any changes.

Another way to install Java UDRs is to install them into an individual database. This requires an sbspace (already setup for you) to store your JAR file.

Let's go to back to our VM and look again at our java udr project code. There is not one, but two java files defined.

```
[$[VM]> ls src/main/java/com/informix/judrs/  
CustomUUID.java DateTime.java  
  
[$[VM]> cat src/main/java/com/informix/judrs/CustomUUID.java  
. . .  
  
public class CustomUUID {  
    public static String getDateUUID() {  
        Random r = new SecureRandom();  
        Date d = new Date();  
        return "" + (d.getTime() / 1000) + "-" + r.nextLong();  
    }  
}
```

Let's install and register this new UDR. But before we do, remember we mentioned briefly about a properties file being included in the JAR (judr.properties)? This file is used to automatically register functions when you install a JAR this way. Note this does not work for the System UDR's that are on the Informix CLASSPATH that we learned in the prior section. This file is only discovered when we install a UDR into a database.

```
[$[VM]> cat configuration/judr.properties  
  
register-class-name: com.informix.judrs.CustomUUID  
register-class-prefix: j_  
register-class-default-properties: parallizable, variant  
register-class-default-grantee: public, informix
```

Notice several properties are defined here. Instead of writing SQL statements to issue the CREATE FUNCTION... and any corresponding permission granting, they can be defined here, for as many classes you have in your JAR file. For each class defined (we have one, CustomUUID) J/Foundation will automatically scan for public static java methods and auto-register them for you! It will also auto-deregister them if you uninstall the JAR file. Very handy!

Now, if you did the prior section, you already have built this and it is sitting in the docker container in the /tmp directory. We can reuse that work.

Go back to your docker session.

```
[$DOCKER]> dbaccess stores_demo -  
> execute procedure sqlj.install_jar('file:/tmp/judr-examples.jar', 'judr-  
examples', 1);
```

What does this do ? First it creates some tables in your database to hold UDR information. Then it inserts this jar file into one of those tables. Next it scans the JAR file for that judr.properties file and uses that to scan for matching classes and methods to register.

You can view the details by looking at the jvp.log file created in \$INFORMIXDIR/tmp. Look for entries relating to "Registering class"

```
[$DOCKER]> cat $INFORMIXDIR/tmp/jvp.log  
.  
.  
.  
16:16:23.514 [jvp-worker-1] INFO i.jvp.dbapplet.impl.JarHandler -  
Installing jar stores_demo.informix.judr-examples...  
16:16:23.525 [jvp-worker-1] INFO i.jvp.dbapplet.impl.JarHandler -  
Processing file: judr.properties  
16:16:23.528 [jvp-worker-1] INFO i.jvp.dbapplet.impl.JarHandler -  
Registering class: com.informix.judrs.CustomUUID  
.  
.  
.
```

Now that we have installed our jar, we can test out the new java udr

```
[$DOCKER]> dbaccess stores_demo -  
> execute function j_getDateUUID();  
  
(expression) 1568996723-1151730817033292052  
  
Ctrl-c  
  
#this part is just to show part of the UUID we created is a real UNIX  
timestamp  
> date -d @1568996723  
Fri Sep 20 16:25:23 UTC 2019
```

Conclusion

After completing this lab, you now see how to setup your server for Running Java UDRs as well as how to use a sample project and install new UDRs into the system.