SEEBEYOND®
beyond integration

Using Properties for runtime ICAN 5.0.x JCD Configuration

Michael Czapski

OCTOBER 2004

SEEBEYOND 2004
HORIZONS
ASIA PACIFIC

Table of Contents

Introduction

One of the questions, frequently asked by developers using ICAN Java Collaboration Definitions, is ‘How do I pass runtime values to a collaboration?’.

The quick answer to this question usually is “can’t do it”. On reflection the answer is supplemented by “not if you need to comply with the EJB 2.0 spec.”. On further prodding the answer becomes “there are a bunch of ways, all of which involve programming and most of which violate EJB 2.0 specification, making the solution non-J2EE-compliant and, potentially, liable to be broken by a product upgrade”. One must further say that there may be legitimate reasons to break the rules, so long as the rule breaker understands and accepts the implications.

Given this, why would one bother? There may be reasons why one would need to pass runtime values to Java collaboration.

The most obvious one is the need to avoid hard-coding configuration information. The reason might be that the site cannot afford to re-activate the solution containing the collaboration each time a volatile configuration item, such as file location, server port number, password, or similar piece of information used at runtime, changes. Another reason might be that a change to a Java collaboration is a change and the code change must go through change control procedures, testing, whatever, making it hard to respond to events in a timely manner and increasing the likelihood of introducing issues to a running production system.

This document discusses some of the ways of setting and passing runtime configuration values to Java Collaboration Definitions.

IS JVM Properties

A Java program can access runtime properties of the JVM, like “user.home”, “user.name”, “os.name”, that are either ‘built-in’ or specified on the invocation command line.

ICAN 5.0.4 provides an Integration Server property that allows specification of additional “JVM Args”. These JVM Args are passed to the JVM as part of the command and are added to “system properties” that can be queried at runtime.

Setting

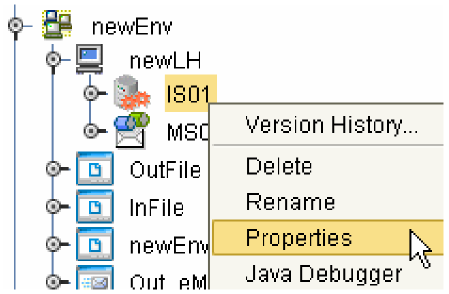
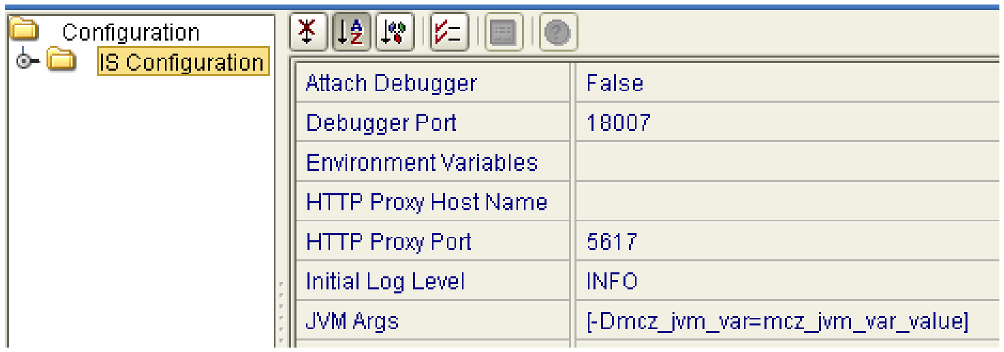
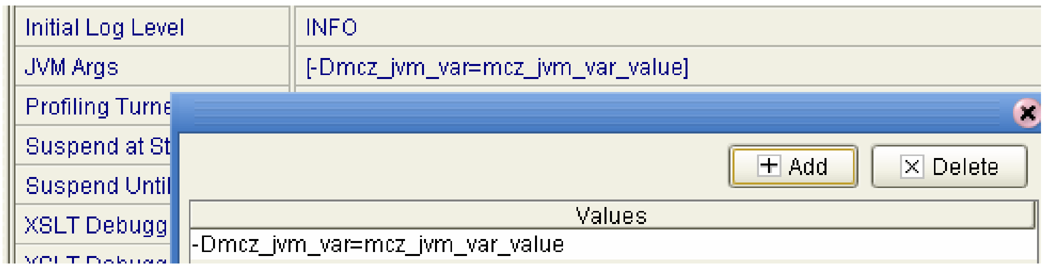


Figure 1 Integration Server in the Environment



Attach Debugger	False
Debugger Port	18007
Environment Variables	
HTTP Proxy Host Name	
HTTP Proxy Port	5617
Initial Log Level	INFO
JVM Args	[-Dmcz_jvm_var=mcz_jvm_var_value]

Figure 2 Integration Server Properties



Initial Log Level	INFO
JVM Args	[-Dmcz_jvm_var=mcz_jvm_var_value]

Values
-Dmcz_jvm_var=mcz_jvm_var_value

Figure 3 Adding a name/value pair

Note that the name/value pair takes the form “-Dname=value”.
Whilst there appears to be no limit to the number of name/value pairs that one can specify one ought to exercise caution as there may be limits to the length of the OS

command line that can be constructed. These name/value pairs get added to the java command that runs the Integration Server class.

Propagating to Runtime

Once set, the value can only be propagated to the runtime environment by performing 'Apply' for the Logical Host to which the Integration Server belongs, or by shutting down the Logical Host and re-bootstrapping it with the '-f' options, forcing reload of the entire deployment. In effect doing 'apply' causes re-start of the solution.

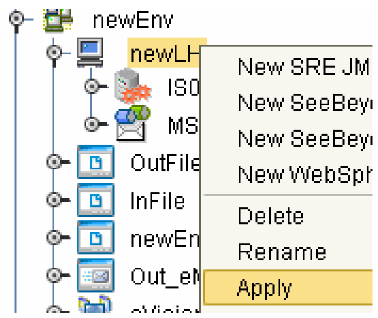


Figure 4 Apply changes to IS Properties

Using in a JCD

Suppose you have 2 variables defined in your JVM Agrgs property, mcz_jvm_var and mcz_jvm_arg2. To access their values at runtime from a JCD you would add code similar to the following:

```
String mcz_jvm_var = System.getProperty( "mcz_jvm_var" );
String mcz_jvm_arg2 = System.getProperty( "mcz_jvm_arg2" );
```

You would then do with the String values whatever you need.

Issues

Uncertainty as to how many properties can be set this way has already been mentioned. To err on the side of caution one would set few properties with short values.

The properties, set through the "JVM Args" property of the Integration Server, are 'static' in the sense that they cannot be changed at runtime. Furthermore a deployment, deployed to the particular integration server, must be re-deployed/reactivated or the logical host has to be re-loaded for the changes to take effect. This effectively shuts down the logical hosts and re-boots it.

Static Startup Properties

It may be that a Java collaboration requires a set of configuration values to be passed to it at startup. The properties are kept in a properties file somewhere in the file system. Where one keeps the properties file determines how one can access it at runtime.

If the properties file sits in some arbitrary place in the file system one must tell the JCD where to find it. One can either hardcode the path in the JCD or pass its value at runtime using a “JVM Args” property to the Integration Server.

In 5.0.4, if the properties file sits in a directory that is included in the IS classpath it can be accessed by just the file name, which needs be hardcoded, derived or passed via “JVM Args” property. Some reasonable places to put the properties file are <LogicalHostRoot>/logicalhost/logconfigs/IS_<integrationServerName> and <LogicalHostRoot>/logicalhost/stcis,

In 5.0.5 one must add the directory in which the properties file lives to the classpath explicitly by setting the ‘Startup Classpath Prepend’ or ‘Startup Classpath Append’ property of the Integration Server so that it includes this directory.

Setting

Create a properties file, called my.properties, in one of the recommended locations, with the following text:

```
prop_01 = This is prop_01 value
prop_02 = 123456
```

Propagating to Runtime

There is nothing to propagate to the runtime environment unless one uses the JVM Args to pass the file location to the JCD. If so then the same steps apply as in the previous discussion. If the properties file is accessed via the classpath, the file is already where it needs to be.

Using in a JCD

Create a new Java Collaboration Definition, jcdProperties02, with whatever input, manipulation and output OTDs might be required.

In “Source Code Mode” add the following Java code just before the ‘receive’ method:

```
private static final String sPropFileName = "my.properties";
private static String prop_01 = null;
private static String prop_02 = null;

public jcdProperties02()
       throws Exception
{
    java.util.Properties props = new java.util.Properties();
    try {
        props.load( this.getClass().getClassLoader().getResourceAsStream( sPropFileName ) );
        prop_01 = props.getProperty( "prop_01" );
        prop_02 = props.getProperty( "prop_02" );
    } catch ( java.io.IOException ioe ) {
        String sMsg = "Properties File [" + sPropFileName + "] not found in classpath ";
        alerter.critical( sMsg );
        throw new Exception( sMsg, ioe );
    }
}
```

Note the sPropFileName constant hardcoding the name of the properties file.
Note private static String variables that will hold the values of the two properties of interest.

Note that JCD’s public constructor is used to read the properties file, which is expected to reside in the classpath, into a Properties object, than the individual properties are extracted and set as values of the class variables.
In the ‘receive’ method one accesses the value of each property in a normal fashion, for example:

```
String sSomeValue = “This is property value “ + prop_01 + “ and the other “ + prop_02;
```

It must be noted that if the properties file is not available at the time the JCD object is created the EJB that attempts to create it will throw an exception because the JCD’s constructor throws an exception.

Issues

This technique contravenes at least two rules set forth in the EJB 2.0 Specification, Runtime Restrictions. This makes the solution that uses this technique non-EJB compliant.

The code, as shown, is not thread-safe. So long as there are no multiple threads concurrently setting/reading property values all will be well. If thread-safe access is required appropriate synchronisation techniques will need to be employed.
Properties set this way are ‘static’ in the sense that any changes to the properties file will not be propagated to the running JCD because the properties file is read and parsed at JCD object creation time (in the constructor).

It must be noted that the ‘static’ characteristic of the property values persists across ‘shutdown’/’startup’ cycle of the collaboration when performed through the Enterprise Manager. The only way to ‘refresh’ the values is to re-deploy the project that contains the JCD or to re-bootstrap the logical host.

Synchronized Version

The following code demonstrates synchronized access to class variables and ought to be thread-safe.

```
private static final String sPropFileName = "my.properties";
private static String prop_01 = null;
private static String prop_02 = null;

private synchronized void setProp_01(String sProp_01) {
    prop_01 = sProp_01;
}
private synchronized String getProp_01() {
    return prop_01;
}
private synchronized void setProp_02(String sProp_02) {
    prop_02 = sProp_02;
}
private synchronized String getProp_02() {
    return prop_02;
}

public jcdProperties02()
        throws Exception
{
    // logger.info( "in jcdProperties02 constructor - doing properties stuff" );
    java.util.Properties props = new java.util.Properties();
    try {
        props.load( this.getClass().getClassLoader().getResourceAsStream( sPropFileName ) );
        setProp_01(props.getProperty( "prop_01" ));
        setProp_02(props.getProperty( "prop_02" ));
    } catch ( java.io.IOException ioe ) {
        String sMsg = "Properties File [" + sPropFileName + "] not found in classpath ";
        // alerter.critical( sMsg );
        throw new Exception( sMsg, ioe );
    }
}
```

Here is the code to access property values at runtime:

```
String sSomeValue = “Value “ + getProp_01() + “ and the other “ + getProp_02();
```

Continually Refresh

It may be that a Java collaboration requires a set of configuration values to be passed to it at runtime and that any changes to the configuration values be used by the JCD as soon as available. The properties are kept in a properties file somewhere in the file system. Where one keeps the properties file determines how one can access it at runtime.

If the properties file sits in some arbitrary place in the file system one must tell the JCD where to find it. One can either hardcode the path in the JCD or pass its value at runtime using a “JVM Args” property to the Integration Server.

In 5.0.4, if the properties file sits in a directory that is included in the IS classpath it can be accessed by just the file name, which needs be hardcoded, derived or passed via “JVM Args” property. Some reasonable places to put the properties file are <LogicalHostRoot>/logicalhost/logconfigs/IS_<integrationServerName> and <LogicalHostRoot>/logicalhost/stcis,

In 5.0.5 one must add the directory in which the properties file lives to the classpath explicitly by setting the ‘Startup Classpath Prepend’ or ‘Startup Classpath Append’ property of the Integration Server so that it includes this directory.

This technique causes the properties to be ‘refreshed’ from the properties file each time through the collaboration. This is as dynamic as it gets.

Setting

Create a properties file, called my.properties, in one of the recommended locations, with the following text:

```
prop_01 = This is prop_01 value
prop_02 = 123456
```

Propagating to Runtime

There is nothing to propagate to the runtime environment unless one uses the JVM Args to pass the file location to the JCD. If so then the same steps apply as in the previous discussion. If the properties file is accessed via the classpath the file already is where it needs to be.

Using in a JCD

Create a new Java Collaboration Definition, jcdProperties03, with whatever input, manipulation and output OTDs might be required.
In “Source Code Mode” add the following Java code just before the receive method:

```
private static final String sPropFileName = "my.properties";
private String prop_01 = null;
private String prop_02 = null;
private java.util.Properties props = new java.util.Properties();
```

In the ‘receive’ method, at the beginning, insert the following code:

```
// load properties each time through the collaboration
try {
    props.load( this.getClass().getClassLoader().getResourceAsStream( sPropFileName ) );
    prop_01 = props.getProperty( "prop_01" );
    prop_02 = props.getProperty( "prop_02" );
} catch ( java.io.IOException ioe ) {
    String sMsg = "Properties File [" + sPropFileName + "] not found in classpath ";
    throw new Exception( sMsg, ioe );
}
```

Here is the code to access property values some time later in the code:

```
String sSomeValue = “Value “ + prop_01 + “ and the other “ + prop_02;
```

Issues

This technique contravenes at least two rules set forth in the EJB 2.0 Specification, Runtime Restrictions. This makes the solution that uses this technique non-EJB compliant.

Each time through the JCD, the properties file is found, loaded and reinterpreted. Depending on the performance requirements and the size of the properties file this may be unacceptable from the performance stand point. If there are no changes to the properties file the work of refreshing properties each time is wasted.

Refreshed until Disable Refresh Flag

It may be that a Java collaboration requires a set of configuration values to be passed to it at runtime and that any changes to the configuration values be used by the JCD as soon as available. The properties are kept in a properties file somewhere in the file system. Where one keeps the properties file determines how one can access it at runtime.

If the properties file sits in some arbitrary place in the file system one must tell the JCD where to find it. One can either hardcode the path in the JCD or pass its value at runtime using a “JVM Args” property to the Integration Server.

In 5.0.4, if the properties file sits in a directory that is included in the IS classpath it can be accessed by just the file name, which needs be hardcoded, derived or passed via “JVM Args” property. Some reasonable places to put the properties file are <LogicalHostRoot>/logicalhost/logconfigs/IS_<integrationServerName> and <LogicalHostRoot>/logicalhost/stcis, In 5.0.5 one must add the directory in which the properties file lives to the classpath explicitly by setting the 'Startup Classpath Prepend' or 'Startup Classpath Append' property of the Integration Server so that it includes this directory.

This technique causes the properties to be 'refreshed' from the properties file each time through the collaboration until a flag is set to stop. Once stopped, the refresh process can only be re-started by re-booting the logical host or re-activating the deployment.

Setting

Create a properties file, called my.properties, in one of the recommended locations, with the following text:

```
prop_01 = This is prop_01 value
prop_02 = 123456
keep_refreshing = TRUE
```

Propagating to Runtime

There is nothing to propagate to the runtime environment unless one uses the JVM Args to pass the file location to the JCD. If so then the same steps apply as in the previous discussion. If the properties file is accessed via the classpath the file already is where it needs to be.

Using in a JCD

Create a new Java Collaboration Definition, jcdProperties03, with whatever input, manipulation and output OTDs might be required.

In "Source Code Mode" add the following Java code just before the receive method:

```
private static final String sPropFileName = "my.properties";
private static String prop_01 = null;
private static String prop_02 = null;
java.util.Properties props = new java.util.Properties();
private static boolean blKeepRefreshing = true;
public synchronized boolean isKeepRefreshing() {
    return blKeepRefreshing;
}
public synchronized void setKeepRefreshing( boolean keep_refreshing ) {
    blKeepRefreshing = keep_refreshing;
}
private synchronized void setProp_01(String sProp_01) {
    prop_01 = sProp_01;
}
private synchronized String getProp_01() {
    return prop_01;
}
private synchronized void setProp_02(String sProp_02) {
    prop_02 = sProp_02;
}
private synchronized String getProp_02() {
    return prop_02;
}
```

In the 'receive' method, at the beginning, insert the following code:

```
if (isKeepRefreshing()) {
    // load properties each time through the collaboration until told to stop
    // the keep_refreshing property, when set to anything other than
    // (case insensitive) true, t, 1 or -1 will stop refresh
    try {
        props.load( this.getClass().getClassLoader().getResourceAsStream( sPropFileName ) );
        setProp_01(props.getProperty( "prop_01" ));
        setProp_02(props.getProperty( "prop_02" ));
        String keep_refreshing = props.getProperty( "keep_refreshing" );
        if (keep_refreshing != null
        && !java.util.regex.Pattern.matches("(?i)true|1|-1|(?i)t", keep_refreshing)) {
            setKeepRefreshing( false );
        }
    } catch ( java.io.IOException ioe ) {
        String sMsg = "Properties File [" + sPropFileName + "] not found in classpath";
        throw new Exception( sMsg, ioe );
    }
}
```

Here is the code to access property values some time later in the code:

```
String sSomeValue = "Value " + getProp_01() + " and the other " + getProp_02();
```

This collaboration will refresh the properties as long as the 'keep_refreshing" property is missing or is set to TRUE, true, T, t, -1 or 1. When it gets set to any other value, next time the collaboration reloads the properties it will unset the keeprefreshing flag and will skip refresh code from then on.

This technique is good for development and experimentation as once the property values settle, the refreshing cycle can be disabled.

Issues

This technique contravenes at least two rules set forth in the EJB 2.0 Specification, Runtime Restrictions. This makes the solution that uses this technique non-EJB compliant.

Once the refresh cycle is disabled it cannot be re-enabled without re-booting the logical host or re-activating deployment.

Refresh When Changed

It may be that a Java collaboration requires a set of configuration values to be passed to it at runtime and that any changes to the configuration values be used by the JCD when indicated. The properties are kept in a properties file somewhere in the file system. Where one keeps the properties file determines how one can access it at runtime.

If the properties file sits in some arbitrary place in the file system one must tell the JCD where to find it. One can either hardcode the path in the JCD or pass its value at runtime using a "JVM Args" property to the Integration Server.

In 5.0.4, if the properties file sits in a directory that is included in the IS classpath it can be accessed by just the file name, which needs be hardcoded, derived or passed via "JVM Args" property. Some reasonable places to put the properties file are <LogicalHostRoot>/logicalhost/logconfigs/IS_<integrationServerName> and <LogicalHostRoot>/logicalhost/stcis, In 5.0.5 one must add the directory in which the properties file lives to the classpath explicitly by setting the 'Startup Classpath Prepend' or 'Startup Classpath Append' property of the Integration Server so that it includes this directory.

This technique causes the properties to be 'refreshed' from the properties file when the properties file changes.

Setting

Create a properties file, called my.properties, in one of the recommended locations, with the following text:

```
prop_01 = This is prop_01 value
prop_02 = 123456
```

Propagating to Runtime

There is nothing to propagate to the runtime environment unless one uses the JVM Args to pass the file location to the JCD. If so then the same steps apply as in the previous discussion. If the properties file is accessed via the classpath the file already is where it needs to be.

Using in a JCD

Create a new Java Collaboration Definition, jcdProperties03, with whatever input, manipulation and output OTDs might be required.

In "Source Code Mode" add the following Java code just before the receive method:

```
private static final String sPropFileName = "my.properties";
private static String prop_01 = null;
private static String prop_02 = null;
private static java.io.File flFile = null;
private static long lastModified = 0;
java.util.Properties props = new java.util.Properties();

private synchronized void setProp_01( String sProp_01 ) {
    prop_01 = sProp_01;
}
private synchronized String getProp_01() {
    return prop_01;
}
private synchronized void setProp_02( String sProp_02 ) {
    prop_02 = sProp_02;
}
private synchronized String getProp_02() {
    return prop_02;
}
private synchronized void setFile( java.io.File pflFile ) {
    flFile = pflFile;
}
private synchronized java.io.File getFile() {
    return flFile;
}
private synchronized void setLastModified( long lLastModified ) {
    lastModified = lLastModified;
}
private synchronized long getLastModified() {
    return lastModified;
}
```

In the 'receive' method, at the beginning, insert the following code:

```
ClassLoader cl = this.getClass().getClassLoader();
// first time init?
```

```
if (getLastModified() == 0) {
    java.net.URL url = cl.getResource( sPropFileName );
    setFile( new java.io.File( url.getPath() ) );
}
// has the file changed?
if (getLastModified() < getFile().lastModified()) {
    setLastModified( getFile().lastModified() );
    try {
        props.load( cl.getResourceAsStream( sPropFileName ) );
        setProp_01( props.getProperty( "prop_01" ) );
        setProp_02( props.getProperty( "prop_02" ) );
    } catch ( java.io.IOException ioe ) {
        String sMsg = "Properties File [" + sPropFileName + "] not found in classpath";
        throw new Exception( sMsg, ioe );
    }
}
// do whatever needs doing
```

Here is the code to access property values some time later in the code:

```
String sSomeValue = "Value " + getProp_01() + " and the other " + getProp_02();
```

This collaboration will refresh the properties each time it detects that the properties file was modified.

This technique can be combined with elements of the previous technique such that when the property values settle the refresh can be disabled by a property setting.

Issues

This technique contravenes at least two rules set forth in the EJB 2.0 Specification, Runtime Restrictions. This makes the solution that uses this technique non-EJB compliant.

The 'last modified time' is obtained each time through the collaboration. This imposes an overhead that may or may not be acceptable depending on the circumstances.

Conclusion

Examples provided in this paper illustrate some of the techniques that may be used to supply a Java Collaboration Definition with configuration values at runtime. From an EJB compliance standpoint there are other ways to do this which are more or less legitimate, however most of the techniques presented here contravene Runtime Restrictions set forth in the EJB 2.0 Specification. Breaking the rules may be appropriate in specific circumstances as long as the rule breaker is aware of the rules and understands and accepts the consequences.