Getting Started with Dipforge backend programming.

Table of Contents

Getting Started With Dipforge

Contained within this document are some basic tutorials which should allow you to get Dipforge up and running quickly and easily.

Installation

Requires Java JDK 1.7.

Follow these steps to install a primary Dipforge instance:

- Download Dipforge install package from sourceforge.net . File “Dipforge_Install_'latest version'.jar”
- Run Dipforge Install package. In Unix this is done via the command line by typing “java -jar Dipforge_Install_'latest version'.jar” and in Windows by double clicking on the downloaded file.
- When prompted to enter a Dipforge Id, supply a unique name, that will identify this instance, such as a host name.
- When asked to choose an installation path you can stick with the default or choose you own.
- When asked for the JDK location use the browse button to locate it on your computer, you will need to have JDK 1.5 or higher installed in order to run Dipforge.

Running Dipforge

To run Dipforge on Windows or any Unix based OS simply navigate to the bin folder within your installation using the command prompt and run either the run.bat or run.sh file based on OS. Note: run.bat is for Windows and run.sh is for any Unix based OS.

Web

Once a Dipforge instance has been started it is possible to access the embedded Tomcat Daemon. This daemon is normally bound to port 8080. Simply open a browser and point it at http://[host name]:8080. Doing this brings up the home page for Dipforge, which offers links to the Tomcat administration, the Dipforge Administration console and documentation.

Tomcat

1. Go to the URL http://[host name]:8080.
2. Click on the tomcat link.
3. Enter a user name and password that has been configured to access the management frontend. The default of admin and 112233 will work on a stock system. To change this look at the user configuration section.
4. Once the correct user name and password has been entered the Tomcat management page will appear.

Note: It is possible to deploy a war file using the tomcat management frontend, but once the Dipforge instance is restarted these applications will be lost. Instead use the standard deployment process of copying the war file to the Dipforge deploy directory. Dipforge will take care of the rest.

Dipforge Administration

The console supplies the ability to look at interact with all the deployed MX Beans, Daemons and Web Services.

1. Go to the URL http://[host name]:8080.
2. Click on the “Dipforge Admin” link.
3. Enter a user name and password that has been configured to access the management frontend. The default of admin and 112233 will work on a stock system. To change this look at the user configuration section.
4. Once the correct user name and password has been entered the Dipforge Admin console will appear.

Deployment

Dipforge provides auto deployment for daemons and web applications. It expects all daemons and web applications to be self contained in a jar or war file. This means that if the code replies on another library that is not provided by Dipforge, that it must be included in the war or jar file.

Daemon Jar

The daemon jar file must contain all the compiled classes for the daemon plus all the libraries that are not provided by Dipforge. The libraries must be included in the base of the jar and not in any sub directory. For an example look at the sample projects provided with the Dipforge distribution.

Web Application War

Web applications must be deployed in war files. These war files do not differ in any way from traditional Application server war files, as Tomcat is used to run the Web Applications.

Deployment Processes

Deployment:

To deploy a daemon or web application copy the daemon jar or web war to the Dipforge deploy directory. This can be done before starting Dipforge or while it is running.

Un-deployment:

To un-deploy a daemon or web application simply remove the jar or war from the Dipforge deploy directory. This can be done while Dipforge is running or after it is stopped.

Re-deployment:

To re-deploy a daemon or web application simply replace the jar or war file in the deploy directory. If Dipforge is running the existing instance will be stopped and the new instance started.

Web Applications

Dipforge supports standard web applications through the use of Tomcat. This means that it is possible to write a web based application that interacts seamlessly with any Dipforge deployed daemon. This is because any deployed application has access to Dipforges JNDI and any exposed Dipforge resources; making it possible to resolve any daemon or any JNDI bound resource, and providing central configuration, security, transactions and more.

Anything accessible to a daemon is accessible to a Web Application.

JNDI

Dipforge exposes its own JNDI context to any servlets or JSP pages. This makes it possible to resolve data sources, the user transaction manager and Dipforge daemons from an Web Application. There is no need setup a context to do it on deployment of an application, as is the traditional mechanism with Tomcat.

Data Sources

The following example gets a reference to the standard HsqlDB data source that comes with every Dipforge instance.

```
// intanciate a context
Context context = new InitialContext();
java.sql.DataSource ds = (java.sql.DataSource)context.lookup(“java:comp/env/jdbc/hsqldb”);
```

User Transaction Manager

If a web application wishes to manage a transaction than it can get a reference to the transaction manager in Dipforge as follows.

```
// Instanciate a context
Context context = new IntialContext();
UserTransaction ut = (UserTransaction)context.lookup(“java:comp/UserTransaction”);
```

Dipforge Daemon

It is possible to get a connection to any Dipforge daemon using JNDI. The process is the same as a data source lookup except it returns an RMI reference that can than be narrowed and than called appropriately. The code below uses the MessageService as an example.

```
// Instanciate a context
Context context = new IntialContext();
Object ref = context.lookup(“message/MessageService”);
MessageService messageService = (MessageService)PortableRemoteObject.narrow(ref,
   MessageService.class);
```

Configuration

Dipforge implements centralized configuration, and makes this available to any Web Applications. This means that when implementing a web application, there is now the possibility of storing configuration information in the web war and instance configuration in Dipforges configuration.

For more information on using the Configuration objects look at the section titled Configuration.

Security

Dipforge implements container based security. That means that a user accessing a container must have the appropriate rights in order to perform an action. This same security is carried over into the Tomcat Daemon. What this means is that for a web application to access a Dipforge daemon, it must first authenticate the user. Once a user has an authenticated session that user will than be able to access daemons and makes call onto them. This means that Dipforge implements Single Sign On (SSO). This also means that an application will not have to implement its own security module as it will be provided for by Dipforge.

Dipforge currently supports the following types of authentication (More options will be available in time):

1. **Form Based Authentication**

   Form based authentication relies on a form requesting authentication information from the user. In Dipforge this

is currently limited to Username and Password.
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security5.html

2. **Basic Authentication**
   Like Form based authentication basic authentication requires the user to supply identification information, but unlike form based authentication this information is gathered by the users browser and posted when required.

Note: For more information on setting up security in a J2EE web based application look at the following tutorial

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security5.html.

Documentation

The Dipforge distribution comes with a example web application. This can be found in the samples directory of the distribution. This sample is there to give people an under standing of how to interact with Dipforge using a web application, it is not there to train users on JSP, and Servlets. More on that can be found online at: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

Note: Dipforge does not implement EJB's. This means that traditional Enterprise beans are not available but the same functionality and more, is available via the daemon container framework implemented by Dipforge.

Daemons

A Dipforge daemon is deployed in a self contained jar file. This means that if there are any libraries that the daemon is dependant on, that are not provided by Dipforge, they must be included in the deployment jar.

This jar file also contains the deployment information for the daemons it contains. This information is stored in a Dipforge.xml file within the /META-INF directory of the jar.

Jar File Layout

The jar file must be laid out in the following manor:

```
/
/META-INF
/META-INF/Dipforge.xml
/Example.jar
/com/rift/example.class
```

In the above example the jar consisted of a META-INF directory that contained the “Dipforge.xml”, an “Example.jar” or library and a compiled class. This is the standard layout for Dipforge daemon files

Dipforge.xml

This file defines the deployment information for a daemon. It describes the interfaces that must be exposed by RMI, and Locally, JNDI bindings, the web services, transaction information, caching information, user information, security information, threading etc.

The file contents are broken down as follows:

```
<!-- The base element for the file-->
<Dipforge version=”1” name=”This is the name of the daemon”>
  <description>The description of the daemon</description>
  <!-- The basic daemons that will be started and bound to JNDI -->
  <beans>
    <!-- a bean definition -->
    <bean>
      <interface>The interface that will be exposed for this bean</interface>
      <class>The class that implements the interface</class>
      <bindName>The jndi name to use “example/test1”</bindName>
      <role>The role that a user must be able to access in order to access the container</role>
      <username>The name of the user that this daemon will run as. This is optional and only required
      if the class implements the BeanRunnable interface.</username>
      <classes>The classes such as factory objects that must get containers</classes>
      <cache_by_key>true or false, set if factory objects are going to be used with unique key
      identifiers</cache_by_key>
      <cache_timeout>The number of milliseconds to cache an entry</cache_timeout>
      <transaction>True if container based transaction are going to be used.</transaction>
      <!-- If this bean is going to have other threads they are configured here -->
      <thread>
        <class>The name of the class implementing the basic thread </class>
        <username>The name of the user this thread will run as</username>
        <number>The size of the thread pool for this thread</number>
      </thread>
    </bean>
  </beans>
  <!-- The jmx daemons that will be bound to the JMX manager and JNDI -->
  <jmxbeans>
    <!-- a bean definition -->
    <bean>
      <interface>The interface that will be exposed for this bean</interface>
      <class>The class that implements the interface</class>
      <bindName>The jndi name to use “example/test1”</bindName>
      <objectName>The name to bind the object to the JMX manager with
      “com.test:type=JMXBean1”</objectName>
      <role>The role that a user must be able to access in order to access the container</role>
      <username>The name of the user that this daemon will run as. This is optional and only required
      if the class implements the BeanRunnable interface.</username>
```

```
        <classes>The classes such as factory objects that must get containers</classes>
        <cache_by_key>true or false, set if factory objects are going to be used with unique key
        identifiers</cache_by_key>
        <cache_timeout>The number of milliseconds to cache an entry</cache_timeout>
        <transaction>True if container based transaction are going to be used.</transaction>
        <!-- If this bean is going to have other threads they are configured here -->
        <thread>
          <class>The name of the class implementing the basic thread </class>
          <username>The name of the user this thread will run as</username>
          <number>The size of the thread pool for this thread</number>
        </thread>
      </bean>
    </jmxbeans>
    <!-- The web services to expose -->
    <webservices>
      <webservice>
        <path>The path that the web service will be bound to</path>
        <class>The class implementing the web service call</class>
        <wsdl>The path to the wsdl file resource within the deployed jar.</wsdl>
        <role>The role that the caller must have access to</role>
         <transaction>True if container based transactions are going to be used.</transaction>
      </webservice>
    </webservices>
</Dipforge>
```

Daemon Naming

Dipforge auto deploys everything it finds in its deploy directory. This means that deploying something new is simply a matter of copying the jar, containing the new daemon, into the deploy directory and waiting. Dipforge deploys files in order based on the name of the file, they will be un-deployed on shutdown in reverse order. This means that if one daemon must start before another its name should order it before the other.

To make sure daemons load in the appropriate order all jars must be prefixed with a four digit number and a dash. Example:

0010-{Name}

This is very similar to the process ordering found on a Unix box.

Ranges

0000 – 0100 Critical processes such as databases, timers, message services and service brokers.

0100 – 1000 Business Logic related daemons.

1000 - & Anything.

A very basic Dipforge Daemon

To write a Daemon, requires the writing of a Dipforge Bean. This bean will get bound to JNDI and supply access to the daemon. The following tutorial will cover all the steps required to create an absolute bare bones instance, any further expansion is very easy.

Start by creating an empty project in your favorite IDE. In the source root directory create a “META-INF” folder, this is where we will be putting the “Dipforge.xml” file that will define the bean.

The basic layout of the “Dipforge.xml” file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<Dipforge version="1" name="DipforgeTest">
   <beans>
      <bean>
         <bindName><!-- Bean name --></bindName>
         <role><!-- role --></role>
         <username><!-- username --></username>
      </bean>
   </beans>
</Dipforge>
```

There are an additional 2 parameters required to create the most basic Dipforge bean possible, these parameters are <interface><!-- bean interface --></interface> and <class><!-- interface implementation --></class>. You will need to create a package within your source root folder in which the bean interface and interface implementation will be created. Your “Dipforge.xml” file will then look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<Dipforge version="1" name="DipforgeTest">
   <beans>
      <bean>
         <bindName><!-- Bean name --></bindName>
         <role><!-- role --></role>
         <username><!-- username --></username>
         <interface><!-- bean interface --></interface>
         <class><!-- interface implementation --></class>
      </bean>
   </beans>
</Dipforge>
```

If you wish the bean to be globally available than it must implement java.rmi.Remote and throw java.rmi.RemoteException from all of its interface methods. If you want it to be a local interface than this is not a requirement.:

Example:

```
package com.test;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BeanInterface extends Remote {
   public void exampleMethod() throws RemoteException;
}
```

The interface implementation is where the logic of the bean is located. An example of this follows:

Example:

```
package com.test;

import java.rmi.RemoteException;

public class BeanImpl implements BeanInterface {

   public BeanImpl() {
```

```
    }

    public void exampleMethod() throws RemoteException {
        /** Do something. */
    }
}
```

Once you have created both of these files and put them into their respective folders one simply has to build the project and copy the generated .jar file into the Dipforge deploy folder and Dipforge will take care of the rest. A sample daemon can be found within the Dipforge installation under samples. Note: This is a Netbeans project.

Libraries and Dependencies

For a project that is dependent on other libraries such as database, daemons, or utils, one simply has to include these jars in the final deployment jar. This results in a single deployment archive reducing the complexity of the deployment and means that no extra modifications need to be made to a Dipforge instance in order to run another daemon.

Note: The jars that must not be included in the deployment archive are those that get stored in Dipforge's own lib directory.

Creating an RMI connection to Dipforge

The Dipforge RMI connection supports both local and remote connections and is easy to implement.

You will need to include the DipforgeClient.jar file within your class path.

The required Java imports are:

```
import java.util.Hashtablel
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
```

The code to create the connection is as follows:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.rift.coad.client.naming.DipforgeInitialContextFactory");
env.put(Context.PROVIDER_URL,/** localhost:2000 String */);
env.put("com.rift.coad.username",/** Username String */);
env.put("com.rift.coad.password",/** Password String */);
Context ctx = new InitialContext(env);

Object obj = ctx.lookup("java:network/env//** Dipforge server ID *///** Bean name */");
/** Bean Interface e.g. com.rift.coad.BeanInterface */ beanInterface =
      (/** Bean Interface */)
      PortableRemoteObject.narrow(obj,
      /** Bean Interface.class */);
```

Once you have inserted the code and made the appropriate changes, by replacing the comments with the necessary details, you will have your first Dipforge RMI connection.

Security Manager

This functionality relies on the ability to dynamically download the generated stub code. To do this the security manager must be enabled. If the code is running in a standalone environment, and this does not work, there is a good chance the security manager is not enable.

To enable the security manager add the following to the command line of the java environment.

Windows

```
set JAVA_OPTS=%JAVA_OPTS% -Djava.security.policy==C:\server\policy\file
set JAVA_OPTS=%JAVA_OPTS% -Djava.security.manager
```

Unix

```
JAVA_OPTS=$JAVA_OPTS -Djava.security.policy==/server/policy/file
JAVA_OPTS=$JAVA_OPTS -Djava.security.manager
```

The server policy file contains the policies used by the java environment to see if objects have the rights to access one another. Use the following if you are not concerned about this and just want to get it running.

```
grant {
   permission java.security.AllPermission;
};
```

For more information on the JDK security manager visit: http://java.sun.com/j2se/1.4.2/docs/guide/security/.

JacORB

Because Dipforge implements RMI/IIOP using JacORB, all clients making an RMI connection to it, will have to use it as well. This simply means linking with the libraries, the DipforgeClient library will take care of the rest.

JacORB and its library dependancies are provided as part of the clientlib directory. They are as follows:

- avalon-framework-4.1.5.jar
- jacorb.jar
- logkit-1.2.jar

Bean Factory Objects

Dipforge supports two types of factory objects. Unique key based factory objects and time based factory objects. Key based factory objects work by caching an object identified by a unique key, for a set period of time. Time based factory objects cache an object purely on time.

A key based factory object would be used to manipulate an identifiable unit such as a mailbox, account or file; while a time based factory object could be used as a session based connection such as a database connection.

The cache period time in both cases is the period of time without any activity that the object must be cached for before being removed. Activity on the object will length the time the object are in memory.

A basic sample of a bean factory can be found within the Dipforge installation under samples. Note: This is a Netbeans project.

Entity Object

To implement an entity object factory pattern there must some unique identifying information for the object. If this is not a single value than a composite key object needs to be constructed. This object must be serializable and implement the equals and hashId methods.

Factory Manager pattern method naming and standards

The management interface must follow the following rules in order to act as a factory object.

- **create:** The method that creates entries in the factory must contain either create or add in the method name, and it do not have to take the primary key as a value. It must return a reference to this object to cache.
- **find:** Methods that return a reference to the factory object must take the primary key and contain either get or find in the method name. They must return a reference to the factory object.
- **Remove:** The method that removes a entry must take the primary key as the only parameter and contain remove or delete in the method name.

Factory Object implementation

The factory object must be implemented as follows.

- It must implement the ResourceIndex interface. This contains a simple method that returns the unique key for this object. This is used by the create method to identify the object in the cache.
- If the entity would like to know when it is being removed from memory in order to serialize information, it must implement the Resource interface. This has methods to name the resource and release it when it is removed from memory.

Dipforge.xml

The Dipforge file in the META-INF directory needs to be modified to support the factory interface. The following parameters must be added to the bean implementing the the factory methods.

```
<cache_by_key>true</cache_by_key>
<cache_timeout><!-- a cache time in milli seconds.--></cache_timeout>
<!-- List the classes that get returned as factory objects -->
<classes><!--a name of a class that will get return as a factory object --></classes>
<classes><!--a name of a class that will get return as a factory object --></classes>
```

If the cache_timeout is set to zero it will remain in memory until the daemon is un-deployed or until the instance is stopped.

Time Object

The time object is very simple to implement. The factory management interface must simply return a reference to the object that must be timed. This will generally be a new object each time. The container will than wait until the object has not been modified or touched for a given period of time before removing it.

The management method names should not use, add, create, get, find, remove or delete to reference the object, use connect instead.

Dipforge.xml

The Dipforge file in the META-INF directory needs to be modified to support the factory interface. The following parameters must be added to the bean implementing the the factory methods.

```
<cache_timeout><!-- a cache time in milli seconds.--></cache_timeout>
<!-- List the classes that get returned as factory objects -->
<classes><!--a name of a class that will get return as a factory object --></classes>
<classes><!--a name of a class that will get return as a factory object --></classes>
```

If the cache_timeout is set to zero it will remain in memory until the daemon is un-deployed or until the instance is stopped.

Web Services

Dipforge supports web services as a means to communicate with a daemon. These web services are exposed via a web server which by default runs on port 8085(This is configurable). To implement a web service follow the following rules:

- There must be a one to one mapping between the web service interface and the WSDL definition, Ie, the WSDL cannot describe more than one end point.
- The WSDL and implementation class must be specified in the Dipforge.xml file for the daemon.
- The interface for a web service must inherit from java.rmi.Remote.

Dipforge.xml

To add a web service to the Dipforge.xml file add the following section:

```
<webservices>
   <webservice>
      <path><!-- path on web server, the end point it will end up being http://server/path--></path>
      <class><!-- the class implementing the WSDL interface --></class>
      <wsdl><!-- the path to the WSDL file --></wsdl>
      <role><!-- the role that is required to access this web service --></role>
      <classes><!-- add an extra class name not used directly by the java web service interface
--></classes>
   </webservice>
</webservices>
```

WSDL

Here follows a WSDL example, please note, that the end point address is over ridden when loaded in to a Dipforge instance.

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions name="WebService" targetNamespace="urn:WebService/wsdl"
xmlns:tns="urn:WebService/wsdl" xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types/>
  <message name="WebServiceSEI_helloWorld">
    <part name="String_1" type="xsd:string"/></message>
  <message name="WebServiceSEI_helloWorldResponse">
    <part name="result" type="xsd:string"/></message>
  <portType name="WebServicePortType">
    <operation name="helloWorld" parameterOrder="String_1">
      <input message="tns:WebServiceSEI_helloWorld"/>
      <output message="tns:WebServiceSEI_helloWorldResponse"/></operation></portType>
  <binding name="WebServiceSEIBinding" type="tns:WebServicePortType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
    <operation name="helloWorld">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal" namespace="urn:WebService/wsdl"/></input>
      <output>
        <soap:body use="literal" namespace="urn:WebService/wsdl"/></output></operation></binding>
  <service name="WebService">
    <port name="WebServiceSEIPort" binding="tns:WebServiceSEIBinding">
      <soap:address location="http://localhost:8085/WebServiceTest"/></port></service></definitions>
```

Using Axis 1.4 to generate WSDL

It is possible and recommended to build the WSDL files from the interface definition rather than hand writing them. This can be done simple by using Ant and Axis 1.4. The example below is an extract from an ant build file.

```
  <target name="-pre-jar">
    <path id="wsdl.classpath">
```

```
      <pathelement location="axis jar location"/>
      <pathelement location="axis ant jar location"/>
      <pathelement location="apache commons logging jar"/>
      <pathelement location="log 4j jar location"/>
      <pathelement location="apache commons discovery jar"/>
      <pathelement location="axis rpc jar"/>
      <pathelement location="axis wsdl jar"/>
      <pathelement location="a path to the directory in which the compiled interface can be found"/>
    </path>

    <taskdef resource="axis-tasks.properties" classpathref="wsdl.classpath"/>

    <axis-java2wsdl
      classname="com.Example.WebService.Interface"
      location="http://localhost:8080/Example/WebService"
      namespace="com.Example"
      output="${build.classes.dir}/com/Example/WebService.wsdl"
      extraclasses="”the path to a class file not used directly by the interface”>
    </axis-java2wsdl>
  </target>
```

For more information on this visit http://ws.apache.org/axis and for java2wsdl information visit http://ws.apache.org/axis/java/ant/axis-java2wsdl.html.

Web Service Interface

The web service interface must inherit from java.rmi.Remote and implement the methods defined in the WSDL file.

```
public interface WebServiceSEI extends java.rmi.Remote {
    /**
     * This method will be called to test the web service end point interface.
     *
     * @param msg The string containing the message for the server.
     * @return The containing the message from the server.
     */
    public String helloWorld(String msg);
}
```

Web Service Implementation

```
public class WebService implements WebServiceSEI {

    /**
     * Creates a new instance of WebService
     */
    public WebService() {

    }


    /**
     * This method will be called to test the web service end point interface.
     *
     * @param msg The string containing the message for the server.
     * @return The containing the message from the server.
     */
    public String helloWorld(String msg) {
        System.out.println(msg);
        return “Test message”;
    }
}
```

Threading

Dipforge supports the ability to run user bound threads. These are threads that are know to Dipforge and have a user associated with them. If no user is associated with a thread, that thread will not be able to access another container. Note: Threads that are not known to Dipforge will not be notified of a shutdown but simply destroyed leaving no room to clean up. There are ways around this involving thread management patterns that will be described later.

Threads in Dipforge do not process using the standard run method they instead supply a **process** method that works in conjunction with a **terminate** method. The **process** method is called to perform the thread processing, the **terminate** to notify the thread that it has been requested to stop processing.

Note: If a thread does not respond to the **terminate** method with in a configured time, the thread stop method is called, resulting in thread death and no time to clean up.

A sample daemon interface based thread can be found within the Dipforge installation under samples. Note: This is a Netbeans project.

Implementing Threads

There are two main ways of implementing threads within a Dipforge daemon. The first is by turning a daemon interface into a thread, the second by implementing a stand alone thread that can be started as a single entity or a pool.

Daemon Interfaces

A daemon interface can be turned into a thread my implementing the BeanRunnable interfaces on the final implementation class. This requires the implementation of the **process** and **terminate** methods.

Example:

```
import com.rift.coad.lib.bean.BeanRunnable;

class DaemonInterfaceImpl implements DaemonInterface,  BeanRunnable {

    /**
     * The process method
     */
    public void process() {
        /* perform the processing until terminate is called */
    }

    /**
     * The terminate method
     */
    public void terminate() {
        /* notify thread that it has been terminated and must shut down */
    }
};
```

Standalone Threads

To implement a standalone thread extend the BeanThread and implement the process and terminate methods.

Example:

```
import com.rift.coad.lib.bean.BeanThread;

class StandaloneThread extends BeanThread {

    /**
     * Must implement a constructor that throws Exception.
     */
    public StandaloneThread() throws Exception {
        /* setup the environment */
    }


    /**
     * The process method
     */
    public void process() {
```

```
      /* perform the processing until terminate is called */
   }


   /**
    * The terminate method
    */
   public void terminate() {
      /* notify thread that it has been terminated and must shut down */
   }
};
```

Dipforge.xml

The daemon interface thread and the standalone threads must run as a user, and in the case of the standalone thread the number of threads must be specified. This is done through the Dipforge.xml file.

```
<beans>
   <bean>
      <interface><!-- path to the interface --></interface>
      <class><!-- path to the implementation class --></class>
      <classes><!-- Factory object path --></classes>
      <cache_by_key><!- if true cache the key defined methods --></cache_by_key>
      <cache_timeout><!- The time out in milli seconds --></cache_timeout>
      <bindName><!-- Name of bean to be bound to JNDI --></bindName>
      <role><!-- The role describing the access permissions --></role>
      <username><!-- The user the daemon interface thread will start as--></username>
      <thread>
         <class><!-- Path to thread implementation thread--></class>
         <username><!-- The name of the user--></username>
         <number><!-- The number of threads to start--></number>
      </thread>
   </bean>
</beans>
```

Starting threads on demand

Dipforge does support the ability to start threads on demand. This functionality is supplied by the DipforgeThread object. Simply extend it and implement both the process and terminate methods appropriately. Than call start on the the thread when required. The start method supplied by the DipforgeThread takes a user-name. This is the name of the user that the thread must run as. In order to call start on the thread the current thread must be granted the write to that role.

DipforgeThread

```
// imports
import com.rift.coad.lib.thread.DipforgeThread;

/**
 * The implementation of the on demand thread.
 */
public class OnDemandThread extends DipforgeThread {
   /**
    * The constructor must throw an Exception
    */
   public OnDemandThread() throws Exception {
   }


   /**
    * The processing method
    */
   public void process() throws Exception {
      /* thread processing */
   }


   /**
```

```
     * This method will be implemented by child objects to terminate the
     * processing of this thread.
     */
    public synchronized void terminate() {
        /* terminate the thread processing */
    }
}

// starting the thread
OnDemandThread thread = new OnDemandThread();
thread.start(/* the name of the user */);
```

Security

The thread starting the Dipforge thread instance must have the rights to do so. This is controlled by an entry in the configuration file.

```
<object name="com.rift.coad.lib.thread.ThreadGroupManager">
  <entry key="role" type="string">
     <!-- the name of the role that the thread starting the on demand thread must have access to -->
  </entry>
</object>
```

Thread Management Pattern

There are various circumstances where using only Dipforge user based threads is not going to work. Such as embedding Jakata or another daemon that has its own threading. Under these circumstances the developer will have to use the thread management pattern.

This pattern works by implementing a single user bound Dipforge thread. (This can be anyone of the available Coaduantion threads.) Its role will be to start the processing of the embedded daemons threads and when required terminate the daemon.

Security

The embedded daemons threads will not be able to access another container, through there are ways around this. Using the interceptor framework supplied by Dipforge will make it possible to assign a none Dipforge based thread an appropriate user. This is normally done by supplying user authentication data to the framework, such as user name and password, thus enabling the framework to assign the current thread an active session id, so that it can be validated against other containers.

JNDI

Dipforge has its own implementation of JNDI. This is broken into two URL patterns **java:comp** and **java:network**. The **java:comp** base falls within the process and the **java:network** is scoped within the Dipforge cluster. This means anything added to **java:comp** will only be accessible to things running within that process such as database source, user transaction managers and local interfaces. Where as the **java:network** base is accessible to to any daemon within a Dipforge cluster.

java:comp

The **java:comp** environment is divided into a global process environment, this covers data source added in the configuration file and the user transaction manager. There is a second level that is scoped within a daemon. Any entry added here such as a local interface or a data source added by code is only accessible to the code within that daemon.

Example:

```
// a data source that could be available to that processing environment
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/test");

// a local interface only available within a daemon
com.test.BeanInterface beanInterface = (com.test.BeanInterface)
        context.lookup("java:comp/env/bean/testbean");
```

java:network

The **java:network** environment is defined by the cluster. The basic url is as follows **java:network/env/instanceid/interface/name**.

Example:

If a entry in a cluster is defined by the name Test than the url to get to a interface named TestingInter would be as follows.

```
java:network/env/Test/TestingInter

TestingInter obj = (TestingInter)
        PortableRemoteObject.narrow(context.lookup("java:network/env/Test/TestingInter"),
        TestingInter.class);
```

Lookups do not have to supply the entire URL when performing a lookup on a network URL, only the relative path is required if the entry is found within the same process. This is supported as local interfaces are not accessible to anything outside of the daemon.

Example:

```
TestingInter obj = (TestingInter)
        PortableRemoteObject.narrow(context.lookup("TestingInter"),TestingInter.class);
```

Using the Timer Daemon

The timer daemon supplies the ability to register an event for a specific time. When this time is reached the timer daemon will make a call onto the service implementing the timer event handler, identified by the JNDI name for that event. It passes to the event handler the id of the event identified by a serializable object.

Registering an Event

To register an event with the timer first narrow, and than register the event as follows:

```
com.rift.coad.daemon.timer.Timer timer =
        (com.rift.coad.daemon.timer.Timer)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.timer.Timer.class);

timer.register(“jndi url of target”,10,12,13
        ,24,(Serializable)"id for event, eg end of trading",true);
```

Handling an Event

To handle an event, a daemon must simply implement the event handler interface.

```
// imports
import com.rift.coad.daemon.timer.TimerEventHandler;
import com.rift.coad.daemon.timer.TimerException;

public class /* daemon name */ implements TimerEventHandler {
    /**
     * Implementation of the processEvent method.
     *
     * @param event Identifying object.
     */
    public void processEvent(Serializable event) throws RemoteException, TimerException {
        .
        .
    }

}
```

API

Jdoc for the timer daemon is available on line via the web site and is supplied with Coaduntion install.

Using the Command Line Tool

The command line tool comes with Dipforge and both the timer.bat and timer.sh files for running it are found in the bin folder of Dipforge. The Command Line Tool allows a user to register an event, list all the events and delete a specific event via the command line.

Registering an Event

In order to register an event through the command line you need to supply several fields. These required fields are as followed.

-r This is called to register an event.

-m This is to be followed by the month and can be -1 if it is to recur once monthly.

-da This is to be followed by the day of the month and can be -1 if it is to recur on a daily basis.

-ho This is to be followed by the hour and can be -1 if it is to recur on a hourly basis.

-mi This is to be followed by the minute and can be -1 if it is to recur every minute.

-j This is to be followed by the JNDI for the daemon that is to be called.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

List Events

In order to list all stored events supply the following fields:

-l This is called to list all the events.

-j This is to be followed by the JNDI for the daemon that is to be called.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

Delete Event

In order to delete an event supply the following fields:

-d This is called when you want to delete an event.

-id This is to be followed by the ID of the event that you wish to delete.

-j This is to be followed by the JNDI for the daemon that is to be called.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

Using the Service Broker Daemon

The Service Broker Daemon stores the JNDI for daemon's and then associates that JNDI with a list of services that the daemon can perform. A single JNDI or multiple JNDI's can then be retrieved from the database using a list of appropriate services.

Registering a Service

To register a service with the Service Broker first narrow, and then register the event as follows:

```
com.rift.coad.daemon.servicebroker.ServiceBroker beanInterface =
        (com.rift.coad.daemon.servicebroker.ServiceBroker)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.servicebroker.ServiceBroker.class);

beanInterface.registerService("JNDI",serviceList);
```

Retrieving a Service

To retrieve a service from the Service Broker first narrow, and then retrieve the service as follows:

```
com.rift.coad.daemon.servicebroker.ServiceBroker beanInterface =
        (com.rift.coad.daemon.servicebroker.ServiceBroker)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.servicebroker.ServiceBroker.class);

String JNDI = beanInterface.getServiceProvider(serviceList);
```

Retrieving multiple Services

To retrieve multiple services from the Service Broker first narrow, and then retrieve the services as follows:

```
com.rift.coad.daemon.servicebroker.ServiceBroker beanInterface =
        (com.rift.coad.daemon.servicebroker.ServiceBroker)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.servicebroker.ServiceBroker.class);

List JNDI = beanInterface.getServiceProviders(serviceList);
```

Removing a Service

To remove a service from the Service Broker first narrow, and then remove the service as follows:

```
com.rift.coad.daemon.servicebroker.ServiceBroker beanInterface =
        (com.rift.coad.daemon.servicebroker.ServiceBroker)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.servicebroker.ServiceBroker.class);

beanInterface.removeServiceProviders("JNDI",serviceList);
```

API

Jdoc's for the Service Broker Daemon are available from the web site and are supplied with the Coaduntion install.

Using the Command Line Tool

The command line tool comes with Dipforge and both the servicebroker.bat and servicebroker.sh files for running it are found in the bin folder of Dipforge. The command line tool gives users the ability to register services, retrieve service JNDI's and delete services.

Registering a Service

In order to register a service supply the following fields:

-r This is called when you want to register a new service.

-S This is to be followed by the Service that the Daemon is to perform and there can be multiple in a single statement.

-j This is to be followed by the JNDI of the Daemon that performs the service.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

Retrieve Service

In order to retrieve a service supply the following fields:

-o This is to be called when you want to retrieve a single JNDI for a service or services.

-m This is to be called when you want to retrieve multiple JNDIs for a service or services.

-S This is to be followed by the Service that the Daemon is to perform and there can be multiple in a single statement.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

Deleting a Service

In order to delete a service supply the following fields:

-d This is to be called when you want to delete a service.

-S This is to be followed by the Service that the Daemon is to perform and there can be multiple in a single statement.

-j This is to be followed by the JNDI of the Daemon that performs the service.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

Using the Jython Daemon

The Jython Daemon is an implementation of Jython within Dipforge, this gives users the ability to run Python scripts within Dipforge. The Jython Daemon also includes Jython Timer which is an implementation of a TimerEventHandler which gives the user the ability to run scripts using the Timer Daemon.

Running a script

There are 2 methods that allow you to accomplish this, they are both called “runScript”. The difference however is that the once instance of runScript simply runs the script and returns a value from the script in whichever format suits you. The second instance will do this as well however it will also give you the ability to set values within the script itself. Below are 2 examples of a call to both of the above mentioned methods. Note: As with other Daemons we must first perform a narrow.

Example 1:

```
com.rift.coad.daemon.jython.JythonDaemon beanInterface =
        (com.rift.coad.daemon.jython.JythonDaemon)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.jython.JythonDaemon.class);

returnedValue = beanInterface.runScript(“ExampleScript”,”varA”,Class.forName(“java.lang.Integer”));
```

Example 2:

```
com.rift.coad.daemon.jython.JythonDaemon beanInterface =
        (com.rift.coad.daemon.jython.JythonDaemon)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.jython.JythonDaemon.class);

returnedValue = beanInterface.runScript(name,returnValue,Class.forName(“java.lang.Integer”),
        arguments);
```

Registering a script

There is a method that gives a user the ability to remotely upload scripts to the Dipforge instance, this method is “registerScript”. As with calling other Daemons a narrow must be performed first.

```
com.rift.coad.daemon.jython.JythonDaemon beanInterface =
        (com.rift.coad.daemon.jython.JythonDaemon)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.jython.JythonDaemon.class);

beanInterface.registerScript(“String containing the contents of a script file”,”The name that the script is to be called”);
```

Using the Command Line Tool

The command line tool comes with Dipforge and both the jython.bat and jython.sh files for running it are found in the bin folder of Dipforge. The command line tool gives users the ability to register scripts and run them.

Registering Scripts

In order to register a script supply the following fields:

-r This is called when you need to register a script.

-l This is to be followed by the full path of the local script.

-n This is to be followed by the name of the script. Note: in windows this is to include the file extension “.py”

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

-U Followed by the user-name to authenticate the call with.

-P Followed by the password to authenticate this call with.

Running Scripts

In order to run a script supply the following fields:

-e This is called when a user wishes to execute a script, it is followed by the name of the script to execute.

-p This is to be followed by the full path to the properties file which contains properties for the script. This field is optional.

-j This is to be followed by a java class and will determine the type that is passed back by the daemon after running the script. e.g “java.lang.Integer”

-v This is to be followed by the name of the variable who's end value is to be returned by the daemon after running the script.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

-U Followed by the user-name to authenticate the call with.

-P Followed by the password to authenticate this call with.

Using the Deployment Daemon

The Deployment Daemon gives a user the ability to remotely deploy and upload files to a Dipforge server.

Deploying a Daemon

This is done by call the “daemonDeployer” method, this will then upload a Daemon to the deployment folder within the Dipforge instance. This like all other daemon call first requires that the user perform a narrow.

```
DeploymentDaemon beanInterface =
        (DeploymentDaemon)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.deployment.DeploymentDaemon.class);

beanInterface.daemonDeployer(“byte[] containing the contents of a daemon”,
        ”example.jar”);
```

Uploading a File

This is done by calling the “copyFile” method, this method will copy any file to any folder existing folder on the Dipforge instance. This like all other daemon call first requires that the user perform a narrow.

```
DeploymentDaemon beanInterface =
        (DeploymentDaemon)
        PortableRemoteObject.narrow(obj,
        com.rift.coad.daemon.deployment.DeploymentDaemon.class);

beanInterface.copyFile(“byte[] containing the contents of a daemon”,”example.txt”,”C:\test\”);
```

Using the Command Line Tool

The command line tool comes with Dipforge and both the deploy.bat and deploy.sh files for running it are found in the bin folder of Dipforge. The command line tool gives users the ability to deploy a daemon and a file.

Uploading a Daemon

In order to deploy a daemon supply the following fields:

-d This is called when you want to deploy a daemon.

-l This is the full location of the local Daemon that is to be deployed.

-n This is the name that the Daemon is to be called. Note on a windows system this is to include the extension “.jar”

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

Uploading a file

In order to upload a file supply the following fields:

-f This is to be called when you want to upload a file to the Dipforge server.

-l This is the full location of the local Daemon that is to be deployed.

-n This is the name that the Daemon is to be called. Note on a windows system this is to include the extension “.jar”

-p This it the full path to which the file is to be copied.

-s This is to be followed by the host name and port e.g. Test:2000.

-u This is to be followed by the JNDI for the Timer Daemon.

Using the Message Service

The Dipforge message service allows for asynchronous distributed processing with guaranteed delivery.

Asynchronous RPC Messages

Server side implementation

In order to use the the message service one simply has to make a standard Dipforge bean on the server side.

Below is an example Dipforge bean interface:

```
package test.server;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RPCServerTest extends Remote {

   public int testMethod(String msg) throws RemoteException;

}
```

Below is an example Implementation of the bean interface:

```
package test.server;

import java.util.Date;
import java.rmi.Remote;
import java.rmi.RemoteException;

public class RPCServerTestImpl implements RPCServerTest {

   public RPCServerTestImpl() {
   }

   public int testMethod(String msg) throws RemoteException {
      return msg.length();
   }

}
```

Client side implementation

However on the client side it is necessary to create an asynchronous interface. This will contain all the methods from the server side bean effectively mirroring it but it must always return a String object. It is also necessary for the client side object to extend “AsyncCallbackHandler” and implement the methods “onSuccess(String messageId, String correllationId, Object result) throws RemoteException” and “onFailure(String messageId, String correllationId, Throwable caught) throws RemoteException”. The client side implementation also requires that both “MessageServiceClient.jar” and “DipforgeUtil.jar” be linked into the project.

Below is an example asynchronous interface for the above bean:

```
package test.client;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RPCServerTestAsync extends Remote {

   public String testMethod(String msg) throws RemoteException;

}
```

Below is an example Dipforge bean interface that extends AsyncCallbackHandler:

```
package test.client;

import java.rmi.Remote;
import java.rmi.RemoteException;
import com.rift.coad.daemon.messageservice.AsyncCallbackHandler;

public interface RPCClientTest extends AsyncCallbackHandler, Remote {

   public void runBasicTest(String testString) throws RemoteException,
         MessageTestException;

}
```

Below is an example of an implementation of the above interface:

```
package test.client;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import com.rift.coad.daemon.messageservice.rpc.RPCMessageClient;

public class RPCClientTestImpl implements RPCClientTest {

   public String testID = “”;

   public RPCClientTestImpl() {
   }

   public void runBasicTest(String testString) throws RemoteException,
         MessageTestException {

      System.out.println("The runtime class");
         RPCServerTestAsync async = (RPCServerTestAsync)RPCMessageClient.create(
               "RPCClientTest",RPCServerTest .class,
               RPCServerTestAsync.class,"RPCServerTest");

         testID = async.testMethod(testString);
   }

   public synchronized void onSuccess(String messageId, String correllationId,
         Object result) throws RemoteException {

      if (messageId == testID) {
         System.out.println(result.toString());
      }

   }

   public synchronized void onFailure(String messageId, String correllationId,
         Throwable caught) throws RemoteException {

      System.out.println("The exception is not a message test " +
            "exception : " + caught.getClass().getName());

   }
}
```

Text Message

Server side implementation

The server side implementation for a text message differs from an asynchronous RPC message in that one has to use “MessageHandler” as the implemented interface. And as such it is necessary to only create a single class.

Below is a basic example of what can be done:

```
package test.server;

import java.util.Date;
import com.rift.coad.daemon.messageservice.Message;
import com.rift.coad.daemon.messageservice.MessageHandler;
import com.rift.coad.daemon.messageservice.MessageServiceException;
import com.rift.coad.daemon.messageservice.TextMessage;

public class TextServerTextImpl implements MessageHandler {

   public TextServerTextImpl() {
   }

   public Message processMessage(Message msg) throws MessageServiceException {
      TextMessage textMessage = (TextMessage)msg;
      textMessage.setTextBody(textMessage.getTextBody() + " changed");
      textMessage.acknowledge();
      return textMessage;
   }
}
```

Client side implementation

On the client side implementation a text message differs from an asynchronous RPC message in that it doesn't have an asynchronous class. It must also extend "MessageHandler" as well as "Remote".

Below is an example of a text message interface:

```
package test.client;

import com.rift.coad.daemon.messageservice.MessageHandler;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RPCClientTest extends Remote,
      MessageHandler {

   public void runBasicMessageTest(String testString) throws RemoteException,
         MessageTestException;

}
```

Below is an example of a text message implementation:

```
package test.client;

import com.rift.coad.daemon.messageservice.Message;
import com.rift.coad.daemon.messageservice.MessageProducer;
import com.rift.coad.daemon.messageservice.MessageServiceException;
import com.rift.coad.daemon.messageservice.Producer;
import com.rift.coad.daemon.messageservice.TextMessage;
import com.rift.coad.daemon.messageservice.rpc.RPCMessageClient;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class RPCClientTestImpl implements RPCClientTest {

   public RPCClientTestImpl() {
   }

   public void runBasicMessageTest(String testString) throws RemoteException,
         MessageTestException {
      System.out.println("The beginning of the start test method");
      Context context;
      try {
         context = new InitialContext();
         MessageProducer messageProducer =
```

```
                (MessageProducer)PortableRemoteObject.narrow(
                context.lookup(MessageProducer.JNDI_URL),
                MessageProducer.class);
        Producer producer = messageProducer.createProducer("RPCClientTest");
        TextMessage textMessage = producer.createTextMessage(
                Message.POINT_TO_POINT);
        textMessage.setTarget("TextServerTest");
        textMessage.setReply(true);
        textMessage.setTextBody(testString);
        producer.submit(textMessage);
      } catch (Exception ex) {
        throw new MessageTestException("Text message failure:",ex);
      }
    }

    public Message processMessage(Message message) throws RemoteException,
        MessageServiceException {
      TextMessage textMessage = (TextMessage) message;
      System.out.println("Message is : " + textMessage.getTextBody());
      textMessage.acknowledge();
      return textMessage;
    }
}
```

Using the Command Line Tool

The command line tool comes with Dipforge and both the messageservice.bat and messageservice.sh files for running it are found in the bin folder of Dipforge. The command line tool gives users the ability to list the named queues, list the messages in a named queue and purge a named queue of any messages.

Listing Named Queues

In order to list the named queues supply the following command line options:

-u This must be followed by the JNDI URL for the message server instance.

e.g java:network/env/newbie/message/MessageService

-s The host name information for the Dipforge instance. This must be in the format host:port.

e,.g newbie:2000

-U The username of connection to Dipforge.

-P The password to authenticate the call

-l The option to list named queues

Examples:

These example assume that a user is within the Dipforge bin directory.

Unix:

./messageservice.sh -u java:network/env/newbie/message/MessageService -s newbie:2000 -U test -P 112233 -l

Windows:

messageservice -u java:network/env/newbie/message/MessageService -s newbie:2000 -U test -P 112233 -l

List messages with a named queue

In order to list the messages within a named queues supply the following command line options:

-u This must be followed by the JNDI URL for the message server instance.

e.g java:network/env/newbie/message/MessageService

-s The host name information for the Dipforge instance. This must be in the format host:port.

e,.g newbie:2000

-U The username of connection to Dipforge.

-P The password to authenticate the call

-lq This option must be followed by the name of the queue being queried.

Examples:

These example assume that a user is within the Dipforge bin directory.

Unix:

./messageservice.sh -u java:network/env/newbie/message/MessageService -s newbie:2000 -U test -P 112233 -lq name

Windows:

messageservice -u java:network/env/newbie/message/MessageService -s newbie:2000 -U test -P 112233 -lq name

Purge a named queue of all its message

In order to purge a named queue of all its messages supply the following command line options:

-u This must be followed by the JNDI URL for the message server instance.

e.g java:network/env/newbie/message/MessageService

-s The host name information for the Dipforge instance. This must be in the format host:port.

e,.g newbie:2000

-U The username of connection to Dipforge.

-P The password to authenticate the call

-pq This option must be followed by the name of the queue being queried.

Examples:

These example assume that a user is within the Dipforge bin directory.

Unix:

./messageservice.sh -u java:network/env/newbie/message/MessageService -s newbie:2000 -U test -P 112233 -pq name

Windows:

messageservice -u java:network/env/newbie/message/MessageService -s newbie:2000 -U test -P 112233 -pq name

Building the source

The Dipforge source comes with both a build.sh and build.bat for the various OS's. Running the files in their respective environments will build the source creating a dist folder within the build directory which will contain a built version of Dipforge. It is necessary to alter the various files such as config.xml and all the scripts within the bin folder in order to run Dipforge. In order to run the build.bat within Windows either simply double click the icon or navigate to the build folder within command prompt and type “build”. In Unix simply navigate to the build folder with shell and type “./build.sh”.

Configuration

Dipforge uses pluggable configuration. These means that the configuration source can be changed from XML, which is what it comes with, to LDAP, or a windows Registry. (Currently only XML is supplied by default, though this can be changed by implementing the Configuration Factory API.)

This is a achieved by supplying Dipforge with a system property at startup. This property identifies the name of the Class implementing the Configuration factory API. Other system properties will have to be supplies so that class can be bootstrapped.

Example Windows:

```
set JAVA_OPTS=-Dcoad.config=com.rift.coad.lib.configuration.xml.XMLConfigurationFactory
set JAVA_OPTS=%JAVA_OPTS% -Dxml.config.path="C:\Dipforge\etc\config.xml"
```

Example Unix:

```
JAVA_OPTS="-Dcoad.config=com.rift.coad.lib.configuration.xml.XMLConfigurationFactory"
JAVA_OPTS="${JAVA_OPTS} -Dxml.config.path=/user/local/Dipforge/etc/config.xml"
```

Dipforge configuration is class based. This means that a class requiring configuration will retrieve a configuration object from the factory by suppling its class reference. The configuration factory will return to the requesting object a configuration object instance containing the default information plus the configuration information specific to that object. The class specific information will override the default information where keys clash occur.

Example:

```
// configuration imports
import com.rift.coad.lib.configuration.ConfigurationFactory;
import com.rift.coad.lib.configuration.ConfigurationException;
import com.rift.coad.lib.configuration.Configuration;
.
.
.
class ConfigExample {
.
.
      Configuration config = ConfigurationFactory.getInstance().getConfig(this.getClass());
      String stringValue = config.getString(“StringValue”);
      long longValue = config.getLong(“LongValue”);
      boolean booleanValue = config.getBoolean(“BooleanValue”);
.
.
```

XML Configuration

This configuration source is the default for a Dipforge instance. It relies on simple XML layout, that can be modified by a System administrator without requiring the restart of the Dipforge (With the exception of any entries that affect the core of Dipforge). The simplest way to get a Daemon to re-read its configuration is by redeploying it.

The XML configuration factory and object rely on a layout that identifies the object being dealt with and the values associated with that object. There is typing information associated with these values in order to validate the requests being made on the configuration.

XML Parameters:

```
<object>
<entry>
```

Object

This parameter identifies the object that the configuration information pertains to. If the object is not supplied the information is assumed to be default, affecting an entire installation and not just that object. There can only be one default section per configuration file.

```
<object>
   <!-- default information -->
</object>

<object name="object.name”>
   <!-- object configuration informtion -->
```

```
</object>
```

Entry

This parameter identifies a single configuration value. It defines the key for the value and the type of that value. This type can be either string, long or boolean.

```
<entry key="string" type="string">
   string
</entry>
<entry key="long" type="long">
   7
</entry>
<entry key="boolean" type="boolean">
   true
</entry>
```

Data-source Support

Dipforge supports data-sources. Configured data source are available via JNDI under “java:comp/env/jdbc/”, and can either act in a JDBC controlled transaction or a JTA transaction. Adding a data source to Dipforge can be done via the configuration, requiring a restart or via code, using the DBSourceManager. Note: data-sources added via the DBSourceManager will not be globally available, but will be scoped to the daemon that added them.

XML Configuration

The following example defines how a data source should be added to an XML configuration file.

```
<object name="com.rift.coad.lib.db.DBSourceManager">
   <entry key="db_sources" type="string">
        <!-- The name of the diver comma seperatored-->
        com.mysql.jdbc.Driver,another.driver,
   </entry>
</object>
<object name="com.mysql.jdbc.Driver">
   <entry key="db_source" type="string">
      name
   </entry>
   <entry key="url" type="string">
      url
   </entry>
   <entry key="username" type="string">
      username
   </entry>
   <entry key="password" type="string">
      password
   </entry>
</object>
```

- **db_sources:** This is a comma separated list of database driver objects. These object must be in the class path or Dipforge will not start.
- **db_source:** This is the name of the source that will be bound to JNDI.
- **url:** The url of the database to connect to.
- **username:** The name of the user to use to attach to the database.
- **password:** The password to use to attach to the database.

DB Source Manager

The DB Source Manager is a singleton that manages the database sources for a Dipforge instance. It is the object responsible for adding the data sources during startup, and can be used to add data sources programmatically.

Example:

```
HashMap env = new HashMap();
```

```
env.put(“url”,”url of database”);
env.put(“username”,”username for database connection”);
env.put(“password”,”password for database connection”);
DBSourceManager.getInstance().addDBSource(“com.mysql.jdbc.Driver”,“JNDI name”, env);
```

Note: Any data source added this way will only be available to that daemon.

Hibernate Configuration

Dipforge supplies a Hibernate utility. This utility manages a Hibernate database connections and supplies a session when requested to. It is shipped as a tool to make developers lives easier and it is not required that developers use it.

XML Configuration

The Hibernate utility utilizes the Dipforge configuration tools; as a result the configuration for the utility is attached to a class entry. (Best practice: This class should in some way identify the project or the data store for the project. This will make it easier to manager later on.)

```
<object name="Class Path">
   <entry key="db_dialect" type="string">
      <!--Place the db dialect class name here-->
   </entry>
   <entry key="db_datasource" type="string">
      <!--Place the db source jndi url here -->
   </entry>
   <entry key="hibernate_sql" type="string">
      <!--Enable hibernate sql string values are true|false-->
   </entry>
   <entry key="hibernate_hbm2ddl_auto" type="string">
      <!--Type of hibernate database creation type update|add -->
   </entry>
   <entry key="hibernate_resource_config" type="string">
      <!--Path in daemon deployment archive to resource configuration-->
   </entry>
   <!-- optional flag -->
   <entry key="JDBC2_SUPPORT" type="boolean">
      <!--true|false, this flag is used to switch jdbc verstion 2 support on or off.
         Oracle will require it off,
         Mysql will require it on -->
      supports ”
   </entry>
</object>
```

Use

The Hibernate utility is implemented as a singleton. It can therefore be accessed from anywhere but the first time it is accessed is the first time it will read in its configuration and initialize the hibernate configuration. This initialization process can take some time.

Example:

```
// import
import com.rift.coad.hibernate.util.HibernateUtil;
.
.
.
   // using the hibernate utility.
   Session session = HibernateUtil.getInstance(Foo.class).getSession();
```

Adding Distributed Environment Configurations

By default each Dipforge installation is set as the primary instance. To change this you need to alter the config.xml file within the etc folder in your Dipforge directory. First change primary to false then change the master URL to the intended primary installation.

Before:

```
<object>
```

```
   <entry key="primary" type="boolean">
      <!-- the primary flag-->
      true
   </entry>
</object>
<object name="com.rift.coad.lib.naming.cos.CosContext">
   <entry key="instance_cos_url" type="string">
      corbaloc:iiop:<!-- Local Hostname -->:2000/StandardNS/NameServer-POA/_root
   </entry>
   <entry key="master_cos_url" type="string">
      corbaloc:iiop:<!-- Local Hostname -->:2000/StandardNS/NameServer-POA/_root
   </entry>
   <entry key="cos_user" type="string">
      daemon
   </entry>
</object>
```

After:

```
<object>
   <entry key="primary" type="boolean">
      <!-- the primary flag no set to false indicating that this instance must connect to another-->
      false
   </entry>
</object>
<object name="com.rift.coad.lib.naming.cos.CosContext">
   <entry key="instance_cos_url" type="string">
      corbaloc:iiop:<!-- Local Hostname -->:2000/StandardNS/NameServer-POA/_root
   </entry>
   <entry key="master_cos_url" type="string">
      corbaloc:iiop:<!-- Master Hostname -->:2001/StandardNS/NameServer-POA/_root
   </entry>
   <entry key="cos_user" type="string">
      daemon
   </entry>
</object>
```

Name Service Store

The Name Service Store used by Dipforge is purged by default every time an instance is restarted. This is to enable it to be easily re-configured on different networks, as would happen when running on a laptop, that is assigned an IP address from DHCP. This is because Coadnation will not start if there is unresolvable information within the Name Service Store.

It is quite safe to setup Dipforge so that it does not purge this information on restart. This can be done on servers that have been assigned fixed IP addresses or that have valid DNS information that will not change; and have an application requirements for a persistent name server store.

To disable to disable the purge, add the following section to the configuration:

```
<object name=”com.rift.coad.lib.naming.jacorb.JacORBManager”>
   .
   .
   .
   <entry key=”purge_name_store” type=”boolean”>
      false
   </entry>
<object>
```