Understanding WebSphere ClassLoaders

Version	1
Lead Author	Thomas R Gissel
Contributing Authors	Keys Botzum

Understanding WebSphere ClassLoaders

Acknowledgments

Thanks to all the IBMers that contributed especially: Dan Julin, Stephen Cocks, Indrajit Poddar, Stan Cox, Brian Martin, Melissa Modjeski, Michael Fraenkel, and my manager Jim Stetor.

Overview

This document is intended to help the uninitiated Javaä developer understand the classloaders of WebSphereâ 3.5.x. There are significant differences between the classloading mechanisms of Java 2 and the prior releases of Java that are beyond the scope of this document. For further information on this subject please refer to Appendix A.

What is a Classpath?

A classpath is a list of directories and archives the Java Virtual Machine (JVM), searches when it attempts to load a class.

What is a Classloader?

A classloader is an object that is responsible for loading classes. Given the name of a class, it should attempt to locate or generate data that constitutes the definition for that class.

How do Classloaders Work?

Each Java class within a JVM must be loaded by a classloader, which is intriguing because classloaders are themselves Java classes. This raises the paradoxical question: how do the classloaders get loaded? Lucky for us, the answer is not quite so perplexing. The Java launcher contains one classloader, the bootstrap classloader, which is written in native code, as part of the JVM. The bootstrap classloader s primary function is to load the Java core classes, thereby "bootstrapping" it.

Many of the classes referenced in an enterprise Java solution exist entirely outside of the core classes. For instance, a program may reference another class within its project, or a Java extension. Extensions are Java packages that extend the functionality of the core platform. To separate these two different class types the Java launcher has a different classloader for each of them: the application and extension classloaders, both of which are written in pure Java. Ideally this means that classes will be loaded by their specialized classloader, as shown in the example below.

WhichClassLoader

```
public class WhichClassLoader {

 //Constructor

 WhichClassLoader( ) {

 //do nothing

 }

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

  //Retrieve the classpaths

  StringBuffer bootstrapClassPath=new StringBuffer(System.getProperties().getProperty("sun.boot.class.path"));
  StringBuffer extensionClassPath=new StringBuffer(System.getProperties().getProperty("java.ext.dirs"));
  StringBuffer systemClassPath=new StringBuffer(System.getProperties().getProperty("java.class.path"));
  System.out.println("\nBootstrap classpath= \n"+ bootstrapClassPath + "\n");
  System.out.println("Extension classpath= "+ extensionClassPath + "\n");
  System.out.println("System classpath= "+ systemClassPath + "\n" );

  //Create new object instances

  java.lang.Object object = new java.lang.Object();
  javax.naming.InitialContext initialContext = new javax.naming.InitialContext();
  WhichClassLoader whichClassLoader = new WhichClassLoader();

  System.out.println("\nJava Core file,java.lang.Object, was loaded by: " + object.getClass().getClassLoader());
  System.out.println("\nExtension file, javax.naming.InitialContext, was loaded by: " + initialContext.getClass().getClassLoader());
  System.out.println("\nUser file, WhichClassLoader, was loaded by: " + whichClassLoader1.getClass().getClassLoader() + "\n");
 }

}
```

This is a relatively simple program that retrieves and displays the classloaders' classpath, then creates three new object instances, each of differing class type: a java core class (*java.lang.Object*), a java extension (*javax.naming.InitialContext*), and a user class (*WhichClassLoader*). Finally it prints the classloader that loaded each class. When run *WhichClassLoader* produces the following output.

```
D:\Classpath_Project\src>java −classpath . WhichClassLoader

Bootstrap classpath=
D:\jdk1.2.2\jre\lib\rt.jar;D:\jdk1.2.2\jre\lib\i18n.jar;D:\jdk1.2.2\jre\classes

Extension classpath= D:\jdk1.2.2\jre\lib\ext

System classpath=

Java Core file,java.lang.Object, was loaded by: null

Extension file, javax.naming.InitialContext, was loaded by: sun.misc.Launcher$ExtClassLoader@f0272827

User file, WhichClassLoader, was loaded by: sun.misc.Launcher$AppClassLoader@f023282
```

The results provide several "talking points". For instance, you may be wondering why the user class was loaded by *sun.misc.Launcher$AppClassLoader@f023282,* and not something more generic like *sun.misc.Launcher$AppClassLoader* or the Application ClassLoader. The reason is because the method *getClassLoader()* has a return type of *java.lang.ClassLoader*. When the ClassLoader object is sent to an output stream, its instance name, which changes upon each restart of the JVM, is printed. For our particular run, the instance name of the application classloader is *sun.misc.Launcher$AppClassLoader@**f023282***. If we were to rerun this example, its instance name might be *sun.misc.Launcher$AppClassLoader@**a1b1234;*** the same reasoning holds true for the extension classloader. The interesting thing about the bootstrap classloader is that its instance name is *null*. This is because the bootstrap classloader is not written in Java, so there is no instance of *java.lang.ClassLoader* to return.

One other thing that may have intrigued you about our example is how we were able to retrieve the classpath properties from the System class, e.g. *System.getProperties().getProperty("sun.boot.class.path")*. Given this example, you might think we could force a specific classloader to load a particular class by dynamically altering its classpaths. Let's test this conclusion by modifying *WhichClassLoader*:

WhichClassLoader1

```
import javax.naming.InitialContext;

public class WhichClassLoader1 {

 //Constructor

 WhichClassLoader1( ) {

 //do nothing

 }

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

   //Retrieve the classpaths
   StringBuffer bootstrapClassPath=new StringBuffer(System.getProperties().getProperty("sun.boot.class.path"));
   StringBuffer extensionClassPath=new StringBuffer(System.getProperties().getProperty("java.ext.dirs"));
   StringBuffer systemClassPath=new StringBuffer(System.getProperties().getProperty("java.class.path"));

   //modifying the bootstrapclasspath to include the jar file which contains the InitialContext Class
   // This should force the class to be loaded by the bootstrap classloader???????????

   String fileSeparator = System.getProperty("file.separator");
   String InitialContextJar = "D:"+fileSeparator+"jdk1.2.2"+fileSeparator + "jre"+fileSeparator+"lib"+fileSeparator+"ext"+fileSeparator+"iioprt.jar"
```

```
    bootstrapClassPath.append(System.getProperty("path.separator")).append(InitialContextJar);

   System.setProperty("sun.boot.class.path",bootstrapClassPath.toString());
    System.out.println("\nBootstrap classpath=\n"+ bootstrapClassPath + "\n");
   System.out.println("Extension classpath="+ extensionClassPath + "\n");
    System.out.println("System classpath="+ systemClassPath + "\n" );

    //Create new object instances

    Object object = new java.lang.Object()
    InitialContext initialContext = new InitialContext();
    WhichClassLoader1 whichClassLoader1 = new WhichClassLoader1();

    System.out.println("\nJava Core file,java.lang.Object, was loaded by: " + object.getClass().getClassLoader());
    System.out.println("\nExtension file, Javax.naming.InitialContext, was loaded by: " + initialContext.getClass().getClassLoader());

    System.out.println("\nUser file, WhichClassLoader1, was loaded by: " + whichClassLoader1.getClass().getClassLoader() + "\n");

  }


}
```

When run, the application produces an interesting outcome:

```
Bootstrap classpath=
D:\jdk1.2.2\jre\lib\rt.jar;D:\jdk1.2.2\jre\lib\i18n.jar;D:\jdk1.2.2\jre\classes;D:\jdk1.2.2\jre\lib\ext\iioprt.jar

Extension classpath=D:\jdk1.2.2\jre\lib\ext

System classpath=.

Java Core file,java.lang.Object, was loaded by: null

Extension file, Javax.naming.InitialContext, was loaded by: sun.misc.Launcher$ExtClassLoader@f01b290a

User file, WhichClassLoader1, was loaded by: sun.misc.Launcher$AppClassLoader@f01f290a
```

Not exactly what we might have expected. As a matter of fact, changing these System properties seems to have no effect at all. This demonstrates an important characteristic about these classloaders: their classpaths are unchangeable, or *static*, once the JVM has been instantiated.

Recalling the original purpose of this example, so far the Java launcher's classloaders seem to perform as advertised: the bootstrap classloader loads the core java classes , the extension classloader loads java extensions, and the application classloader loads user classes. However, classloaders are also hierarchical in nature, and follow a "delegating parent" paradigm. This means that every classloader, except the bootstrap classloader, has a parent, and before a classloader tries to load a class it first delegates the responsibility to its parent. Here is how it works: a classloader attempts to load a class if it has not been loaded somewhere within its hierarchy, and its parent is unable to load the class.

Traversing this hierarchy from top to bottom we have the bootstrap classloader followed by the extension and application classloaders. If you think of this hierarchy as a sparse tree structure then the bootstrap classloader would be the root and the application classloader a leaf node.

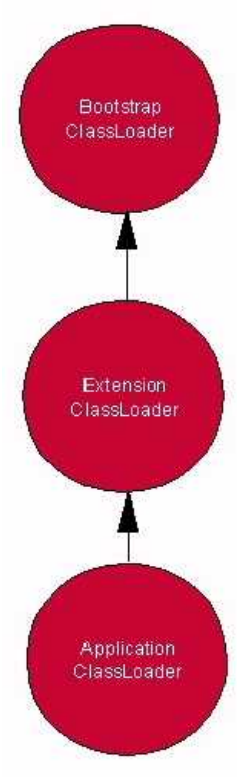

Figure 1

Programmatically this hierarchy can be demonstrated as follows:

ShowHierarchy

```
public class ShowHierarchy{

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

    System.out.println("The System ClassLoader is: " + ClassLoader.getSystemClassLoader().getClass());
   System.out.println("The System ClassLoader's Parent is: " + ClassLoader.getSystemClassLoader().getParent().getClass());
   System.out.println("The System ClassLoader's Parent's Parent is: " +ClassLoader.getSystemClassLoader().getParent().getParent());

  }

}
```

Yielding:

```
D:\WebSphere\Documents\Classpaths\src>java −classpath . ShowHierarchy

The System ClassLoader is: class sun.misc.Launcher$AppClassLoader
The System ClassLoader's Parent is: class sun.misc.Launcher$ExtClassLoader
```

To demonstrate the "delegating parent" relationship, let s place the JAR file that contains *InitialContext.class* into the classpath of each classloader, and execute the WhichClassLoader class from our first example. Remember that once the JVM is running, its classpaths are immutable, so we have to modify them prior to run time using special switches**, −classpath** and **−bootclasspath,** provided by the Java launcher.

```
D:\Classpath_Project\src\java −classpath D:\jdk1.2.2\jre\lib\ext\jndi.jar;.
−XbootclasspathD:\jdk1.2.2\jre\lib\rt.jar;D:\jdk1.2.2\jre\lib\i18n.jar;D:\jdk1.2.2\jre\classes;D:\jdk1.2.2\jre\lib\ext\jndi.jar WhichClassLoader

Bootstrap classpath= D:\jdk1.2.2\jre\lib\rt.jar;D:\jdk1.2.2\jre\lib\i18n.jar;D:\jdk1.2.2\jre\classes;D:\jdk1.2.2\jre\lib\ext\jndi.jar

Extension classpath= D:\jdk1.2.2\jre\lib\ext

System classpath= D:\jdk1.2.2\jre\lib\ext\jndi.jar;.

Java Core file,java.lang.Object, was loaded by: null
Extension file, javax.naming.InitialContext, was loaded by: null
User file, WhichClassLoader, was loaded by: sun.misc.Launcher$AppClassLoader@f01a2987
```

As you can see, the InitialContext class object was loaded by the bootstrap classloader as we had predicted, but do you understand why? For those a little lost, let s detail how the events unfolded:

1. *WhichClassLoader* requested a new instance of *InitialContext* via *javax.naming.InitialContext initialContext = new javax.naming.InitialContext();*
2. The application classloader checked to see if the *InitialContext* class had been loaded somewhere within its hierarchy, i.e. by? bootstrap, extension, or itself.
3. The check returned null.
4. Following the "delegating parent" model, the application classloader deferred to the extension classloader.
5. Again following the "delegating parent" relationship, the extension classloader deferred to the bootstrap classloader.
6. The bootstrap classloader, knowing that it has no parent, attempted to load the class.
7. The bootstrap classloader successfully loaded the *InitialContext* class.
8. A new instance *InitialContext* was created and returned to *WhichClassloader*.

This scenario needs some further explanation. It might seem obvious that the application classloader would be the first to receive the request to get a new instance of *InitialContext* because it is at the bottom of the hierarchy, but this is not the case. With the advent of Java 2, classes load through the caller's classloader, which may or may not be the classloader at the bottom of the hierarchy. In our example the caller was *WhichClassLoader*, and we know that *WhichClassLoader* was loaded by the application classloader from the line *User file, WhichClassLoader, was loaded by: sun.misc.Launcher$AppClassLoader@f0df070a.*

The fact that Java 2 classloaders follow a delegating parent, hierarchical paradigm allows you to do some interesting things, but problems can arise if a request to load a class is made from places other than a leaf node. For example, lets modify *WhichClassLoader*, renaming it to *WhichClassLoader2*, and create two more classes, *WhichClassLoader3* and *WhichClassLoader4*.

Classloading Problem

WhichClassLoader2

```
public class WhichClassLoader2 {

 //Constructor
 WhichClassLoader2() {
     //do nothing
 }

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


      //Retrieve the classpaths
     StringBuffer bootstrapClassPath=new StringBuffer(System.getProperties().getProperty("sun.boot.class.path"));
     StringBuffer extensionClassPath=new StringBuffer(System.getProperties().getProperty("java.ext.dirs"));
     StringBuffer systemClassPath=new StringBuffer(System.getProperties().getProperty("java.class.path"));
```

```
        System.out.println("\nBootstrap classpath="+ bootstrapClassPath + "\n");
        System.out.println("Extension classpath="+ extensionClassPath + "\n");
        System.out.println("System classpath="+ systemClassPath + "\n" );


        //Create new object instances

        java.lang.Object object = new java.lang.Object();
        javax.naming.InitialContext initialContext = new javax.naming.InitialContext();
        WhichClassLoader2 whichClassLoader2 = new WhichClassLoader2();
        WhichClassLoader3 whichClassLoader3 = new WhichClassLoader3();

        System.out.println("\nJava Core file,java.lang.Object, was loaded by: " + object.getClass().getClassLoader());
        System.out.println("\nExtension file, javax.naming.InitialContext, was loaded by: " + initialContext.getClass().getClassLoader());
        System.out.println("\nUser file, WhichClassLoader2, was loaded by: " + whichClassLoader2.getClass().getClassLoader() + "\n");
        System.out.println("\nUser file, WhichClassLoader3, was loaded by: " + whichClassLoader3.getClass().getClassLoader() + "\n");

        whichClassLoader3.getTheClass();

  }
}
```

In a separate file *WhichClassloader3.java*:

```
public class WhichClassLoader3  {
  public WhichClassLoader3  ()  {


  }


  public void getTheClass()  {
    WhichClassLoader4 wcl4 = new WhichClassLoader4 ();


    System.out.println("WhichClassLoader4 was loaded by " + wcl4.getClass().getClassLoader());

 }

}
```

In a separate file *WhichClassloader4.java*

```
public class WhichClassLoader4  {
 WhichClassLoader4  ()  {


  }
}
```

We will leave *WhichClassLoader2* and *WhichClassLoader4* in the current directory, so that they reside only in the application classloader's classpath. Next we will move *WhichClassLoader3* into the extension's classloader's classpath.

```
D:\Classpath_Project\src>ls
WhichClassLoader2.class
WhichClassLoader2.java
WhichClassLoader1.java
WhichClassLoader4.class
WhichClassLoader4.java


D:\Classpath_Project\src>ls D:\jdk1.2.2\jre\lib\ext\
WhichClassLoader3.jar
cosnaming.jar
iiimp.jar
iioprt.jar
jndi.jar
providerutil.jar
rmiorb.jar
rmiregistry.jar
```

Let's find out what happens when we execute *WhichClassLoader2*.

```
D:\Classpath_Project\src>java −classpath . WhichClassLoader2
Bootstrap classpath=D:\jdk1.2.2\jre\lib\rt.jar;D:\jdk1.2.2\jre\lib\i18n.jar;D:\jdk1.2.2\jre\classes

Extension classpath=D:\jdk1.2.2\jre\lib\ext

System classpath=.

Java Core file,java.lang.Object, was loaded by: null

Extension file, javax.naming.InitialContext, was loaded by: sun.misc.Launcher$ExtClassLoader@f01b3492

User file, WhichClassLoader2, was loaded by: sun.misc.Launcher$AppClassLoader@f01f3492


User file, WhichClassLoader3, was loaded by: sun.misc.Launcher$ExtClassLoader@f01b3492

java.lang.NoClassDefFoundError: WhichClassLoader4
 at WhichClassLoader3.getTheClass(WhichClassLoader3.java:8)
 at WhichClassLoaderTest.main(WhichClassLoaderTest.java:38)
Exception in thread "main"
```

You may be asking yourself why the load of *WhichClassLoader4* failed since it was clearly in the application classloader's classpath. The failure occurred not because the class was not in the classloader hierarchy, but because it was not found in *WhichClassLoader3's* classloader hierarchy. To get a better idea of what happened, let s look at things from the perspective of the classloader that loaded *WhichClassLoader3*, the extension classloader.

First, an instance of *WhichClassLoader4* was requested, so following the Java 2 "delegating parent" paradigm, the extension classloader first checked to see if the class was already loaded somewhere in its hierarchy, but that check failed. It then asked its parent, the bootstrap classloader, to load the class; but the request returned a *NoClassDefFound* exception. The extension classloader caught the exception and then resorted to its last option; it attempted to load the class. After searching its classpath it could not find the *WhichClassLoader4* class definition, so it again threw the *NoClassDefFoundError* exception. At this point there was no classloader to catch the exception, so the exception was sent to the screen and the program aborted.

This begs the question, if problems such as this can occur, why bother with having multiple classloaders? Why not go back to the pre−Java 2 framework of one system classloader that loads everything? Are the benefits of the classloader hierarchy and "delegating parent" paradigm really worth the problems they can create? In short, yes for these reasons:

1. Protection − Java predefines the bootstrap and extension classloaders and by default only leaves the applications classloader definition up to the user. Any classes defined here will not over overwrite the extensions or core classes because of the "delegating parent" paradigm.
2. More User−Friendly − Along the same theme, you no longer have to remember to place the classes.zip file in the −classpath property because the bootstrap and extension classloaders are predefined.
3. Separation − This is probably the most significant advantage that this new classloader definition provides. In our experiences up to this point it admittedly doesn't seem all that useful, but what we haven't touched on yet is the fact that Java lets you define your own classloaders. Using inheritance, you can change the once sparse tree into a bushy one.

The best way to understand the benefits that this model provides is an analogy to an operating system's namespace. Let's say we are developing an application called *MyApp* of which there are three versions: one that is in production, another in Beta testing, and yet another still being developed. We want to run any or all versions of the application at any time, so we place them in separate directory structures.

Figure 2

Files that are common to all versions are placed into the *MyApp* directory, and version specific files under the *V1*, *V2*, *V3* directories. Using this approach we efficiently use hard drive space while maintaining the version separation. The same idea holds true for classloaders, the most generic classes, core classes, are placed at the root of the hierarchy, while version specific classes are placed on the leaf nodes.

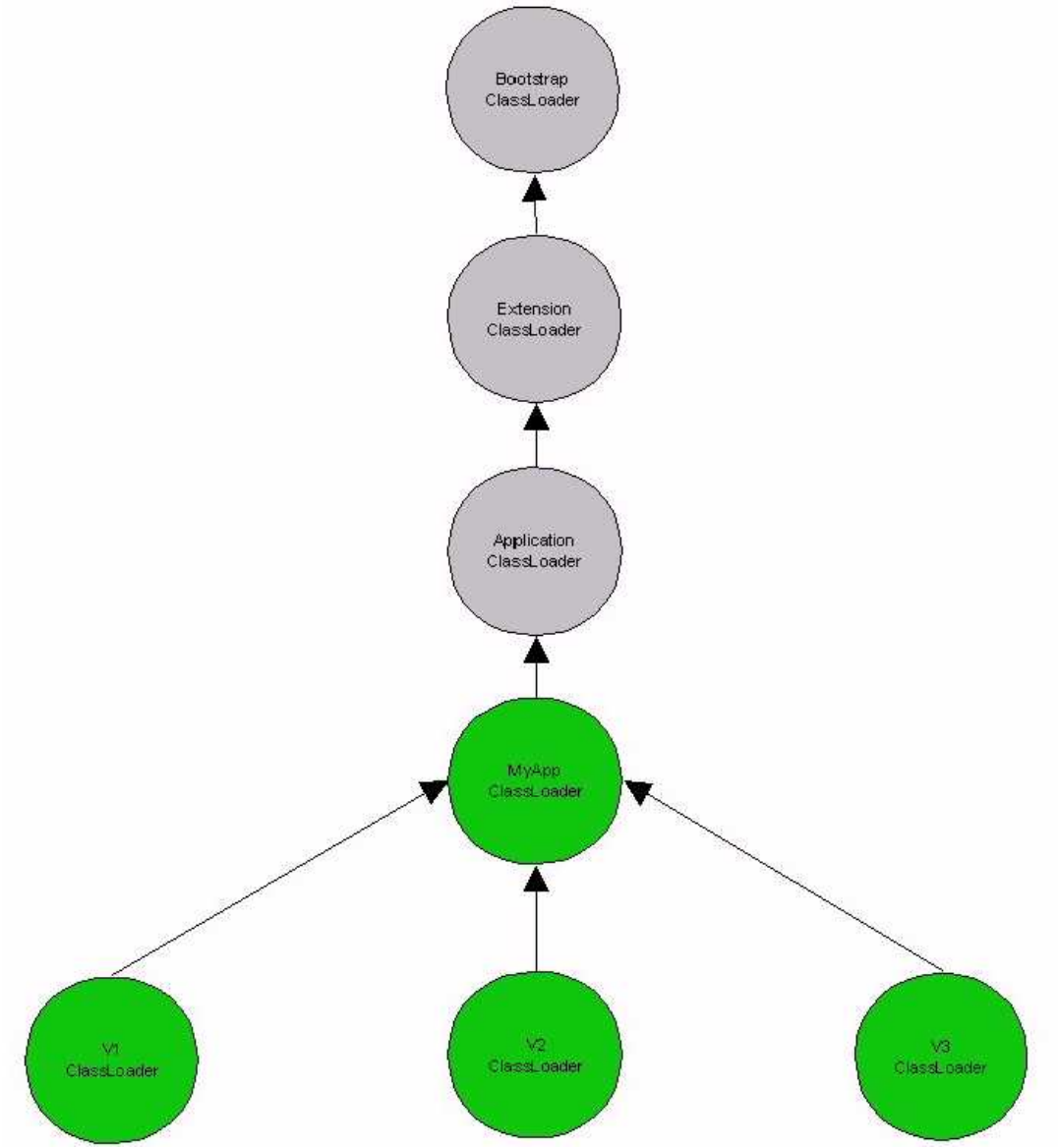

Figure 3

WebSphere's Classloaders

We have now built a foundation of knowledge that will help us understand WebSphere s classloaders, but before we start examining their specifics, let's take a step back and look at the components that make up the WebSphere Environment.

WebSphere Environment

When WebSphere is said to be running on a machine there is one Java process that is always present, the Administrative Server. The Administrative Server is the heart of WebSphere, providing five essential functions:

1. Manages all the Application Servers that are running on a node. It performs operations such as starting, stopping, monitoring.
2. Provides centralized services that Application Servers need to operate, including Java Naming and Directory Interface (JDNI) lookups and transaction logging, and acts as a security server (access to authentication and authorization database).
3. Acts as a centralized point for the management and sharing of WebSphere configuration information.
4. Maintains global information about the state of the systems
5. Responsible for the CORBA Location Service Daemon, i.e. the piece that dynamically allocates TCP/IP ports to different CORBA listener processes

When you start WebSphere in the preferred way, via the startupServer.sh script on UNIX/Linux, or the "IBM WS AdminServer" service on Windows platforms, the Administrative Server's application classloader classpath is set based on the value of the property: *com.ibm.ejs.sm.adminserver.classpath* in the *<WAS_ROOT>/bin/admin.config* file. Alternatively, you could also start WebSphere by executing the <WAS_ROOT>/bin/debug/*adminserver.[bat|sh]* script, in which case its classpath is defined by the CLASSPATH environment variable.

The Administrative Server's managerial responsibilities require it to be the parent process of all application servers running on that node, and as such, these application servers inherit the Administrative Server's classpath. To override this default behavior, you may specify an alternative classpath to be inherited by the Application Servers, by defining the property *com.ibm.ejs.sm.adminserver.managedServerClassPath* in the *<WAS_ROOT>/bin/admin.config* file.

The Application Server s application classloader classpath, unlike "normal" Java processes, is composed of three different components:

1. The first classpath element may be specified in one of two ways: by using the −classpath option in the Command line arguments of the Application Server, or by specifying a CLASSPATH environment variable. If you use both, the −classpath supersedes the CLASSPATH environment variable.
2. The second element contains the JAR files from the JDBC drivers that are installed on the node under which the application server resides.
3. The third element is formed by appending the classpath of the Administrative Server process to the other two classpath constituents.

In addition to the classloaders that the Java launcher provides to JVM, WebSphere has two other classloaders that perform specialized tasks, the *DynamicClassLoader* and *JarClassLoader*. The *DynamicClassLoaders* are unique in that there are a variable number of them, from 0 to *n*, one for each Web Application defined within the Application Server. Their classpaths are determined by the value of Web Application *Classpath* property on each Web Application.

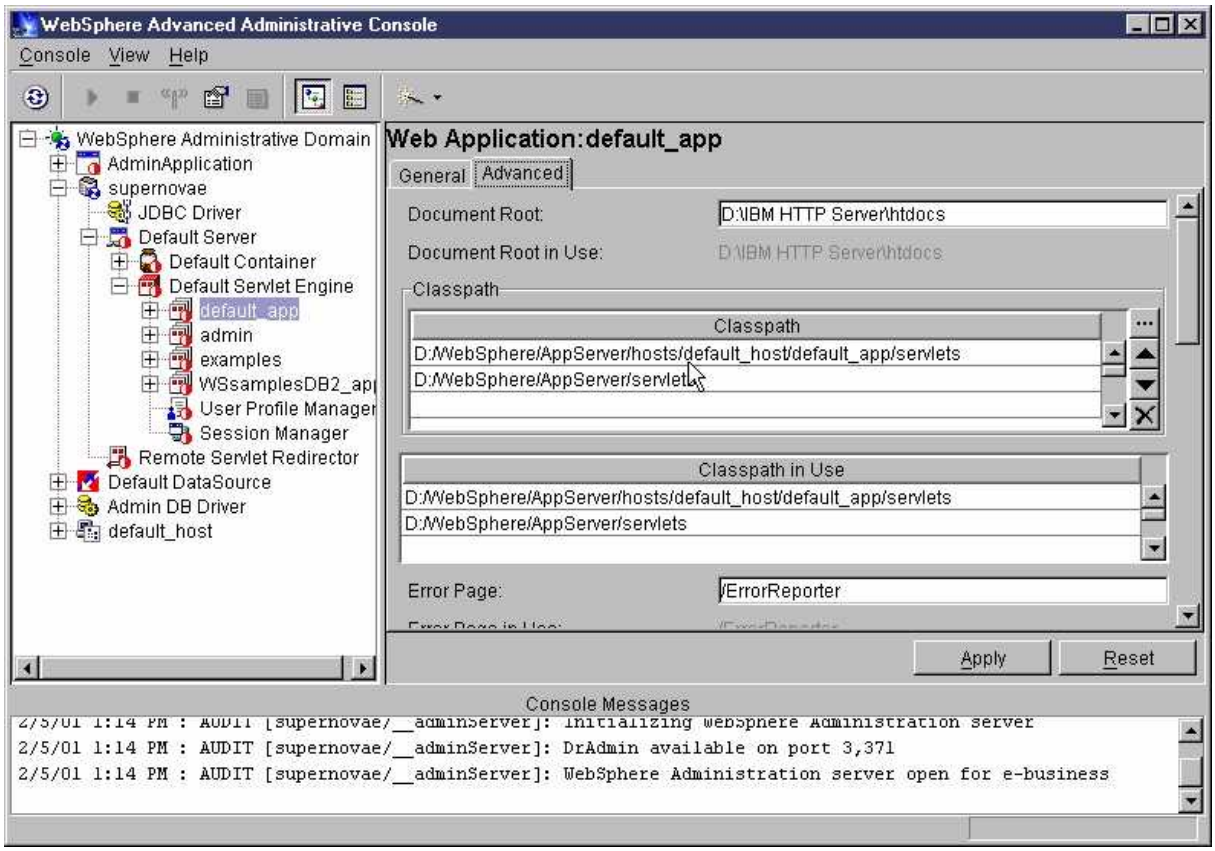

Figure 4

In this example, the "Default Server" has four *DynamicClassLoaders*, one for each of the Web Applications: default_app, admin, example, and WSsamplesDB2_app. Each of these Web Applications has two classpath attributes: "Classpath" and "Classpath in Use". The "Classpath" is what the Web Application Classpath gets set to upon the next restart of the Application Server, whereas "Classpath in Use" was the Web Application Classpath used the when the Application Server was running last.

The *JarClassLoader* is a singleton, only one per Application Server, whose classpath is a composite of the Node Dependent classpath and the JAR files of EJBs deployed within the Application Server. The Node Dependent classpath is a node−wide attribute, meaning it is shared by all Application Servers on that node, which serves two purposes:

1. Allows users to specify any external dependencies while deploying an EJB JAR file
2. Provides a means of adding JAR files and directories to the *JarClassLoader*'s classpath

Figure 4 shows an Application Server that contains one EJB Container, however, WebSphere allows you to define as many EJB Containers within an Application Server as you want. With this in mind, you might think that each EJB Container will have its own *JarClassLoader*, but this is not the case. As sited above, there is only one *JarClassLoader* per Application Server, regardless of how many EJB Containers are within that server. This is one of the reasons that we recommend only one EJB Container per Application Server.

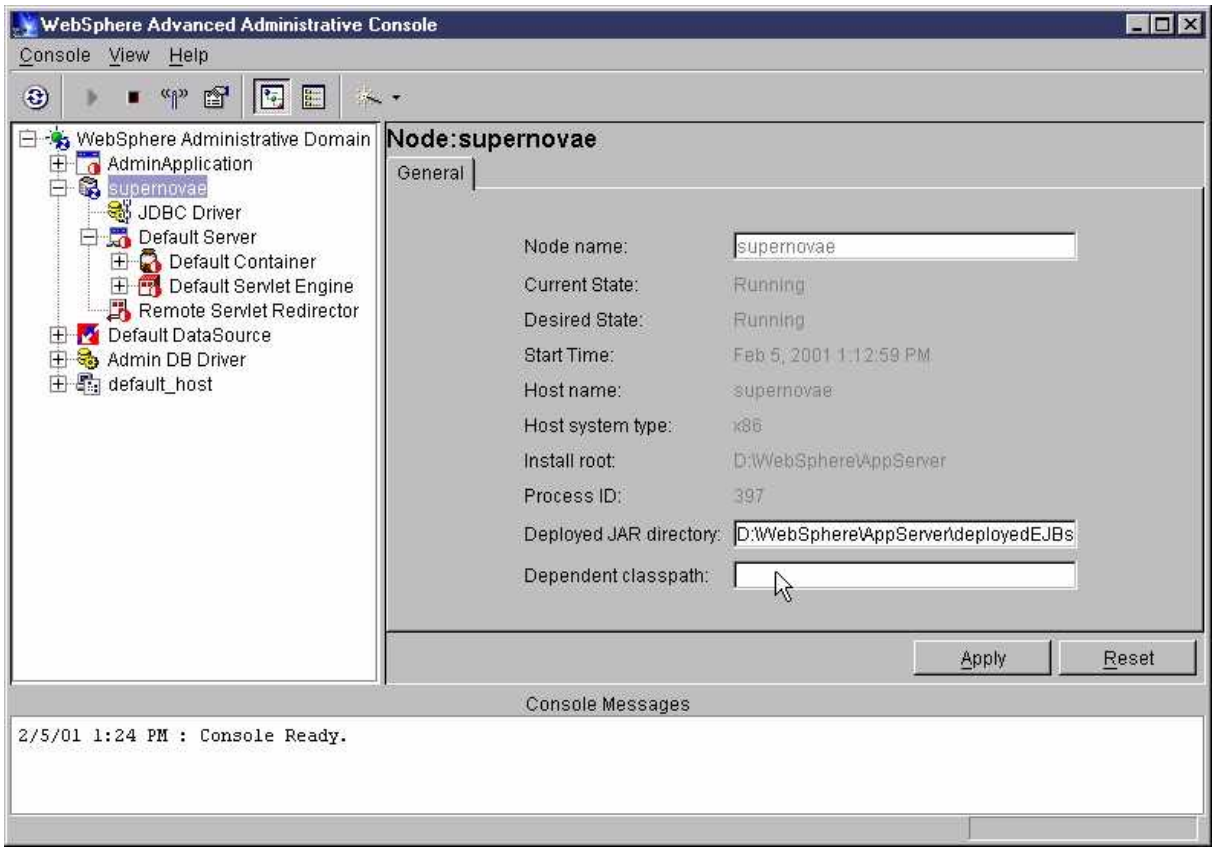

Figure 5

One thing that sets the *DynamicClassLoader* and *JarClassLoader* apart from the classloaders provided by the Java launcher is their dynamic characteristics. The *JarClassLoader* is dynamic in the sense that new JAR files can be added to its classpath by deploying new EJBs into the application server while it is running. If you change an EJB interface after it has been deployed into the EJB Container, you must redeploy it. However if the changes do not involve either of the interface classes, home or remote, then a simple restart of the Application Server will suffice. When the classes loaded by the *DynamicClassLoader* change, you don t need to restart the JVM restart, so therefore they are dynamic. This feature allows existing class definitions to change without restarting the Application Server. WebSphere provides this ability because the underlying classes of an application still in development can change frequently, and it would be very tedious to restart the Application Server restart every time one these classes changed. To avoid this, the Web Applications within WebSphere can check for newer versions of classes and reload them if necessary. The frequency of this check is controlled by the configurable reload parameter of each Web Application . If you set this parameter to zero, the *DynamicClassLoader* becomes static.

Fortunately these new classloaders do not change the previously defined hierarchy, but instead extended it. The application classloader becomes the parent of the *JarClassLoader,* and the *JarClassLoader* the parent of the *DynamicClassLoader*; a visual representation is shown below.

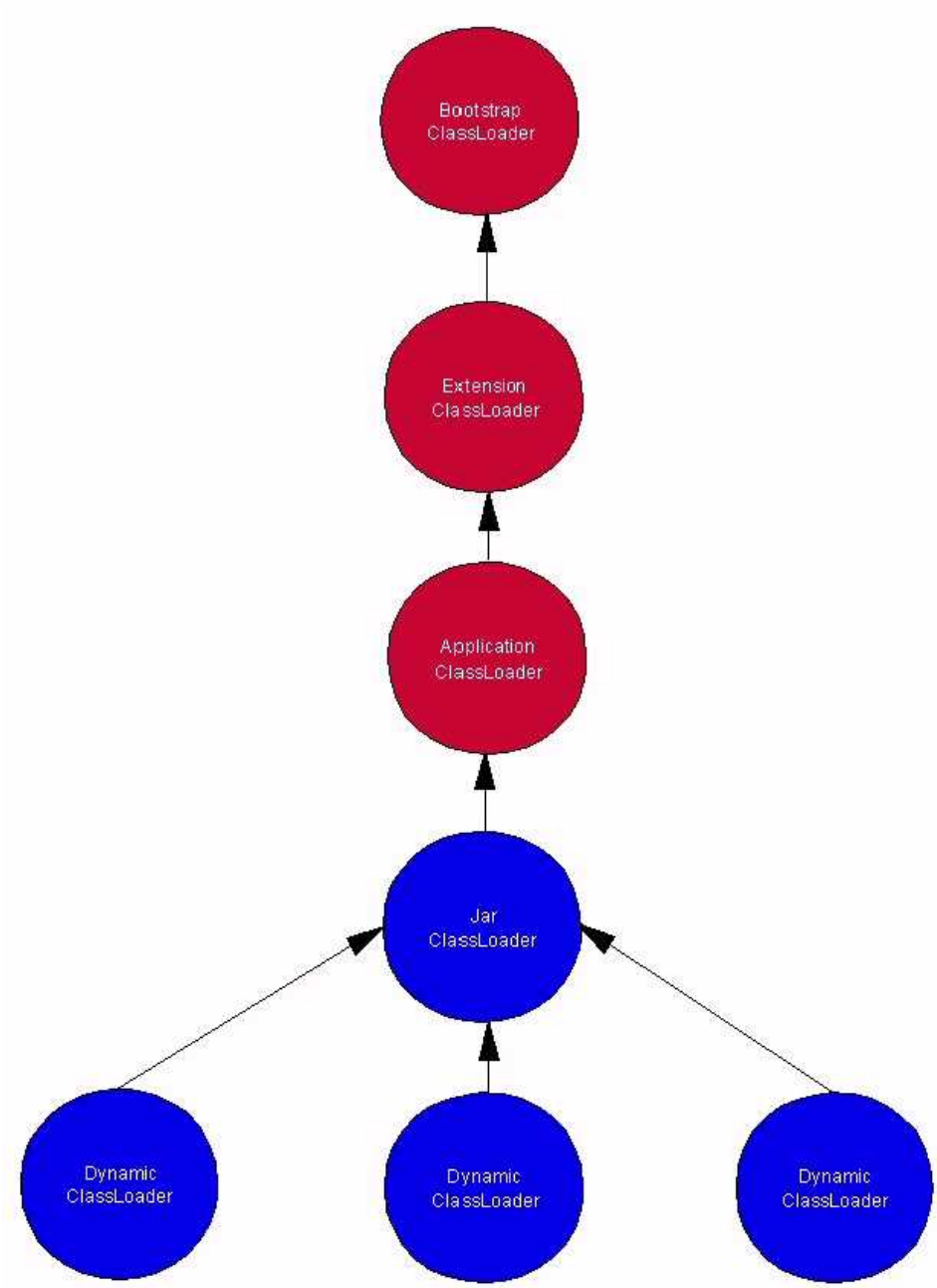

Figure 6

Recalling that classloaders follow a "delegating parent" relationship, we can infer that when an Application Server searches for a class it does so by scanning classpaths in the following order, from top to bottom.

1. Bootstrap Classpath
2. Extension Classpath
3. Application Server Command line classpath attribute or environment CLASSPATH
4. Administrative Server classpath
5. Node Dependent classpath
6. List of all the EJB JARs that have been deployed in the Application Server
7. Web Application classpath

Before taking this list as law, remember that the order of classloaders depends on the context. Thus a servlet loaded by the *DynamicClassLoader* can see the entire classloader hierarchy, but an EJB may only see classes from the *JarClassLoader* to the bootstrap classloader.

With the addition of these new classloaders and classpaths, it can become a daunting task knowing which classpath JARs and class files should be placed in for proper application behavior. We recommend you use a systematic, cookbook approach when deploying applications.

WebSphere File Placement Cookbook

Each application deployed within WebSphere is unique, but there are certain building blocks upon which most of them are created. Using these building blocks as a guide, we can create a cookbook that system administrators and developers can use to deploy their applications. The examples below build off each other, starting with the simplest deployment and ending with the most complex. Unless otherwise specified by the word "Only" in the section title, the more complex examples are assumed to contain all the classes referred to in the previous sections.

EJBs Only

Applications of this type are typically only referenced by EJB clients with no external dependencies. Consequently, the EJBs can be deployed without manipulating any of the classpaths. The Administrative Server can deploy a EJB JAR located anywhere on the file system; when you deploy an EJB jar, the deployed JAR will automatically be created and placed in the Deployed JAR directory. This new deployed JAR can then be loaded into the EJB Container.

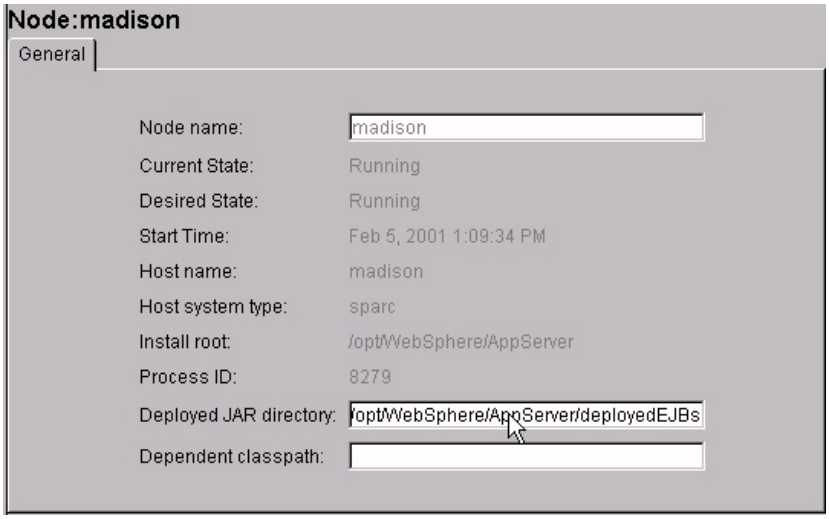

Figure 7

Servlets Only

Deploying an application that contains only servlets is also relatively easy. The only classpath change required is the addition of the servlet JAR files to the Web Application Classpath, and then you just need to define each servlet in the Web Application.

Servlets and EJBs

To deploy an application of this type, combine the EJBs Only and Servlets Only recommendations.

JSPs and Static Files

Applications that have JSPs and static files, in addition to the file types found either of our first three application types, require only the minor configuration change of placing the JSPs and static files into the directory defined as the "Document Root" within the Web Application.

Figure 8

Though the file placement for applications of this type is not difficult, they do require that you make an important decision. Are you going to allow your HTTP Server to load the static files, or is that task going to be delegated to WebSphere? The current recommendation is to let WebSphere deal with Java classes and leave the static content to the HTTP Server. There are several reasons for this, but one of the more compelling is that WebSphere requires the File Serving servlet to service static files. This adds extra overhead within the Application Server and is generally slower than going through an HTTP Server.

Classes That Use Java Native Interface (JNI)

To avoid runtime exceptions, application classes use JNI must be placed in the classpath parameter of the Application Server.

Servlet−Supporting Classes

Before you deploy servlet−supporting classes, (classes that are referenced by servlets) you need to determine if those supporting classes can safely be reloaded.

1. If the supporting classes can safely be reloaded, then you can package them with the servlets and place them in the Web Application Classpath.
2. If the supporting classes can not be safely reloaded, then you need to place them in a separate JAR and put them in the application classloader's classpath, preferably via the classpath parameter of the Command line arguments of the Application Server. All classes that have fields used in HTTP Session State fall into this category.

EJB Helper Classes/Interfaces

Many applications use EJB helper classes and interfaces. The finder helper is probably the most well known entity in this category. Finder helpers are EJB interfaces generated by VisualAgeâ for Java that contain the implementation code for the finder methods on the home interface. There are basically two types of EJB helper objects: ones that are used solely by the EJB bean classes, and others referenced in at least one of the EJB interfaces.

1. **Objects referenced only by EJB bean classes can be:**

· Packaged with the EJBs in the EJB JAR File

· Placed into the application classloader's classpath, preferably via the Command line arguments

· Put into the Dependent classpath

2. **Objects referenced in at least one of the EJB Interfaces can be:**

· Packaged with the EJBs in the EJB JAR File

· Put into the Dependent classpath

Note that placing class files into the Dependent classpath may cause some headaches when attempting to clone across WebSphere machines or nodes. This will be addressed more fully in the next section.

Classes That Reference EJBs and Are Referenced by Servlets

Remote calls from servlets to EJBs are expensive operations, so to minimize this cost you might want to cache the attributes of EJBs in local objects. VisualAge for Java's access beans are commonly used for this purpose. Access beans are generally reloadable objects, but not always. Therefore the deployer needs to know if the classes are reloadable.

1. If the classes can safely be reloaded then they should be placed into the Web Application Classpath, e.g. access beans
2. If the classes can not be reloaded then it is best to place them into the application classloader's classpath via the classpath parameter of the Application Server's Command line arguments

Classes That Are Referenced from Both Servlets and EJBs but Reference Neither

These are relatively rare, but do exist. If your application contains objects of this type, place them into the application classloader's classpath to ensure visibility to both servlets and EJBs.

Other Classes

Some applications contain classes that do not fit into any of the above categories. Place these classes into the application classloaders classpath.

WebSphere Classpath Summary Table

Classpath	**Uses**
Bootstrap Classpath classpath	Core Java Classes − DO NOT MODIFY
Extension Classpath classpath	Java Extensions − DO NOT MODIFY
Application Classpath classpath	· Classes referenced from servlets whose objects are added to sessions, · Classes that call Java Native Interface (JNI) methods. · EJB client JAR files · Java Classes that do not fall into any of the other categories
Administrative Server classpath	Classes used by the Administrative Server −DO NOT MODIFY except when instructed to do so when applying WebSphere patches or when updating the classpath to add a JDBC driver to the Administrative Repository.
Node Dependent classpath	Classes that are referenced by EJBs in their interfaces. This lets the Administrative Server deploy the EJBs.
EJB classpath	EJBs and classes within the EJB jar file − implicitly added when EJBs are deployed into the EJB Container.
Web Application classpath	· Directories or JAR files with servlet classes. · Directories or JAR files with helper classes that are not included in the servlet JAR file and that are expected to be reloadable. · Directories or JAR files with access beans. Access beans are classes which are referenced from servlet classes and that refer to EJBs.

JAR Packaging Best Practice

In this section we will provide a JAR packing best practice that if followed will dramatically simplify application deployment.

For a typical application it is recommended that you divide your class files into 5 different categories.

1. Servlets
2. EJBs
3. EJB Clients
4. EJB Dependent Classes
5. Classes that are none of the above

For an application called BigApp the JAR files would have the following names:

1. BigAppServlet.jar − containing servlet code
2. BigAppEJBs.jar − containing EJBs
3. BigAppEJBClient.jar − containing the client only JAR file for all of the EJBs.
4. BigAppDependentClass.jar − containing EJB Dependent class
5. BiggAppClasses − containing the remaining class files.

With the classes packaged this way it is the JAR file placement is easy.

JAR File	***Classpath***
BigAppEJBClient.jar.	Application Server classpath
BigAppEJBs.jar	Deploy into the EJB Container
BigAppEJBClient.jar	Application Server classpath
BigAppDependentClass.jar	Node Dependent classpath
BiggAppClasses	Application Server classpath

Jar Packaging Best Practice Table

Class Type	***JAR Naming Convention***	***Classpath***
Servlet Classes	*Servlet.jar	Web Application Classpath
EJB Client Classes	*EJBClient.jar	Application Server classpath
EJBs	*EJB.jar	Deploy into the EJB Container
EJB Dependent Classes	*DependentClasses.jar	Node Dependent classpath
Classes that are none of the above	*Classes.jar	Application Server classpath

Classloader Problems in Common WebSphere Topologies

In this section we demonstrate some of the problems that can occur if you don t follow the guidelines specified above. To illustrate this we will purposely misconfigure WebSphere's Trade application in different topologies.

Simple Topology

The simplest topology is one in which everything needed to run WebSphere is contained within one machine, or node, but even in this environment things can go wrong.

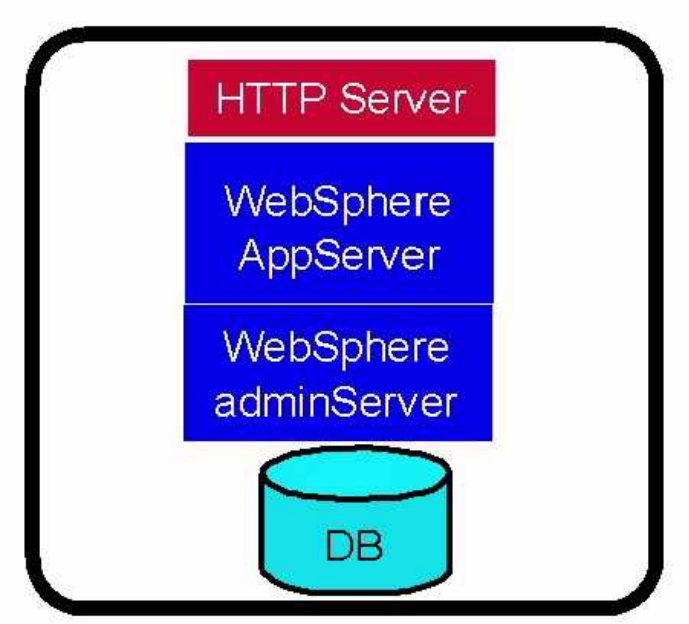

Figure 9

Example 1: Dependent Jars
In this example we will deploy the Trade application class files into three different JARs: *Trade2Client.jar* containing EJB client classes, *TradeEJBs.jar* containing the EJBs and special EJB Objects, and *TradeOther.jar* containing all the other classes. Following standard WebSphere practice, we will place *Trade2Client.jar* in the Web Application Classpath, and deploy the EJBs within *TradeEJBs.jar* into the EJB Container. Now, where we should place the *TradeOther.jar* file? Let s try placing it into a leaf node and use the "delegating parent" paradigm to our advantage. This way all classes in *TradeOther.jar* are visible to each class within WebSphpere's classloader hierarchy.

Unfortunately using this approach gives less than desirable results. We can t even logon to the Trade application. The application appears to do nothing except reset the *Username* field.

Figure 10

Looking in the logs we see error messages saying:

TradeServletAction.doLogin() operation failure − userID/passwd: uid:2 / xxx Error is: java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:com.ibm.ejs.container.UncheckedException: ; nested exception is: java.lang.NoClassDefFoundError: trade/TradeStaticContext

Without knowing anything about the application it is very difficult to know what is going on. Luckily we have seen the *java.lang.NoClassDefFoundError*Exception before, and we know that it is a classloader problem. Judging from the abbreviated stack trace, it looks as though the container or something within the container can t find the *trade/TradeStaticContext.class*. We need to find which JAR file contains *the TradeStaticContext* class . After some investigation, you ll find that it is in *TradeOther.jar*. Now do you know what the problem is? If you re lost, here s a hint: the only things that reside inside the WebSphere Container are EJBs, and the JarClassLoader loads all EJBs . If you are still having problems, draw a diagram of the classloaders, then place the EJB and *trade/TradeStaticContext* beside their appropriate classloaders. Finally draw a simple code flow path.

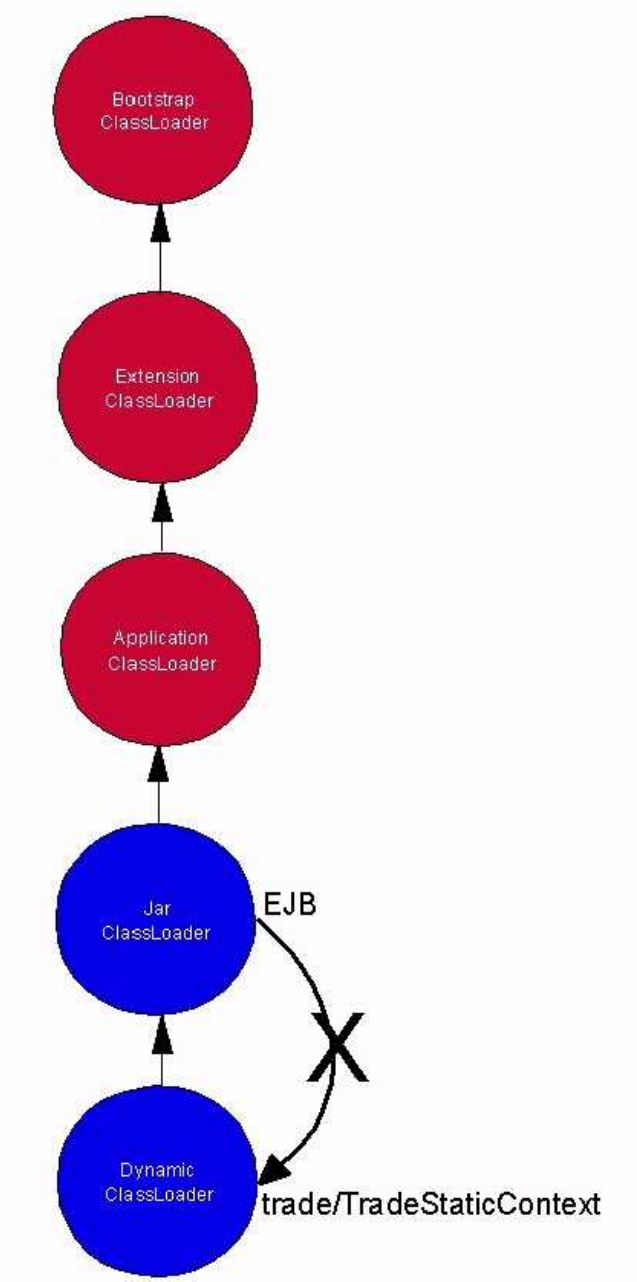

Figure 11

One of our EJBs is trying to use the *trade/TradeStaticContext* class, but it can't find it because *TradeStaticContext* only resides in the classpath of the *DynamicClassLoader*.

The simple way to resolve this problem is to move *TradeOther.jar* to a place where the EJBs can see it, say the application classloader. This solves the *Username* reset problem, but now we see a blank page displayed during Logon.

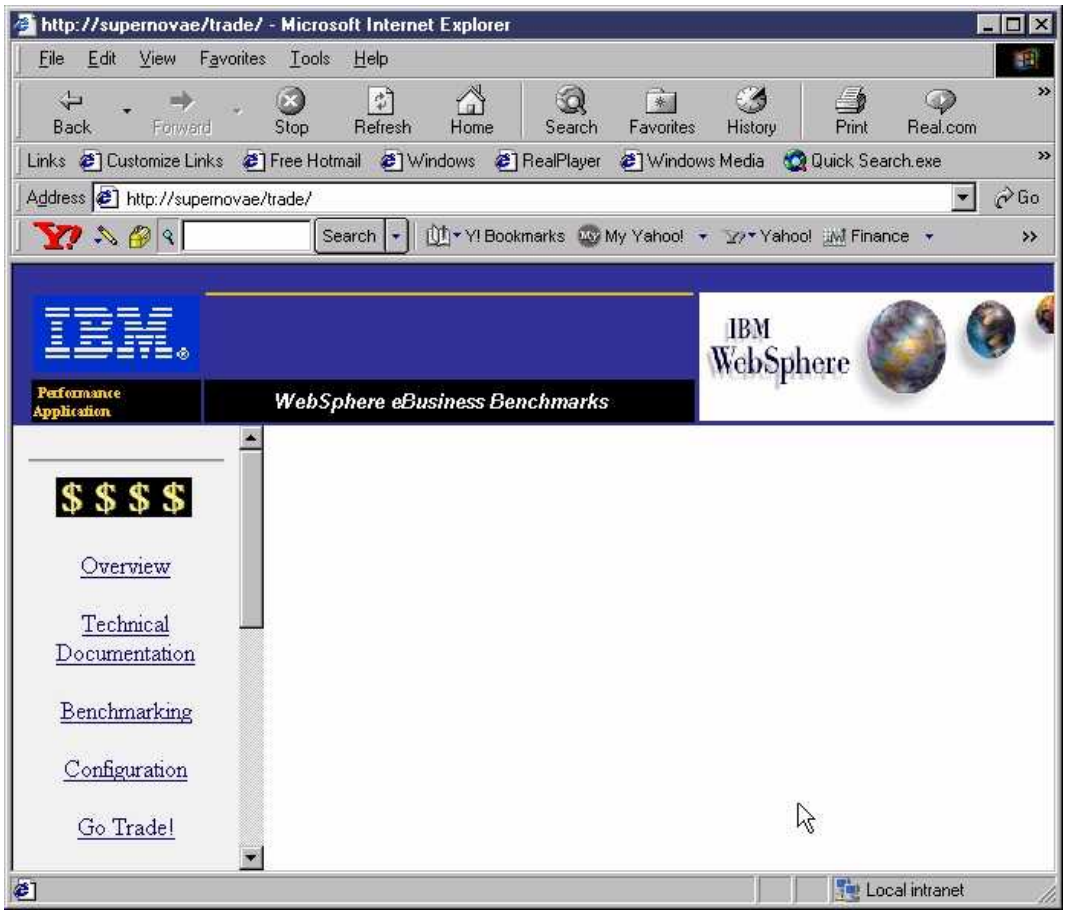

Figure 12

This is obviously not the desired behavior, so let s check the logs again for exceptions. This time we see:

TradeServlet: General Exception: action=login userID=uid:2 Exception=java.lang.NoClassDefFoundError: trade/TradeInterface

This tells us that we have another *NoClassDefFoundError,* this time seemingly coming from *TradeServlet,* trying to find *trade/TradeInterface*. Checking to see where these files reside, we find them in *Trade2Client.jar*, whereas *trade/TradeInterface* is an interface to our Trade EJB. This is very perplexing because this exception seems to be indicating that a class loaded by the *DynamicClassLoader* is having trouble finding something loaded by the application classloader. If we do a bit more research, we discover that some of the classes in *TradeOther.jar* are access beans, and that they are the ones actually throwing the exception. Since theapplication classloader loaded the access beans, they are unable to find the EJBs that were loaded lower in the classloader hierarchy, by *DynamicClassLoader*.

Since placing *TradeOther.jar* in either the *DynamicClassLoader* or application classloader classpath is in an invalid configuration, our only remaining option is to place *TradeOther.jar* into the Dependent classpath, thereby letting the JarClassLoader load it. This should resolve our two problems:

1.

***TradeServletAction.doLogin() operation failure − userID/passwd: uid:2 / xxx Error is: java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:com.ibm.ejs.container.UncheckedException: ; nested exception is: java.lang.NoClassDefFoundError: trade/TradeStaticContext* . **This should be resolved because the problem resulted from an EJB not having visibility to the *TradeStaticContext class.* With this configuration, both the EJB and classes within *TradeOther.jar* are loaded by the JarClassLoader.**

2. ***TradeServlet: General Exception: action=login userID=uid:2 Exception=java.lang.NoClassDefFoundError: trade/TradeInterface*. The access beans can now see the EJBs, so this to should go away.**

Let s hope that no new problems arise. Keep your fingers crossed....

Figure 13

Success, finally. Our problems were fixed by loading the *TradeEJBs.jar* and *TradeOther.jar* files with the same classloader. An alternative solution would have been to combine these two JAR files into one while leaving the client JAR untouched. Probably a better solution would be to divide *TradeOther.jar* into two parts; adding the access beans to *Trade2Client jar* and the remaining classes to *TradeEJBs.jar*.

A more granular solution would involve:

1. Splitting *TradeOther.jar* into three pieces
2. Repackaging the EJB JAR to include the files referenced by the EJBs interfaces
3. Creating an *AccessBeans.jar* containing the access beans,
4. Creating a *TradeRemaining.jar to* house the rest of the classes that were in *TradeOther.jar*

With this packaging, you should place the *AccessBeans.jar* in the Web Application classpath and *TradeRemaining.jar* into the application classloader's classpath. *TradeRemaining.jar* could be placed into the Dependent classpath, but this will inevitably cause headaches when we attempt to clone across nodes.

Separation of WebServer and Application Server

The illustration bellows shows the most prominent runtime configuration in use today, where you separate the Web Server from Application by the DMZ.

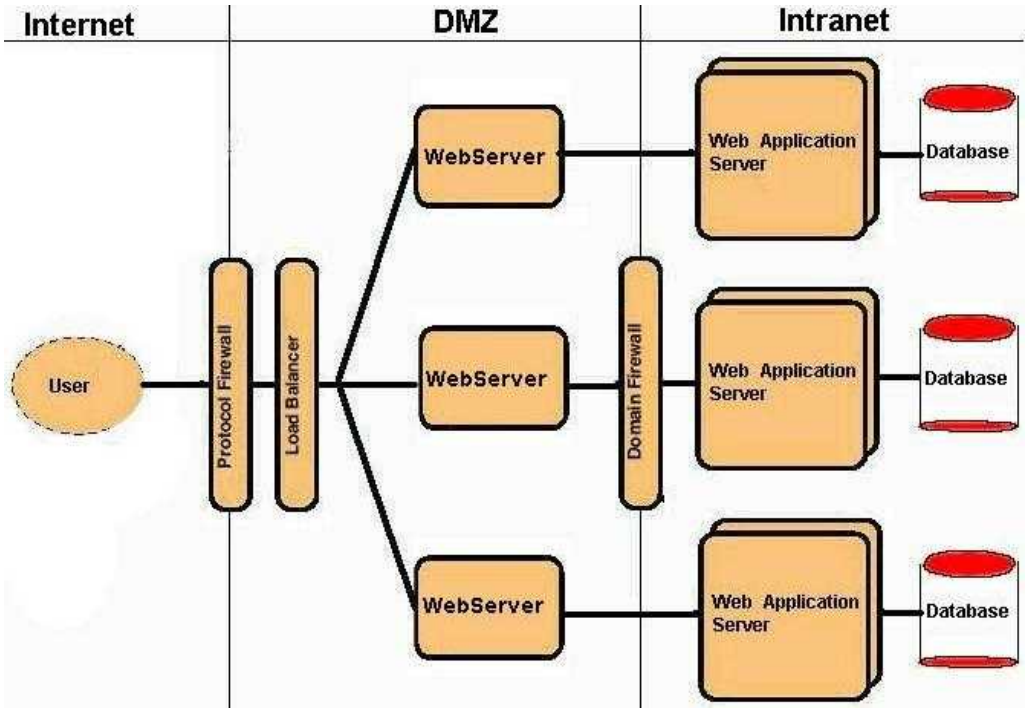

Figure 14

Example 2: Dependent classpath misuse
In this example we will go back to our three JAR model, placing *Trade2Clients.jar* into the Web Application Classpath, and deploying the EJBs within *TradeEJBs.jar* into the EJB Container, and *TradeOther.jar* on the Dependent classpath. Next we will clone the application onto another node.

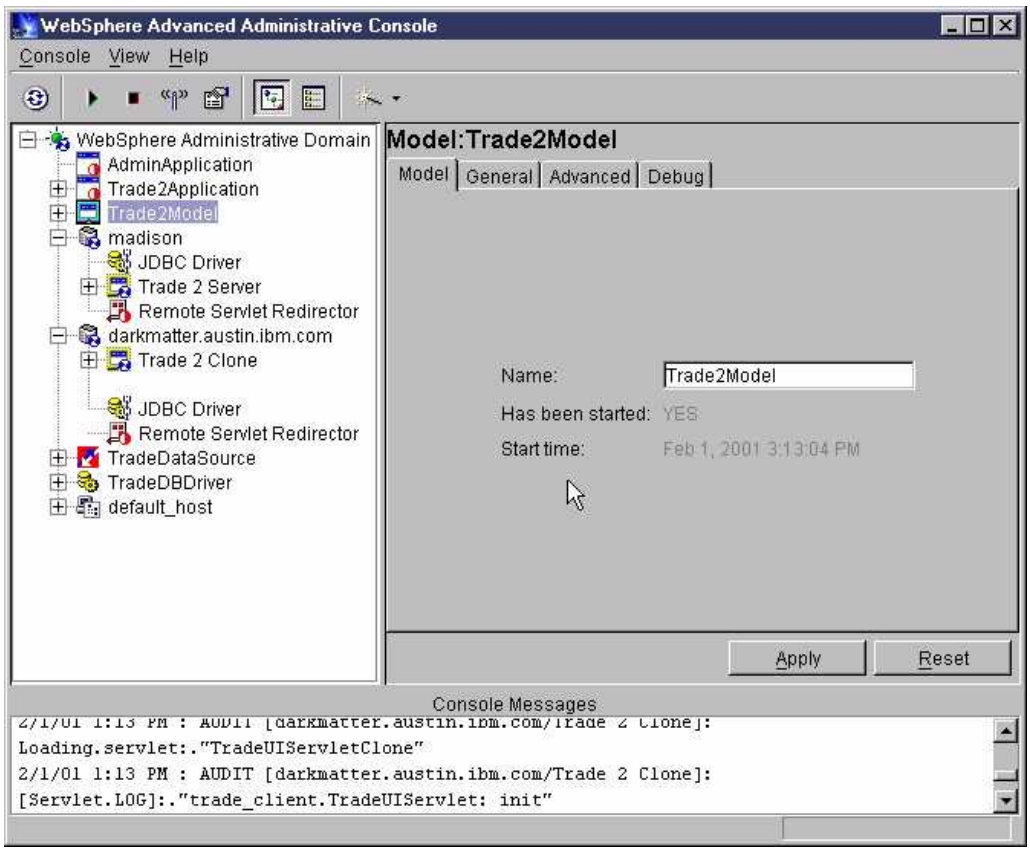

Figure 15

When we attempt to Login with this configuration things fall apart again, and we see the familiar blank screen:

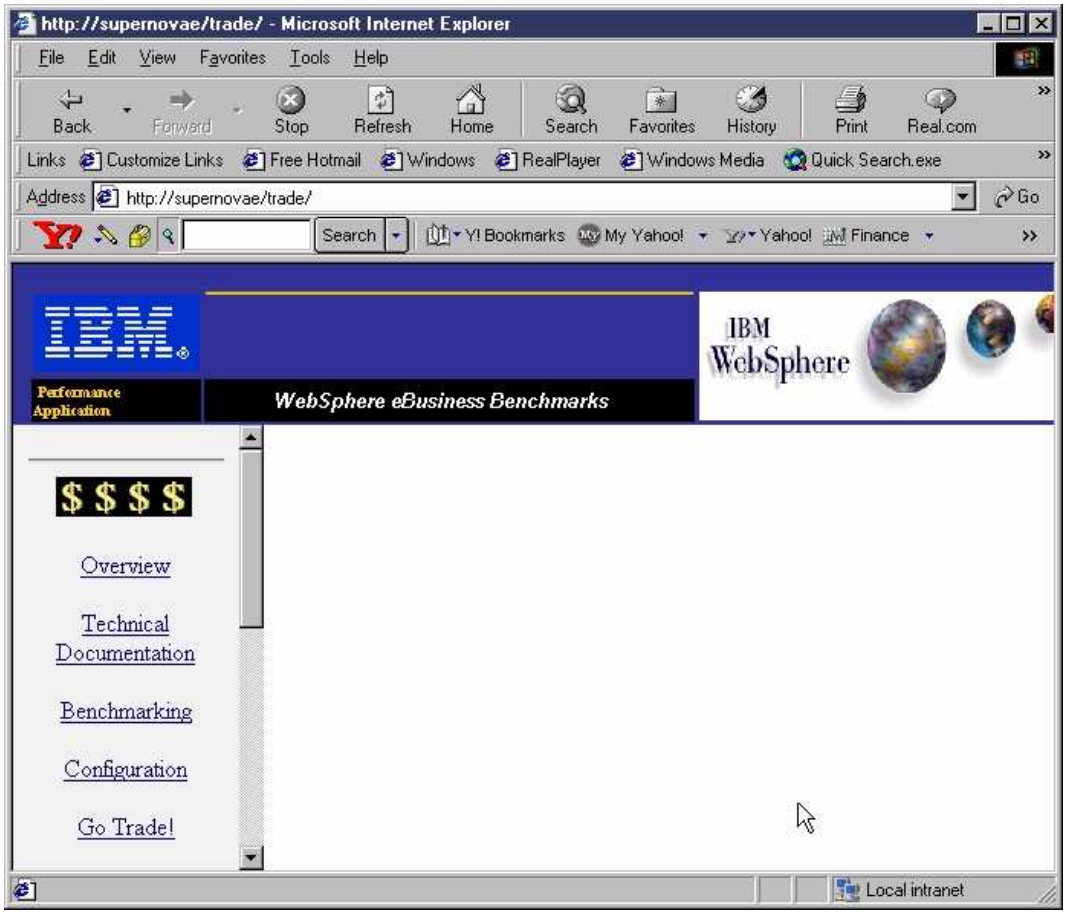

Figure 16

This time we get this exception:

TradeScenarioServlet: General Exception=java.lang.NoClassDefFoundError: trade/ProfileAccessBean action=l userID=uid:426
TradeServlet: General Exception: action=login userID=uid:10 Exception=java.lang.NoClassDefFoundError: trade/ProfileAccessBean

Unlike the previous examples, the exception is not thrown because a classloader failed to locate a class that existed somewhere lower in its hierarchy, but rather the class didn't exist in its hierarchy at all. Remember that the Dependent classpath is a node−specific field, so when the Application Server was cloned the Dependent classpath was not cloned along with it.

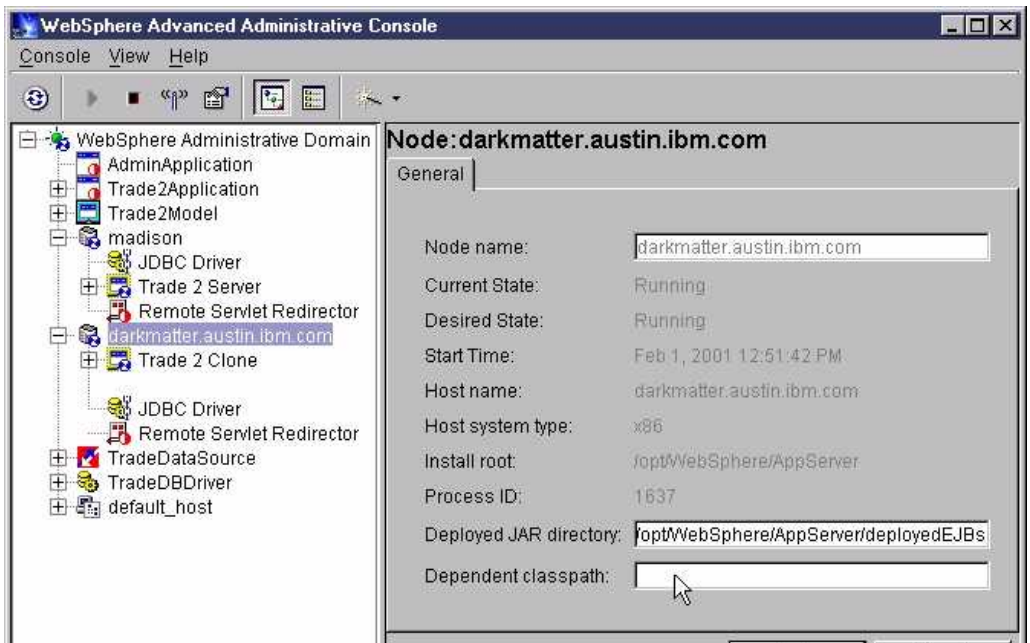

Figure 17

As a result, the Application Server couldn t load any of the classes residing in *TradeOther.jar*. Obviously you can easily fix this problem by adding *TradeOther.jar* into the Dependent classpath attribute of the new node, but this example demonstrates a more important point: Choosing the appropriate packaging structure for your application does matter. This problem could have been avoided by using the granular packaging suggested in the previous section.

Separation of Servlet Engine and EJB Container

With the increased security risks of the Internet, companies are demanding better safeguards for their business logic and application data. To facilitate this, many WebSphere customers are placing their mission critical data, EJBs and Databases, behind a second firewall.

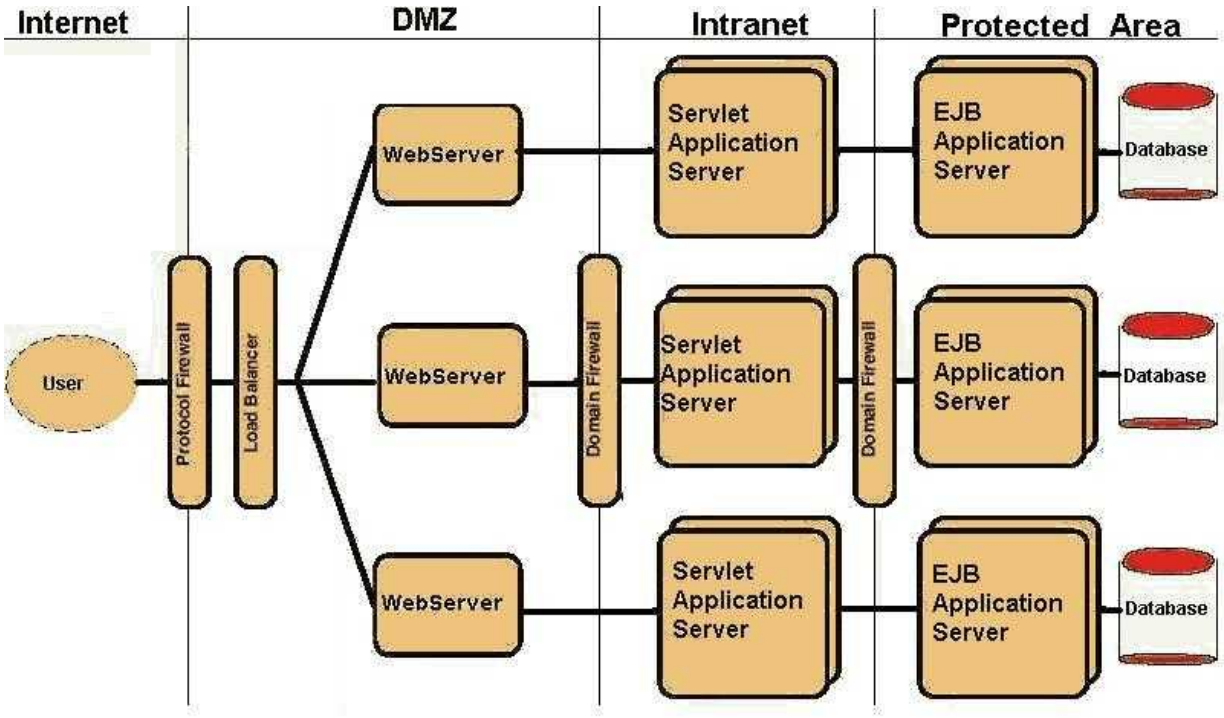

Figure 18

Example 3: EJB Client Jars

Each WebSphere Administrative Domain within this configuration consists of two nodes each containing an Application Server. One of the Application Servers contains the EJBs, while the other contains the servlets. To simplify things we use a two JAR file deployment of the Trade application consisting of the *Trade2Client.jar* file used in the previous examples, and a new jar file, *Trade2Beans35.jar* containing the files formally located in both *TradeEJB.jar* and *TradeOther.jar*. We ll place the *Trade2Client.jar* into the Web Application Classpath of the servlet Application Server, and deploy the EJBs within *TradeEJBs.jar* into the EJB Container on the EJB machine.

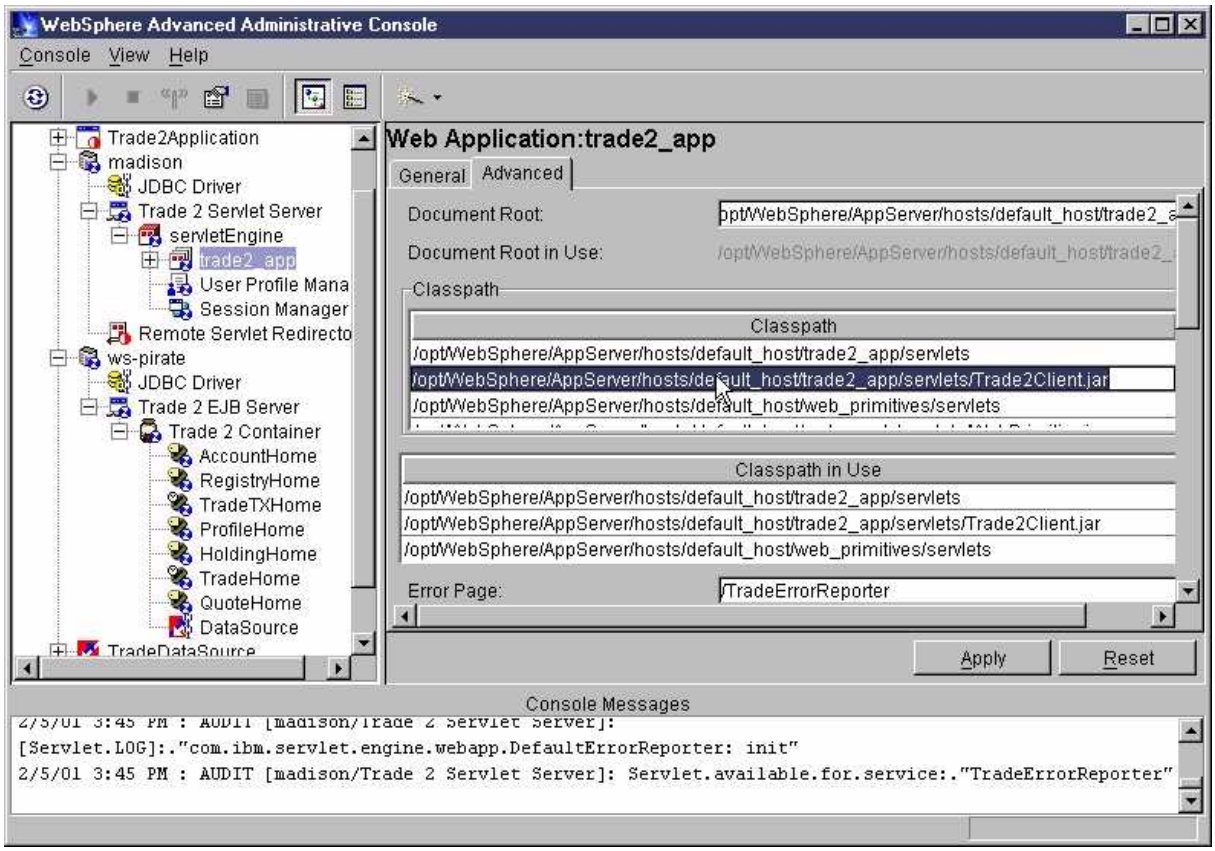

Figure 19

Again, our Login attempt results in a blank screen, accompanied by the *NoClassDef* exception in the logs of "Trade 2 Servlet Server".

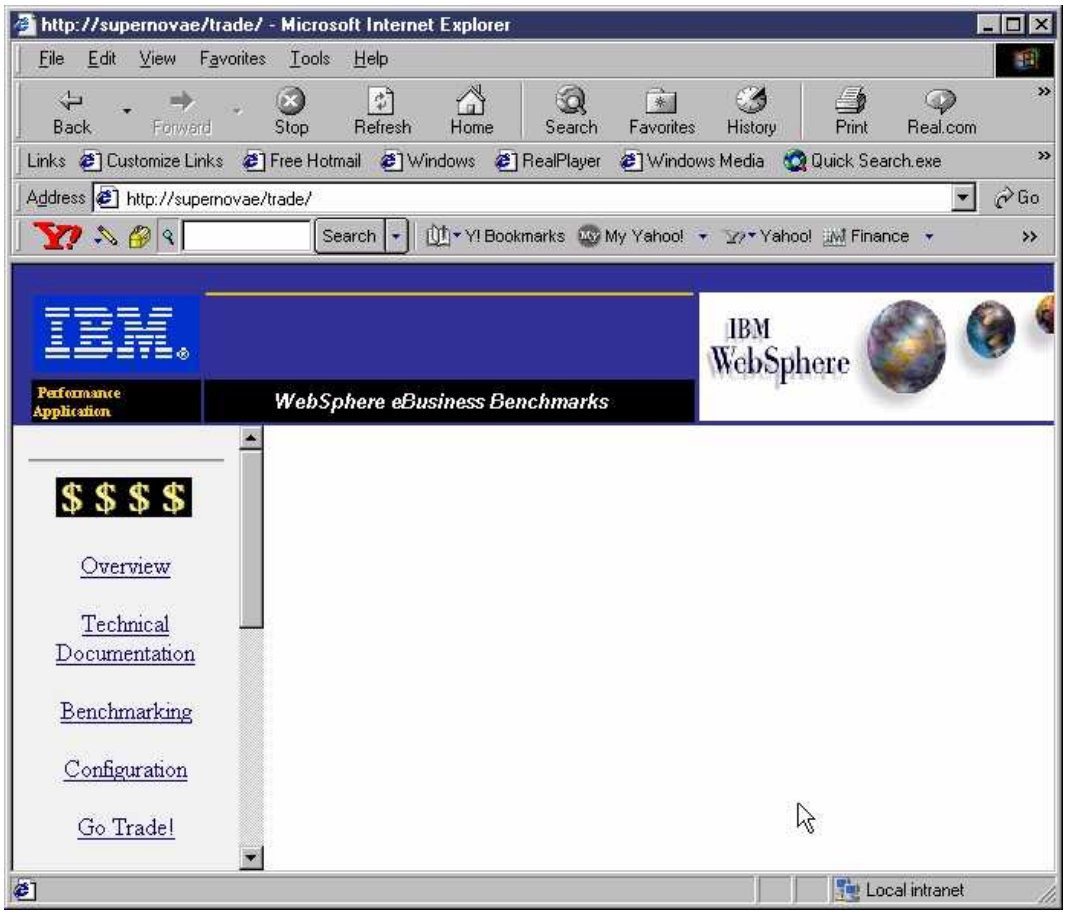

Figure 20

TradeAppServlet: General Exception: action=login userID=uid:17 Exception=java.lang.NoClassDefFoundError: trade/QuoteObject

Without Trade2Beans35.jar anywhere in its classpath, the *DynamicClassLoade*r of the trade2_app Web application cannot find any of the EJBs. There are two possible ways to resolve this matter. The first, the easy way, is to copy *Trade2Beans35.jar* to the servlet machine and place it into trade2_app's Web Application Classpath. The alternative solution would involve creating an EJB client JAR, and placing it into trade2_app's Web Application Classpath. The EJB client JAR would consist of EJB stubs, Home interfaces, access beans, and other classes used to interface with the EJBs. This may seem like a lot of extra work for little reward, but before quickly picking the easy solution remember that the initial reason for choosing the separate EJB Container and ServletEngine configuration was to protect important data and business logic. If we copy the entire EJB JAR file over to the "Intranet" area we are then placing the very code we are trying to protect in a less secure place. Management and security groups will likely agree that creating the EJB client JAR is a better solution.

Conclusion

All classloaders that run within the Java 2 framework are bound by certain rules and laws, and WebSphere's Classloaders are no exception. Once you master the basics of classloaders, you can extend them to specific classloaders, including those of WebSphere. The only unique aspect of WebSphere's classloaders is that they are tailored to perform within the Application Server environment.

Appendix A − Clarifying the Terminology

This document has delineated the three different classloaders provided via the Java launcher: bootstrap, extension, and application classloaders. Earlier releases, pre−Java 2, do not support the idea of extensions and only have one classloader that supports the function of both the bootstrap and application classloaders. When referring to this classloader the term *Primordial Classloader* is often used. This term is also used as a synonym for the bootstrap/null classloader when speaking in the context of the Java 2 Platform. The same sort of problem holds true for the Application Classloader , which is also referred to as the System Classloader in Java 2 terminology, but in pre−Java 2 releases the System classloader is analogous to the Primordial Classloader. Therefore whenever you see the terms Primordial and System classloader,it is very important to know their context.

Appendix B −Glossary

Administrative Server: The primary WebSphere java process, parent of all other WebSphere process, barring the Nanny process
AdminServer: See Administrative Server
Application classloader: A classloader reserved for user files. Also referred to as the System and Default classloader
Application Server: Loosely defined as a Java process that allows EJBs, Servlets, and other J2EE components to be run.
AppServer: See Application Server
Bootstrap classloader: A classloader responsible for bootstrapping the JVM with the core java classes, hence the name. In other documents it also referred to as the null or primordial classloader.
Default classloader: A classloader whose definition changes depending upon the context: (1) In the Java 2 Platform world it is a synonym for the application classloader (2) Prior to Java 2 it is synonymous with the System and Default classloaders, see Appendix A for more details.
J2EE: See Java 2 Platform Enterprise Edition
JAR: acronym for **J**ava **AR**chive.
Java 2 Platform Enterprise Edition: An architecture that consists of many different technologies with a comprehensive Application Programming Model for building server−side applications
Nanny process: A small java process that starts and restarts the Administrative Server if it fails.
Primordial classloader: A classloader whose definition changes depending upon the context: (1) In the Java 2 Platform world, it is a synonym for the bootstrap classloader (2) Prior to Java 2, the primordial classloader is synonymous with the System and Default classloaders, see Appendix A for more details.
System classloader: A classloader whose definition changes depending upon the context: (1) In the Java 2 Platform world, it is a synonym for the application classloader (2) Prior to Java 2, the System classloader is synonymous with the System and Default classloaders, see Appendix A for more details.
WebSphere Administrative Domain: Each Administrative Server must be configured to use an administrative database. Nodes that share the same administrative database are said to be in the same WebSphere Administrative Domain.

References

1. "Using the BootClasspath" by Ted Neward
2. "Understanding Class.forName" by Ted Neward
3. Server−Based Java Programming by Ted Neward
4. "Class loaders as a namespace mechanism" by Stuart Halloway
5. JavaTM 2 SDK, Standard Edition Documentation
6. Patterns for e−business: User−to−Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition
7. WebSphere's InfoCenter

IBM, VisualAge, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Windows and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java−based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

IBM copyright and trademark information