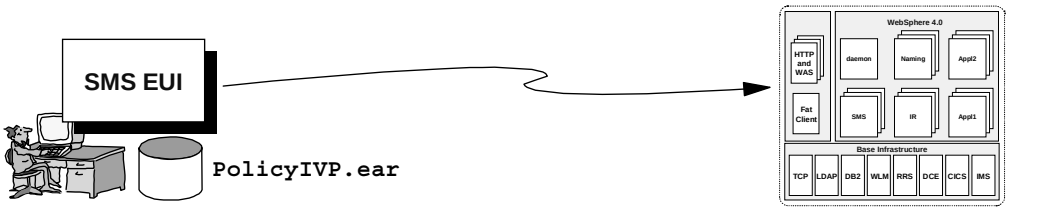
WebSphere Application Server
for z/OS and OS/390

Introduction to PolicyIVP Construction Lab

(This page intentionally left blank)

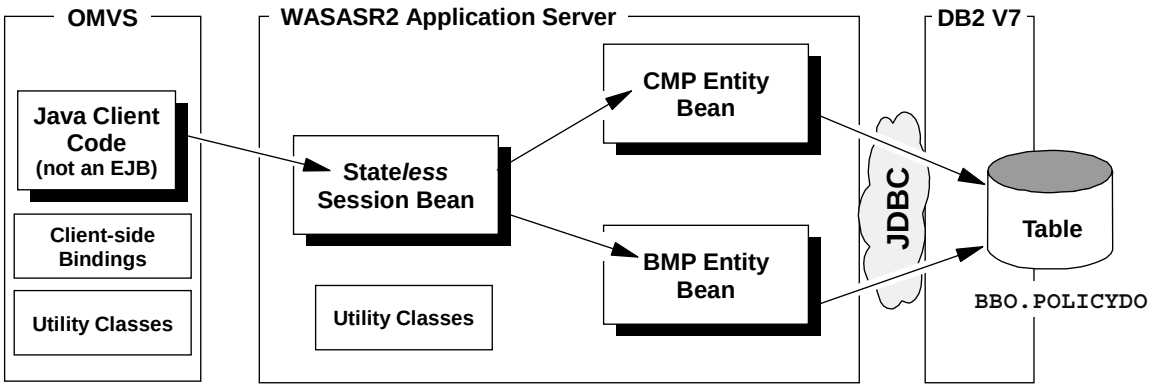
Overview of PolicyIVP Application

This is what you did in an earlier lab ...



WAS Runtime Environment

And this is the application you deployed ...



The "PolicyIVP" application is something shipped with the WAS 4.01 code and is used to verify the workings of your environment. In an earlier lab you were given the "EAR" file that contained the PolicyIVP application, and you used the SMS tool to deploy that application into the `WASASR2` server in the WAS 4.01 runtime.

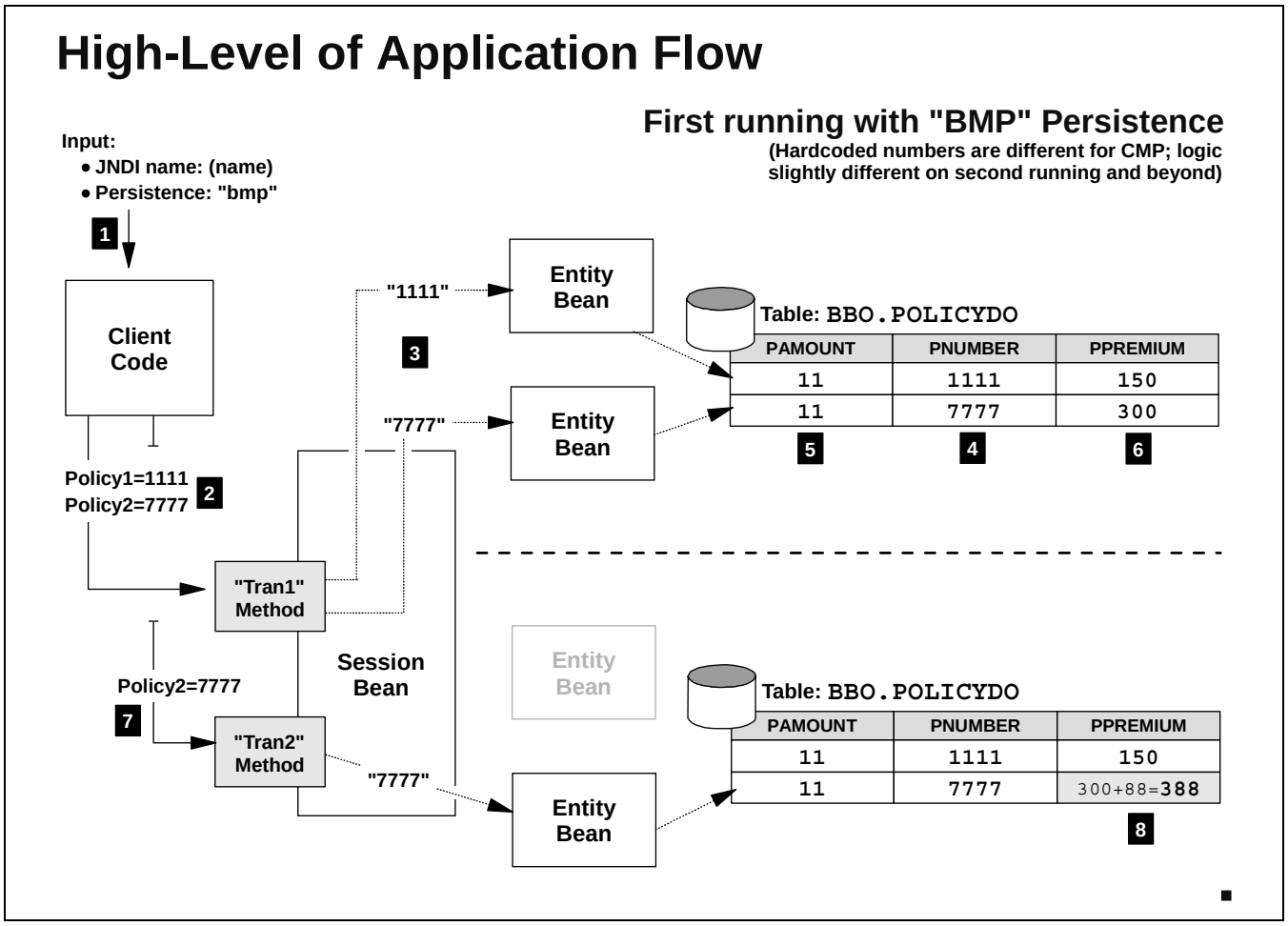
The PolicyIVP application is a relatively simply application consisting of one session bean and two entity beans:

- **Session Bean** -- this bean is *stateless*, meaning that every time a client connects and invokes the bean, the state of the bean is the same each time. Each client will get a copy of the bean that is identical to every other client's copy of the bean. Since there's no state, there no need to maintain any persistence for the session bean. The session bean has no connection to the database; it's whole function is life is to perform the very limited logic of this application and to drive one of the entity beans to store or retrieve data.
- **CMP Entity Bean** -- this bean is used to represent a row from the database table `BBO.POLICYDO`. Each row in that table represents a different "policy" (such as an insurance policy of some kind). The CMP is invoked by the session bean whenever DB access is required and the client has requested that the CMP bean be used (the client passes a parameter requesting which bean to use; you'll see that in a few charts). This bean relies on the services of the EJB container to perform the JDBC connection setup and SQL statement execution. Access to DB2 is via JDBC.

- **BMP Entity Bean** -- this bean is also used to represent a row from `BBO.POLICYDO`. The difference between this bean and the CMP bean is that the JDBC connection setup and SQL statement execution is handled inside the bean. That means the developer of this bean had to provide that code. Because the bean itself manages the persistence, the name "Bean Managed Persistence" is applied. Again, just like with CMP, this bean is driven when the client requests that BMP persistence be used. Access to DB2 is via JDBC.

In addition to those three beans, there are a few other pieces that come into play:

- **Client** -- the application needs a client to drive the session bean, and this code provides it. The client code is *not* part of the EAR file you deployed earlier. It is shipped in one of the WAS HFS directories, and the shell script you invoked drove the code that was in that directory.
- **Utilities** -- a handful of utility class files are bundled up into a JAR file called PolicyUtil.jar. These utility class files are required by the beans to function. They are also required by the client.
- **BBO.POLICYDO** -- this is a DB2 table we created as part of the setup for this class. It's a very simple table consisting of just three columns. The application uses this table to maintain persistence of data entered by the client.



This chart illustrates what happens when you run the PolicyIVP application. This example illustrates the first running when "bmp" persistence is chosen.

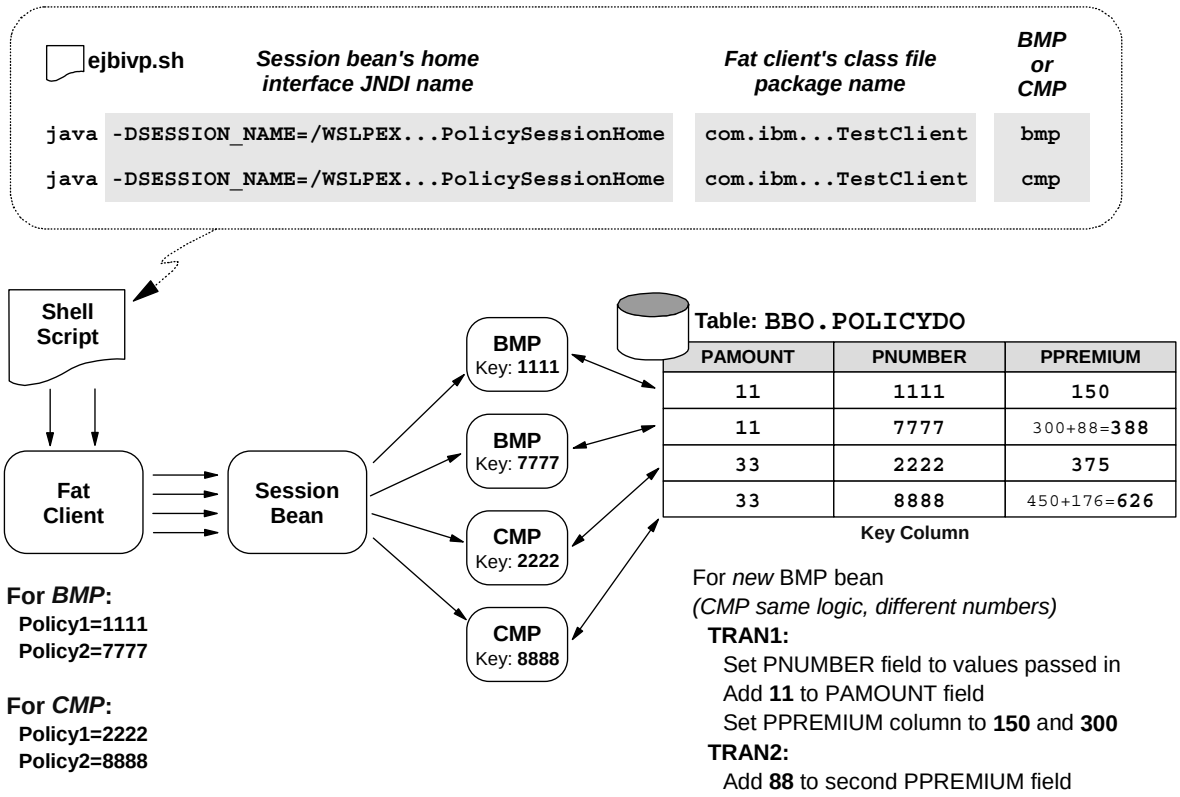
Note: The hard-coded values used when "cmp" is the parameter passed in are different from "bmp". The logic is the same, though. Also, if the bean already exists (the row exists in the table) then the application simply adds a fixed value to the `PAMOUNT` column and that's it.

1. Client is invoked and takes as input two values: the JNDI name of the Session bean's home interface, and either "bmp" or "cmp" as the choice for which type of persistence is requested.
2. In this example "bmp" is chosen. The client code then performs a JNDI lookup of the session bean and then drives the "tran1" method, passing the fixed values "1111" and "7777" in as Policy Number 1 and Policy Number 2.
3. The session bean receives the input and turns to drive the BMP entity beans. Since this is the first running it need to create them, which it does. The creation of the entity beans results in the row being created in the database table.
4. The `PNUMBER` column serves as the unique index to that table, and the values 1111 and 7777 are placed in the column for each row.
5. The fixed value "11" is added to whatever is presently in the `PAMOUNT` column. On first running the column is empty, so the result is a value of 11 in the column for each row.
6. The fixed value "150" is inserted into the `PPREMIUM` column for the first Policy, and "300" inserted into `PPREMIUM` for the second Policy. The "tran1" method is now done.

7. The client now drives "tran2." It passes in only the second Policy Number, which is 7777. The entity bean already exists, so the session bean doesn't need to instantiate it.
8. The logic for "tran2" simply has the session bean add the fixed value "88" to the value found in `PPREMIUM` column. The result in that column after "tran2" is run is 388.

That is a high-level illustration of the running of this application.

Example



Let's run through an example of a single invocation of the shell script:

- The shell script is used to invoke the "fat client" twice: once with a parameter of "bmp" (to invoke the "bean managed persistence" entity bean function), and once with a parameter of "cmp" (to invoke the "container managed persistence entity bean function). The format of the shell script's entry that invokes the fat client is shown in the chart above: first the JNDI home interface of the session bean to which the fat client will connect is provided as a Java parameter, then the fat client itself is invoked, and finally the parameter "bmp" or "cmp" is provided.
- The fat client's purpose is rather simple: connect to the session bean, tell the session bean whether BMP or CMP persistence is desired, pass in two "policy" numbers, and drive the session bean's "tran1" and "tran2" methods. The fat client has hard-coded policy numbers in it: 1111 and 7777 for BMP, 2222 and 8888 for CMP.

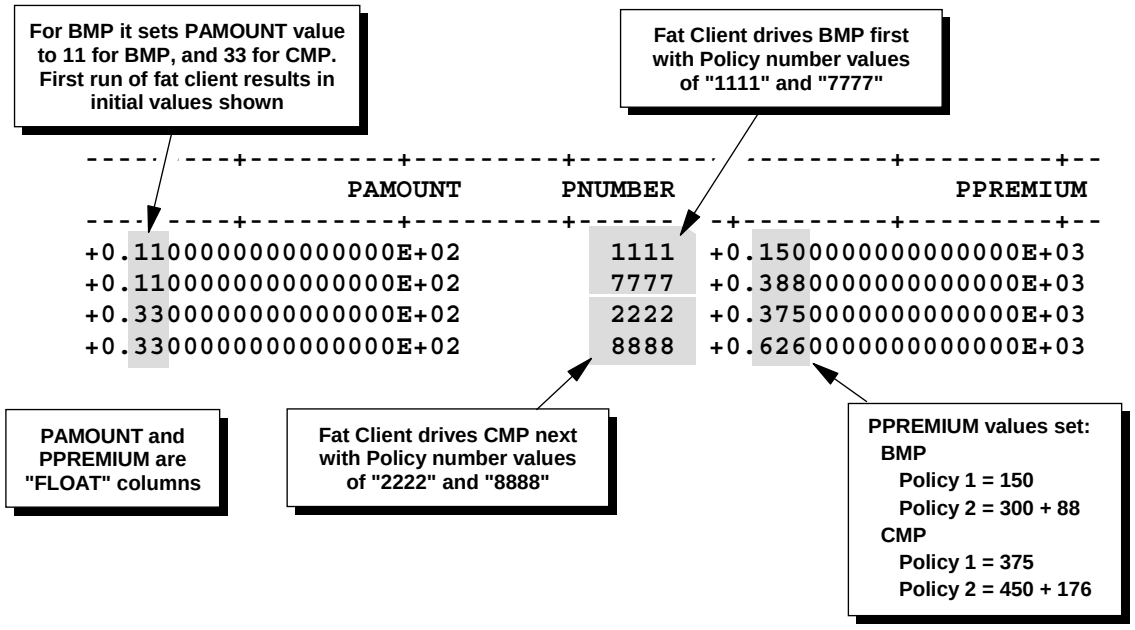
  So for the first invocation of the fat client, bean managed persistence (BMP) is requested. The fat client drives the "tran1" method of the session bean and passes in 1111 and 7777, and upon completion of that it drives "tran2" and passes in just the second number, 7777.
- On the first invocation with "bmp" requested, the session bean's "tran1" method is driven first with the numbers 1111 and 7777, and then with "tran2" just 7777 is passed in. For "tran1" the session bean first checks to see if the BMP bean with key 1111 and 7777 exists, and if not, it creates them. It sets the policy numbers of 1111 and 7777 as the entity bean's primary key class value (making them uniquely identifiable). For "tran1" it adds the value 11 to each bean's PAMOUNT value, then the values 150 to the first bean's PPREMIUM value and 300 to the second bean's PPREMIUM value. For "tran2" (remember, only the second value 7777 is passed in for "tran2"), the value 88 is added to the second bean's PPREMIUM. Thus ends the first invocation of the fat client.

The logic for the second invocation of the fat client -- with CMP persistence requested, is the same. The numbers are different, however. The policy numbers passed in are 2222 and 8888, and the amount added to the PAMOUNT column is 33 rather than 11. The amounts added to the PPREMIUM column is 375 and 450, and the value added in "tran2" is 176.

When all is said and done, the database table `BBO.POLICYDO` is left with the values shown in the chart.

The Database Table

What's in BBO.POLICYDO after one *successful* run of "fat client":

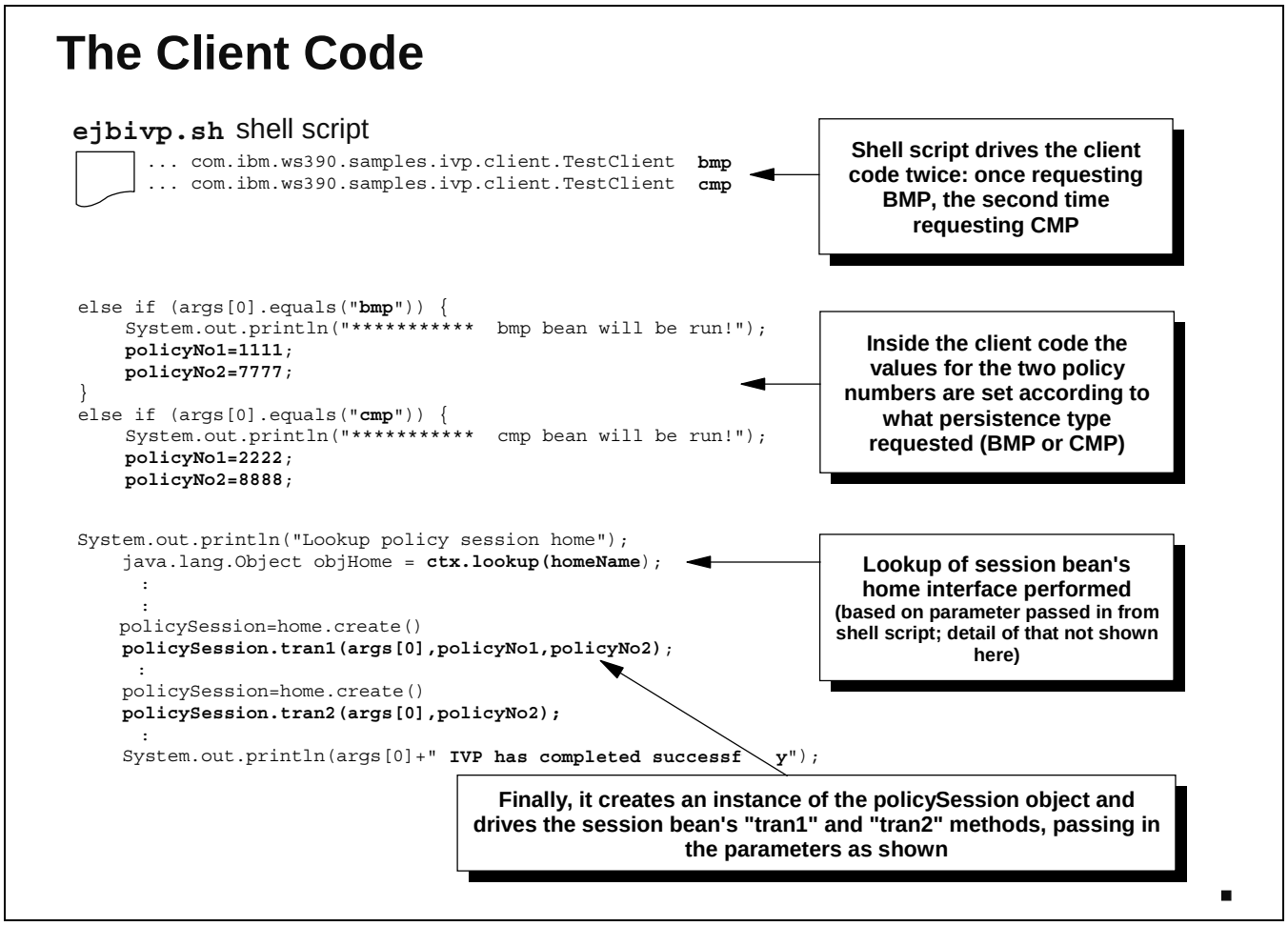


Message: contents of DB represents persistence. Two forms: CMP and BMP. Fat client used both to get four policy numbers along with payment amount and premium values into database

The previous two charts made mention of the database table `BBO.POLICYDO`. Here's what that table looks like after one run of the fat client code. The `PNUMBER` column contains the policy numbers passed in by client and is the key. Numbers in this column must be unique. The column is defined as `INTEGER`. The `PAMOUNT` (payment amount) and `PPREMIUM` (policy premium) columns are defined as `FLOAT` columns. The statements used to create this table are:

```
CREATE TABLESPACE POLICYTS
       IN BBOMDB01
       SEGSIZE 4
       LOCKSIZE ROW
       CLOSE NO;
CREATE TABLE  BBO.POLICYDO
(
 PAMOUNT  FLOAT  NOT NULL  ,
 PNUMBER  INTEGER  NOT NULL  ,
 PPREMIUM  FLOAT
 , PRIMARY KEY
 ( PNUMBER ))
 IN BBOMDB01.POLICYTS;
CREATE UNIQUE INDEX POLICY01
       ON BBO.POLICYDO (PNUMBER ASC)
       CLOSE NO;
```

The fat client drives the sessin bean twice: once for BMP and once for CMP. From this picture you can see the resulting information in the table. You can also see the evidence of the fixed numbers employed in the session bean. The purpose of showing you this is to make the application and its actions "real." The whole EJB thing might be somewhat mysterious, but a real DB2 table is something for which many will have familiarity.



Now let's take a look at the Java source for the "fat client." This code came straight out of VisualAge for Java. Not all of the client's source code is represented here; this is just a few snippets.

- You start with what the shell script will send when it invokes the session bean. There are two lines in that shell script that invoke the bean, so it's actually driven twice. The first time the BMP bean is requested and the second time the CMP bean is requested.
- In the client's source you'll see where the two policy numbers are being set based on hard-coded numbers. For BMP the values are `1111` and `7777`; for CMP the values are `2222` and `8888`.
- A little further down the client does a lookup for the session bean's home interface based on a variable called "homeName." That variable is set based on the string passed in from the shell script. That detail is not shown here, but the point is the client must first find the session bean's home interface before it can create an instance of the bean and drive it.
- Once the session bean is "fluffed up" (an instance created), the bean's "tran1" method is driven with the choice of BMP or CMP passed in as `args[0]`, and the two policy numbers set earlier passed in as well.
- After "tran1" is driven, "tran2" is driven, this time with the CMP/BMP choice passed in and just the second policy number.

If all goes well, you get the message indicating things ran successfully.

The Session Bean

This example shows the session bean driving the BMP entity bean. Session bean logic to drive a CMP bean identical. (This code taken from "tran1" method of "PolicySessionBean" EJB)

```
if (beanType.equals("bmp")) {
  System.out.println("Running a bmp bean");
  com.ibm.ws390.samples.ivp.ejb.PolicyBMPHome home= null;
  com.ibm.ws390.samples.ivp.ejb.PolicyBMPKey key1 = null, key2 = null
  com.ibm.ws390.samples.ivp.ejb.PolicyBMP policyBean1 = null, policyB
  System.out.println("Lookup bmp policy home");
  try {
    objHome = ctx.lookup("java:comp/env/ejb/ivp.policybmp");
  }

System.out.println("Finding bean1 - " + ((new Long(policyNo1)).toString()));
  key1 = new com.ibm.ws390.samples.ivp.ejb.PolicyBMPKey(policyNo1);
  try {
    policyBean1 = home.findByPrimaryKey(key1);
  }
  catch (javax.ejb.ObjectNotFoundException e) {
  try {
    System.out.println("Bean1 does not already exist, cre
    policyBean1 = home.create(policyNo1);
  }

System.out.println("Adding 11 to the amount for both beans ");
  policyBean1.setAmount(policyBean1.getAmount()+11);
  policyBean2.setAmount(policyBean2.getAmount()+11);
  System.out.println("Setting premium for bean1 to 150");
  policyBean1.setPremium(150);
  System.out.println("Setting premium for bean2 to 300");
  policyBean2.setPremium(300);
```

Using java:comp to look up the BMP bean's home interface. Name shown here must match what you set in AAT

Checking to see if a bean with the same Policy number exists. If not, it creates one.

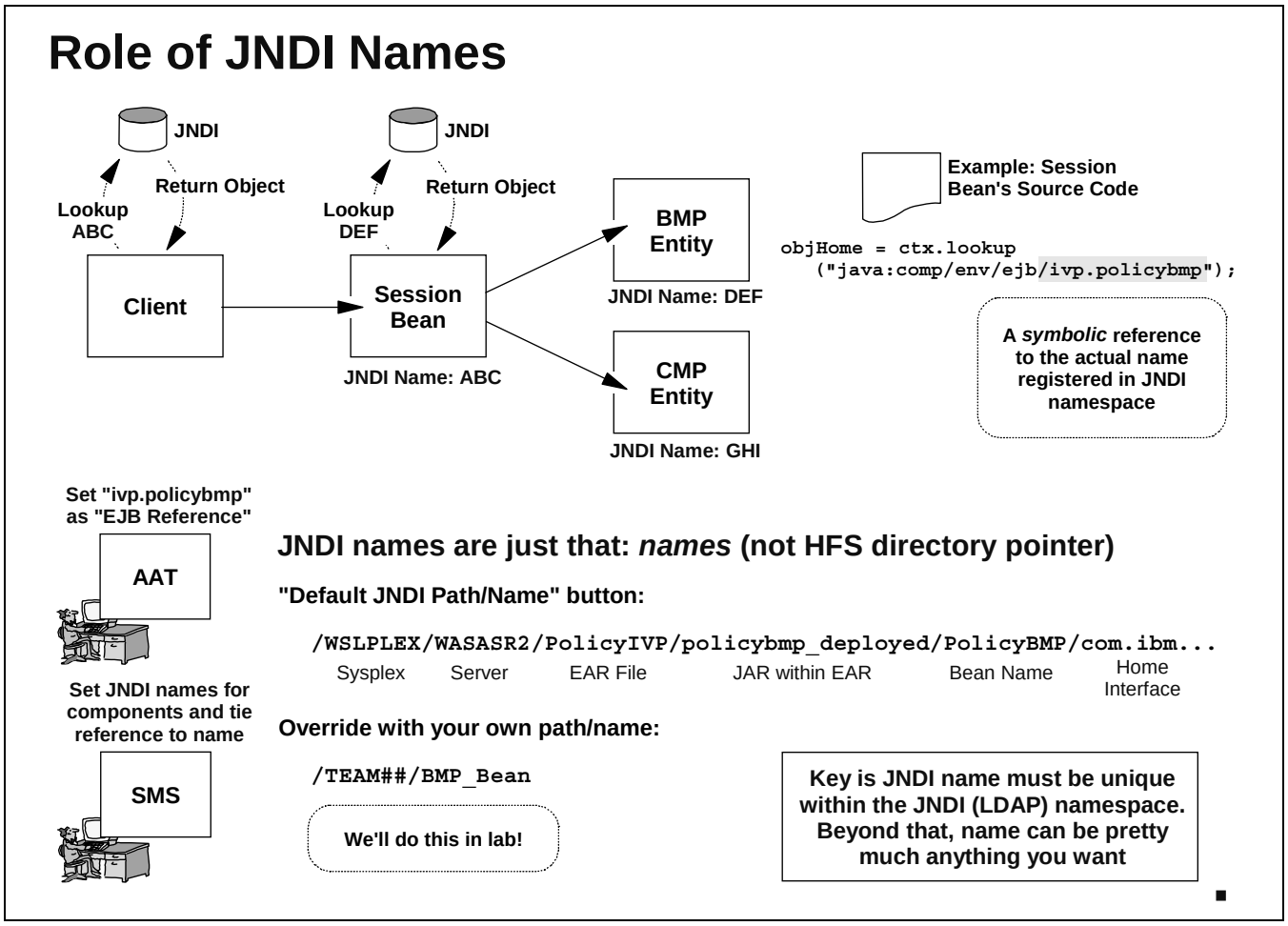
Once the bean has been located (or created), the values are updated. This results in updates to BBO.POLICYDO table.

Here's what the session bean contains. Again, this is just a snippet from one method of the PolicySessionBean:

- The client passes in the BMP/CMP parameter and this drives the session bean down a certain code path. This example is showing the BMP path. Then the session does a `java:comp` lookup on the entity bean's home interface. This is worthy of note because the assembly of the application (using AAT) must set the "reference name" values equal to what's being called out in the code itself. You'll get a chance to see that in the lab later.
- Once the home interface has been located, the bean checks to see if the request is for a new bean or one that already exists. If it doesn't exist, the "create" method is driven to create a new instance of the bean with a key value equal to the policy number passed in. (This example shows only one policy number check, but in reality the session bean does this twice: one for each policy number specified by the client).
- Finally, the session adds the fixed value 11 to the `PAMOUNT` field for each new record (in the case of a BMP; 33 for a CMP) and then sets the `PPREMIUM` value to 150 for the first bean and 300 for the second (again, in the case of a BMP; the CMP has different static values).

The setting of the values in the DB2 table is *not* done by the session bean. The session bean invokes the methods of the BMP or CMP *entity* bean to do that.

Before we look at the entity beans, let's look at this "JNDI lookup" thing.



When one component of a J2EE application needs to make contact with another component, it first needs access to the target component's "home interface" object. In the picture above, the client will need access to the home interface of the session bean, and the session bean will need access to the home interface of the BMP bean and the CMP bean. It gets access to the home interface by doing a JNDI lookup of the bean's interface. A lookup results in a copy the object being returned to the requesting component.

To do a lookup, the component doing the lookup needs to know the JNDI name of the target component. Every EJB component will have a JNDI name associated with it, and JNDI name is registered in the "JNDI namespace" (which for WAS/390 is implemented using LDAP). Your code *could* have the JNDI names hard-coded, but that's not a good practice. Hard-coded references limit the portability of the code. A better practice is to use a symbolic reference in the code, and that's what's used in PolicyIVP with the "java:comp" lookup. In this example, the source code for the session bean looks up the BMP bean with a symbolic reference of "ivp.policybmp."

JNDI doesn't have a clue about "ivp.policybmp" ... it's a symbolic used in the source code only; it is *not* registered in JNDI. So what's the connection? When you assemble an application using AAT you create an "EJB Reference" naming the symbolic used in the code and connecting that to the class name of the target bean's home interface. But that's only half the story. When you deploy the application using the SMS EUI, you specify a JNDI name for each component and tie the symbolic to the JNDI being referenced. When the application is deployed the components are registered into the JNDI namespace with the JNDI names given, and the application can now use its symbolic lookup and have it result in a lookup using the real JNDI names.

Is there any magic to the JNDI names you set? Not really. They're just names. The key is they must be unique within the JNDI namespace. One way to set to the JNDI names is to make use of the

"Default JNDI Path and Name" button of the SMS EUI. That will set a JNDI name for you, and it will insure uniqueness by structuring a JNDI name with the various components shown in the chart. That's a perfectly acceptable thing to do, and many people will do just that.

But using the default button is not required, and the path and name does not need to be as complex as is generated with that button. In the upcoming lab we'll have you set the JNDI path as TEAM## (where ## is your team number) and the JNDI name as the name of the bean (BMP_Bean in this case). You could make it ever shorter than that: "X". The trouble with really short names is it makes it hard to insure uniqueness. Typically the JNDI path/name pair will be something longer than a few characters.

Note: One shortcoming of setting your own names -- as opposed to using the default button -- is that the SMS tool has no way of knowing if the JNDI name you set is already registered in LDAP. If you provide a name that's already registered, the "naming registration" phase of deployment will fail. You can then change the name and try again, but that takes time. Consistent use of the "default button" avoids this problem.

The BMP Bean

From `getConnection()` method of PolicyBMPBean:

Using java:comp to look up the J2EE JDBC resource. Name shown here must match what you set in AAT

```
try {
   if (ds  == null) {
     ctx = new InitialContext();
     try {
     ds = (javax.sql.DataSource) ctx.lookup("java:comp/env/jdbc/policy");
     }
```

From `ejbStore()` method of PolicyBMPBean:

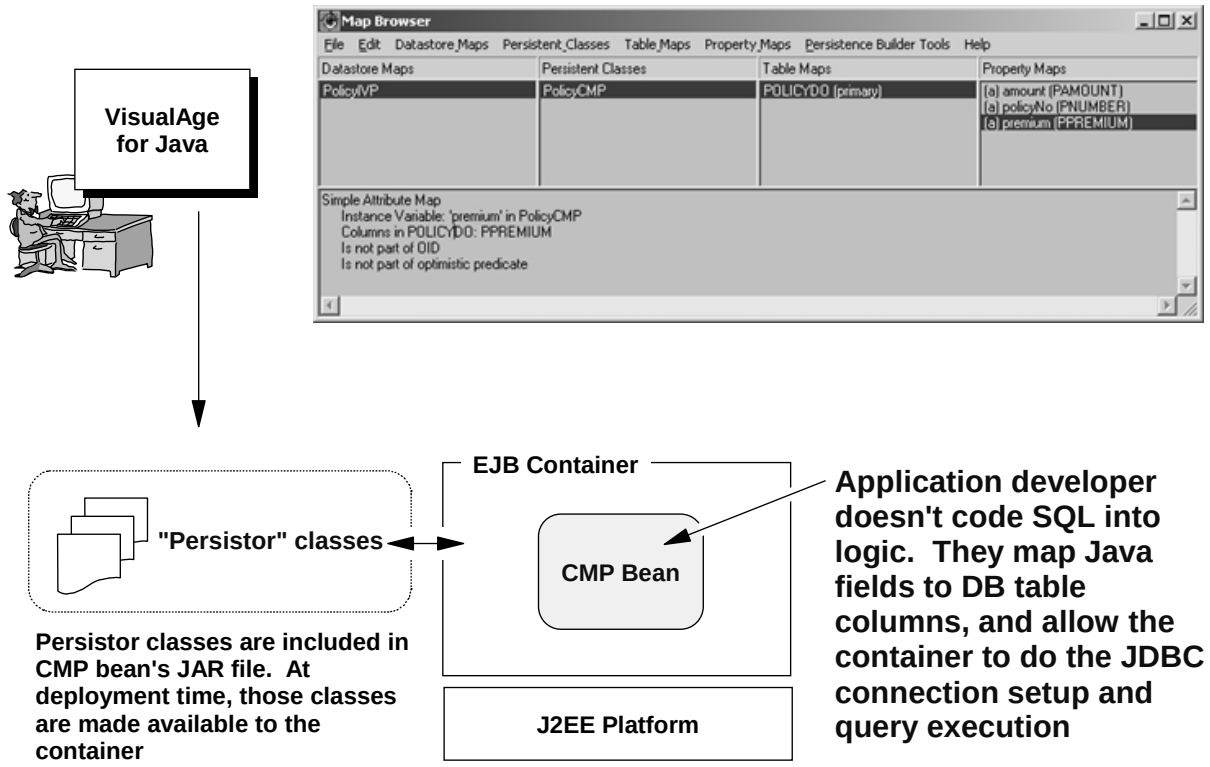
```
conn = getConnection();
pstmt = conn.prepareStatement("update BBO.POLICYDO set PAMOUNT = ?,
                                        PPREMIUM = ? where PNUMBER = ?");
pstmt.setDouble(1, amount);
pstmt.setDouble(2, premium);
pstmt.setInt(3, policyNo);
pstmt.executeUpdate();
```

Construction of SQL query handled in bean. Thus, *Bean Managed Persistence* (BMP)

Here's what the BMP bean is doing (again, just a snippet of the total source code):

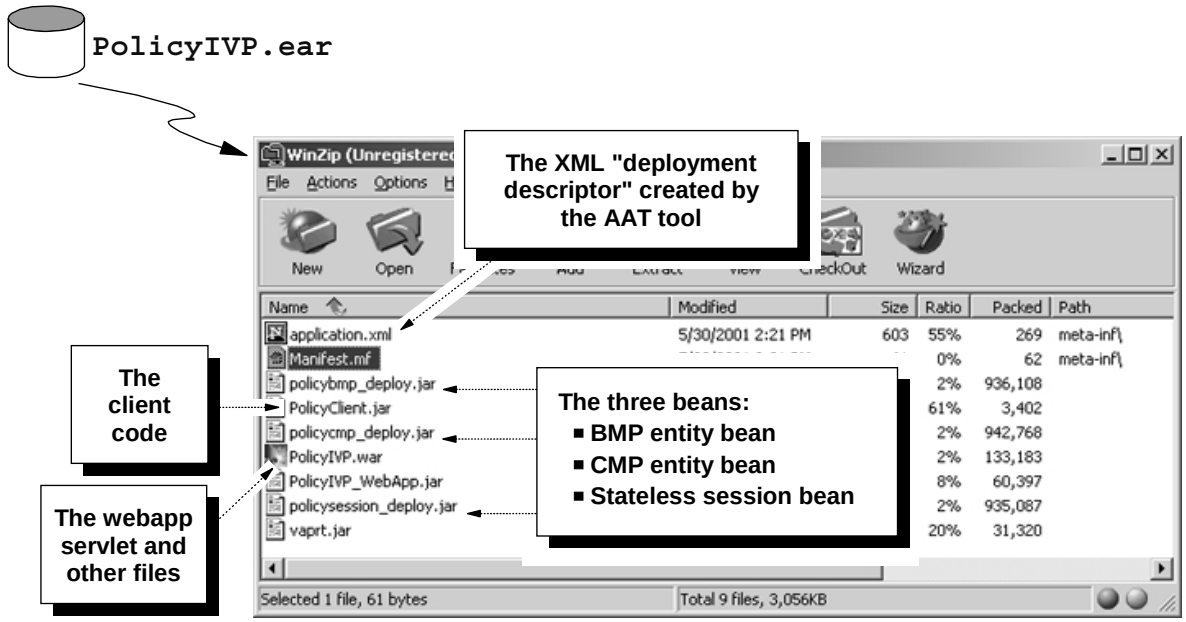
- In the case of a BMP, the bean itself is responsible for the management of the persistence. Therefore, one of the first things it must do is find the datasource to which it'll connect. In this case the datasource is named `jdbc/policy`, and a `java:comp` lookup is used to find it. Here's another case where the settings you provide in the AAT tool must match what's called out in the actual code. In the lab that follows you'll set a value for the "reference name" for the BMP resource, and there you'll name `jdbc/policy`. Set it different and the bean won't find its datasource.
- The `ejbStore()` method is illustrating how an update to the table is being made. First a connection is established, then the SQL is constructed, the values for the SQL update are set and then the statement is executed. This is pretty much just like what you'd do in Java code outside the world of EJBs. Because this is a BMP, the management of the connection and the SQL statements is the responsibility of the bean itself, which means the developer must make it all happen.

The CMP Bean



The CMP bean is different. The developer coding up the EJB using VisualAge for Java uses database mapping tools within the VAJ product to map Java field names to columns in the database. Information about the database columns is provided (type, whether its a key, etc.). VAJ will then generate "persistor" classes -- compiled Java code that will perform the connection work to the datastore. These persistor classes are bundled up in the CMP beans JAR file, but at deployment time those classes are made available to the EJB container in which the CMP bean is deployed. The bean doesn't do "get connection" and "SQL statement preparation" work. It simply calls a method related to the database field it wants to affect, and the underlying container does the work, based on what's in the persistor classes generated by VAJ.

Contents of PolicyIVP.ear File



Where did all these files come from? How did the EAR file get generated?

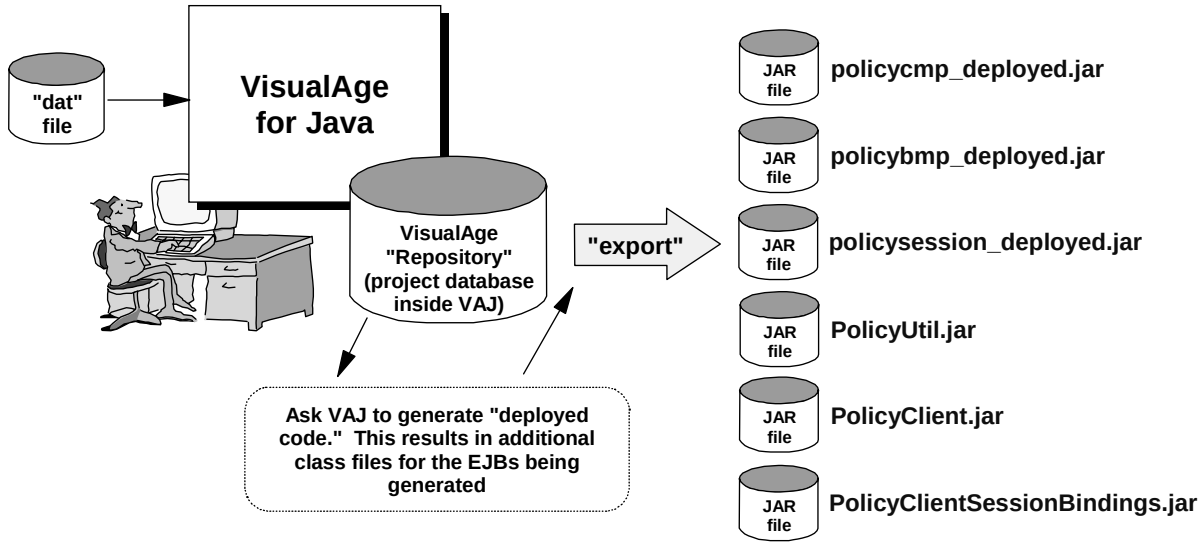
Lab to follow will have you create the EAR

In the earlier lab you were simply given an EAR file. You may have invoked WinZIP® against the EAR file and seen what's illustrated above. The EAR file is a standardized packaging format for EJBs. It is a "zip" format file with certain requirements. One such requirement is for a "deployment descriptor" to be placed in a meta-inf\ directory off the base of the structure. That deployment descriptor is an XML file, and it contains information about the application contained in the EAR file. The XML file is generated by the AAT tool at the time the EAR is created.

You see the two entity beans the session beans in their JAR file format. There is also a "webapp" in this EAR file, and it contains the files for the servlet web application you'll use in a later lab. The client code is also in the EAR, but when deployed into the J2EE Server it isn't used for anything. To use the "fat client" you must extract that JAR out of the EAR and copy it into an HFS directory. The client JAR is in this EAR just so the whole application is contained in one file.

Where did these files come from? A developer generated them up and packaged them into an EAR file. In the lab that follows you will create this EAR file based on a set of JAR files and WAR file supplied to you ... it will be just as if an application developer passed his work to you for assembly and deployment.

From VisualAge to JAR Files

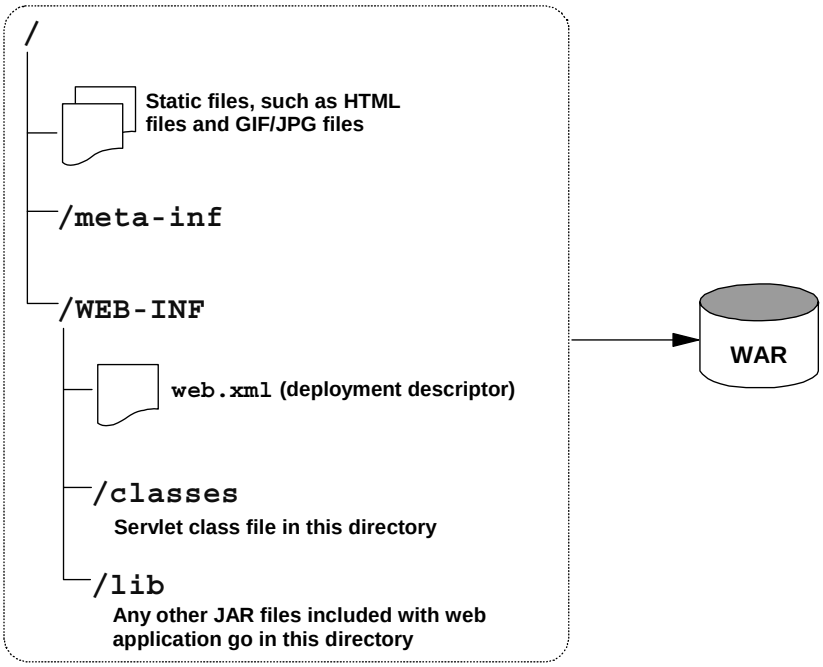


JAR files are created by "exporting" code components from VisualAge for Java. This is something application developers will do.

The first step in this process is to bring up the source code in VisualAge for Java. The way projects are transported between copies of VAJ is in the form of a "dat" file, which is a file format recognized by VAJ as mapping to its "repository." The PolicyIVP application's "dat" file is provided with the WAS product, and is available for downloading and import into a copy of VAJ.

For this lab, we've done this step for you.

The WAR File



WAR files are zip-like files that contain web applications. Tools like WSAD are used to develop webapps and create WAR files. Later unit describes configuring webapp support for WAS 4.01.

■

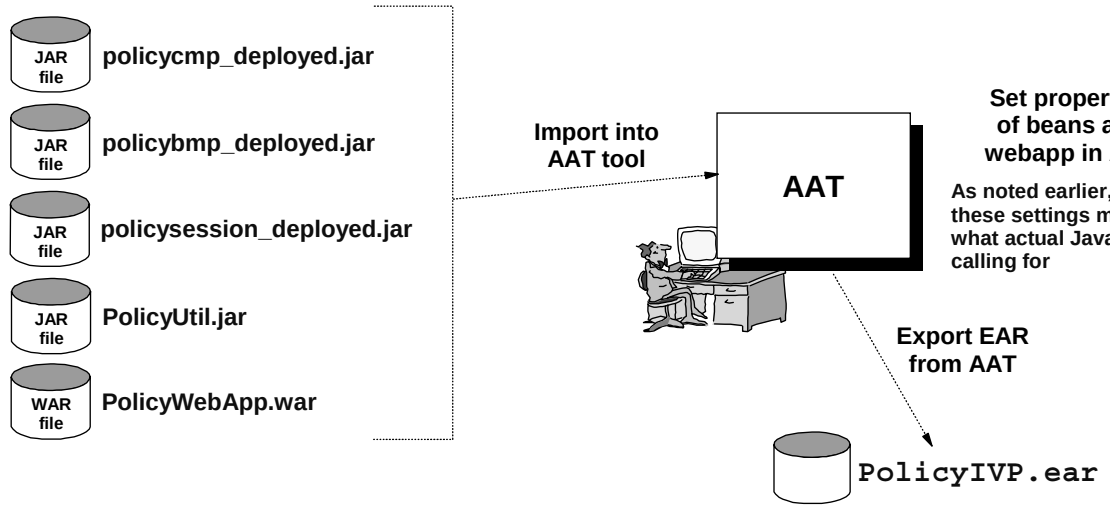
There is also a "WAR" file inside the EAR file. WAR files are Web ARchive files, and they are another form of standardized zip-file format to contain web applications. Web applications are not EJBs, but they are Java code in the form of servlets. Web applications also have static components like HTML and GIF/JPG image files.

The WAR file also has an XML deployment descriptor. For this class we will supply you with that XML file for download and inclusion in your WAR file.

Tools such as WebSphere Studio or the newer WebSphere Studio Application Developer (WSAD) have the ability to create Webapps and generate WAR files.

This class has a whole section devoted to the configuration of "web containers" and the deployment of web applications. For now what you will do is receive a WAR file that we extracted from the EAR file and include it along with all your JAR files into the new EAR.

From JAR/WAR Files to EAR File



The EAR file you will export will be essentially equivalent to the EAR file we gave you in earlier lab.

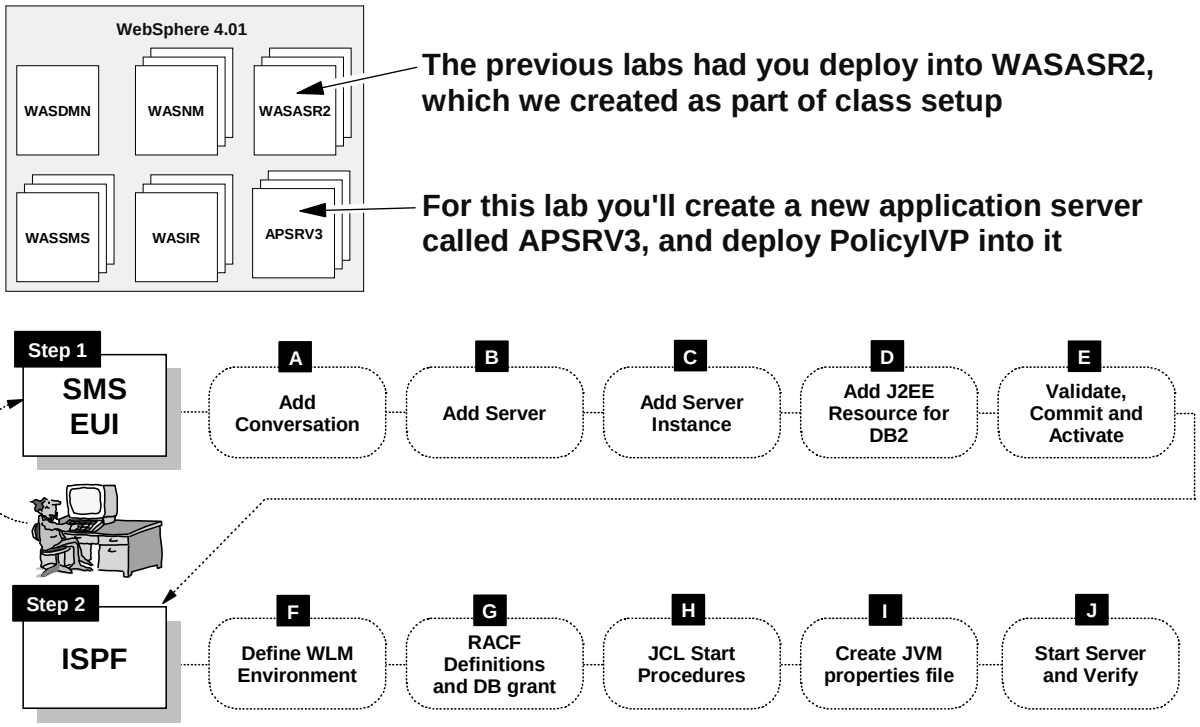
Now you know where the EAR came from: someone in Poughkeepsie coded up the PolicyIVP application in VAJ, exported to JAR files and used AAT to construct the EAR.

In the upcoming lab you will construct the EAR file using AAT.

Next comes the act of packaging the various JARs and WARs into an EAR file. You do that by using the AAT tool that comes with the WAS 4.0 product. The activities related to creating an EAR file are more than just zipping up the file. You set various properties for the application in the panels of the AAT tool. Those properties are then placed in the XML deployment descriptor housed in the EAR file. Some of those properties must match the values from the Java source itself. That's why the packaging of the EAR file will involve coordination with the developer.

The EAR you'll construct in this lab will be nearly (but not exactly) identical to the sample that ships with the product. The person who created the EAR that ships with the product will have gone through the steps you will do in this lab to generate the EAR.

Creating a New Application Server



Why? It's good practice. You could have deployed into WASASR2 again, but the creation of the server provides you good hands-on for that process.

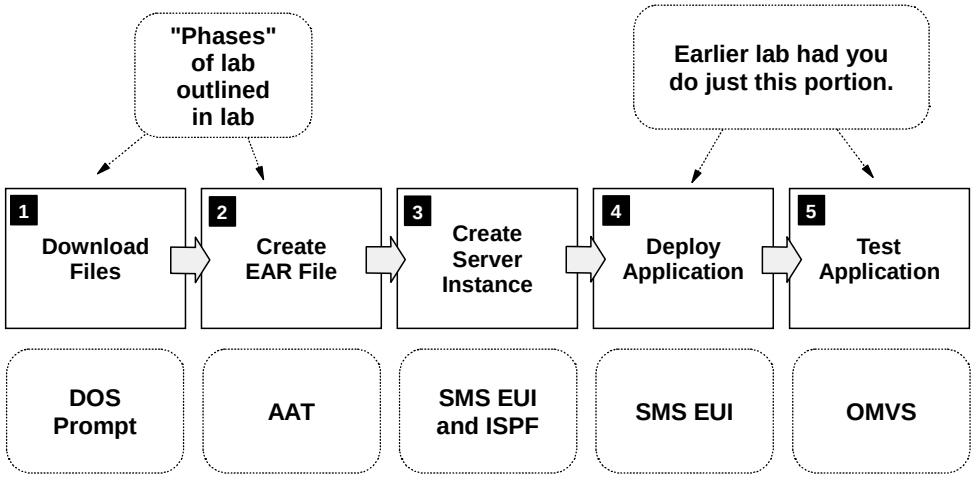
■

Once the EAR is packaged the next step is to deploy the application into an application server of your WAS runtime. Earlier you deployed into the WASASR2 server region, but for this lab we'll have you create another server region and deploy your code into there. We're doing that to give you the experience of creating the server regions and to experience the non-GUI tasks (in other words, the traditional TSO "green screen" tasks) associated with this activity.

We'll have you create the server region first and verify it comes up okay. Then you'll deploy into it. You could do the two at the same time, but we're making it two separate phases of the lab to keep things as clear as possible.

The steps you'll walk through in doing this are shown above. It'll involve the use of both the SMS End User Interface as well as a TSO session.

Lab Overview

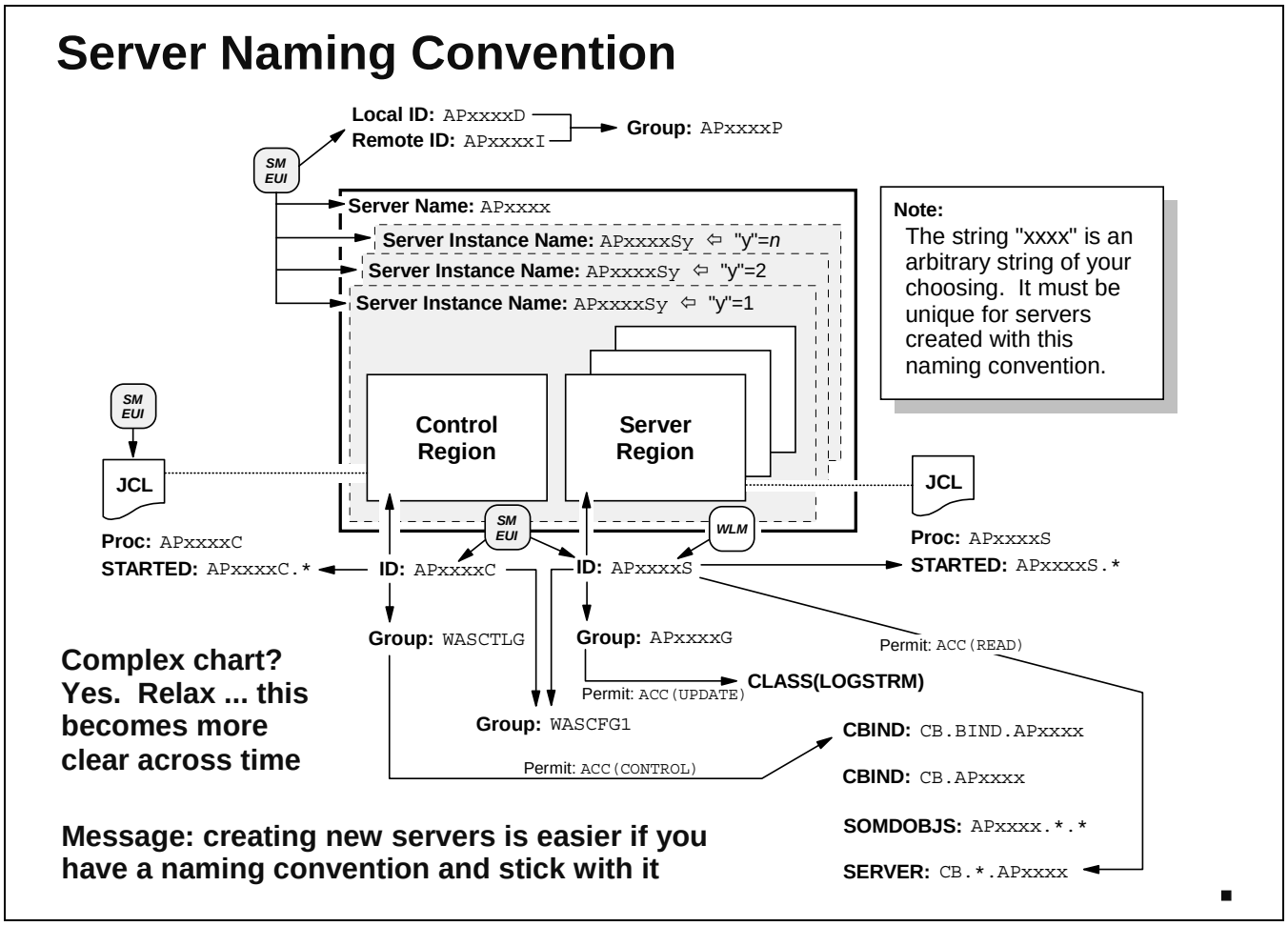


So here is an overview of the lab you'll do now. It has five "phases" as shown above. The lab handout has each phase clearly labeled, and you'll walk through the process from front to back. In the earlier lab when you deployed the supplied sample EAR file you were doing phases 4 and 5. Here you'll do the other four that preceded it.

The dotted boxes under each phase indicate what environments you'll use to do the tasks.

Good luck!

End of Presentation
(reference page follows)



This chart is provided just as a reference sheet. It illustrates the relationship between the various components of the new J2EE application server you will create and all the names associated with it.

The new server region you will create will be called `APSRV3`. It's control region will be called `APSRV3C`, and its server region will be called `APSRV3S`. You'll create one "instance" and it'll be called `APSRV3S1`. Do you see the connection between all those names? They are all based on the naming convention of `APxxxx`, where `AP` is a fixed two letter prefix (standing for APplication server), and `xxxx` being a four character string of your choosing.

Is there magic in this naming convention? No. But having a naming convention will help you considerably when it comes time to perform the various WLM, JCL and RACF work necessary to set up the environment. With a naming convention in place, these other tasks can be done in a more automated fashion. Without a naming convention, you would have to rely on your brain to keep the mappings straight in your mind. You have enough things stuffed in there; there's no reason to add to the confusion.

The picture above looks like a complex chart. It is initially. But as time goes by you will come to see how the naming convention of `APxxxx` will tie all the pieces of the puzzle together.

End of Document