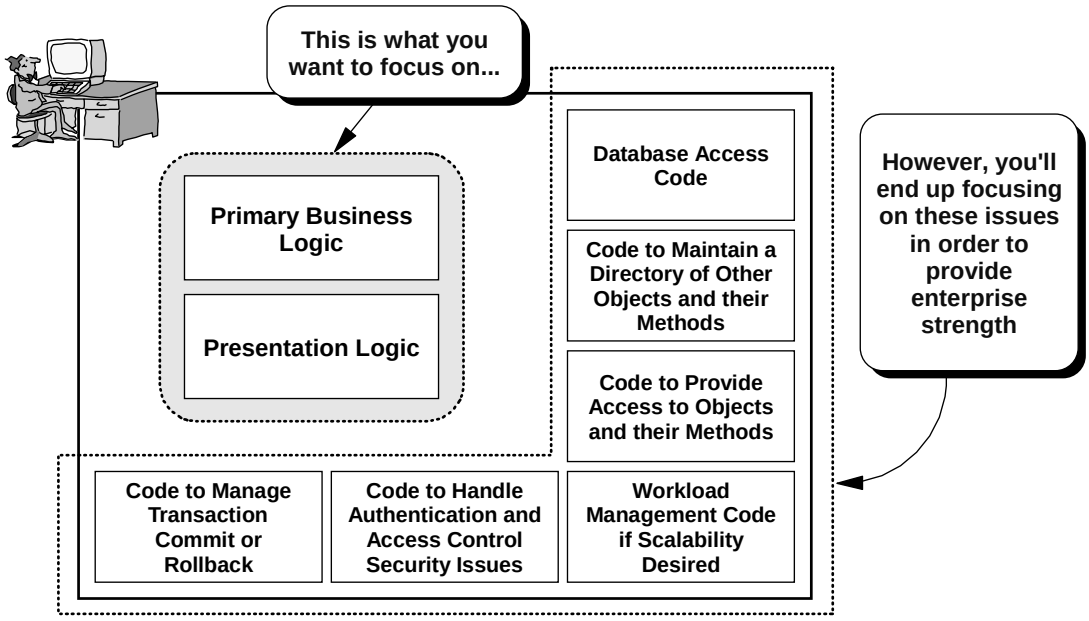
WebSphere Application Server
for z/OS and OS/390

J2EE Architecture Overview

(This page intentionally left blank)

The Challenge ...

Your assignment: write a Java application that provides enterprise-strength functionality



Without the benefit of other software products, this would be a lot of work. The code would probably not port to other platforms, and the code you wrote probably wouldn't be re-used much.

■

The genesis of the J2EE architecture was the dilemma Java programmers faced when they looked to write code that was more than relatively simply servlet and JSP solutions. When the functional requirements of the solution being considered included things often associated with enterprise-wide solutions, the Java developer was faced with the daunting challenge of constructing a great deal of infrastructure code -- or "plumbing" as it is often referred to as -- rather than focusing on the core business logic and presentation logic of the solution.

Like every other programming environment in the past, the answer to this dilemma is to have the server platform vendor provide that "plumbing" as an underlying set of services of the platform. The problem in the past has always been that each vendor implemented it differently, forcing the application developers to choose a platform and be thereafter locked to that platform.

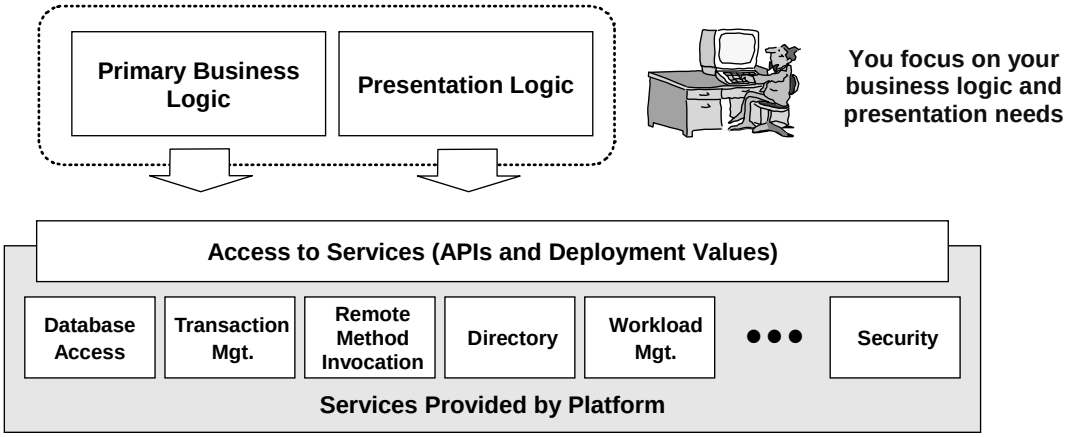
The Java programming environment's most significant benefit is the "write once, run anywhere" aspect of its architecture. This is achieved through a standard specification of what each vendor who supplies a Java platform must provide to be Java compliant. With such a standard in place, developers could be assured that their code could be portable across platforms.

The common things that come into play when considering an enterprise application are shown above. So the challenge was to come up with a way to insure vendors could implement that plumbing according to a standard.

The Vision and Result

The Vision:

Come up with a standardized specification for a server platform that provides those services needed by an enterprise application. Application developers then code their solution using the services provided by the platform.



The Result:

Sun, in cooperation with industry leaders including IBM, defined the "Java 2 Enterprise Edition" ("J2EE") specification. This specification defines the functions a J2EE platform must provide, the structure of the application code to take advantage of the platform, and the API used by the application code to invoke the functions.

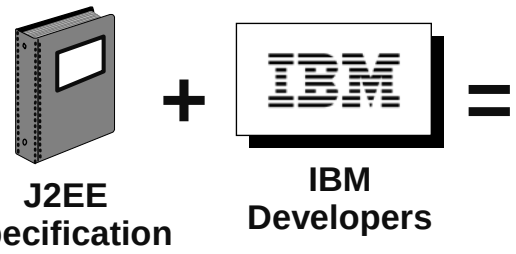
■

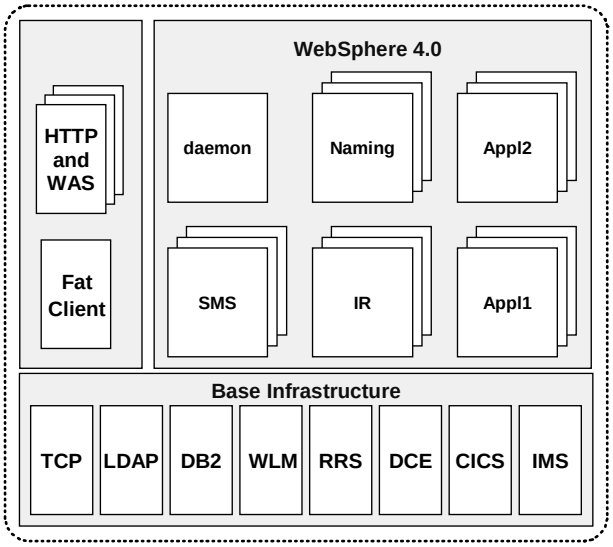
Given the nature of the problem illustrated on the previous page, the vision was to come up with a standardized specification -- not a product, but a written set of rules and guidelines by which server platform vendors would guide the development of their offering -- which provided a server with the necessary services which would be accessible through standard APIs. The application developers could then focus on the primary business and presentation logic of their solution, and rely on the services provided with the platform. They would access those services in a standardized way, which would insure their applications would not be locked to a particular vendor's platform.

The result was the "Java 2 Enterprise Edition" (J2EE) specification. This specification was developed by Sun in conjunction with other software leaders in the industry. The specification itself is not a product; it is a published set of functions and interfaces that must be part of a server platform before it can be called a "J2EE Server." The specification also imposes some structure on the way in which the developers code their solutions.

As mentioned in the previous paragraph, the specification was just a stack of paper. It required vendors to take the specification and build an actual J2EE server. IBM did just that for the z/Series platform ...

IBM's Implementation on z/OS





WebSphere Application Server Version 4.01 for z/OS and OS/390 is IBM's implementation of the J2EE specification for the z/OS platform

You'll be seeing this picture many more times in this course. It is the logical representation of the WAS 4.01 structure on z/OS.

It'll be described in detail in the next presentation.

■

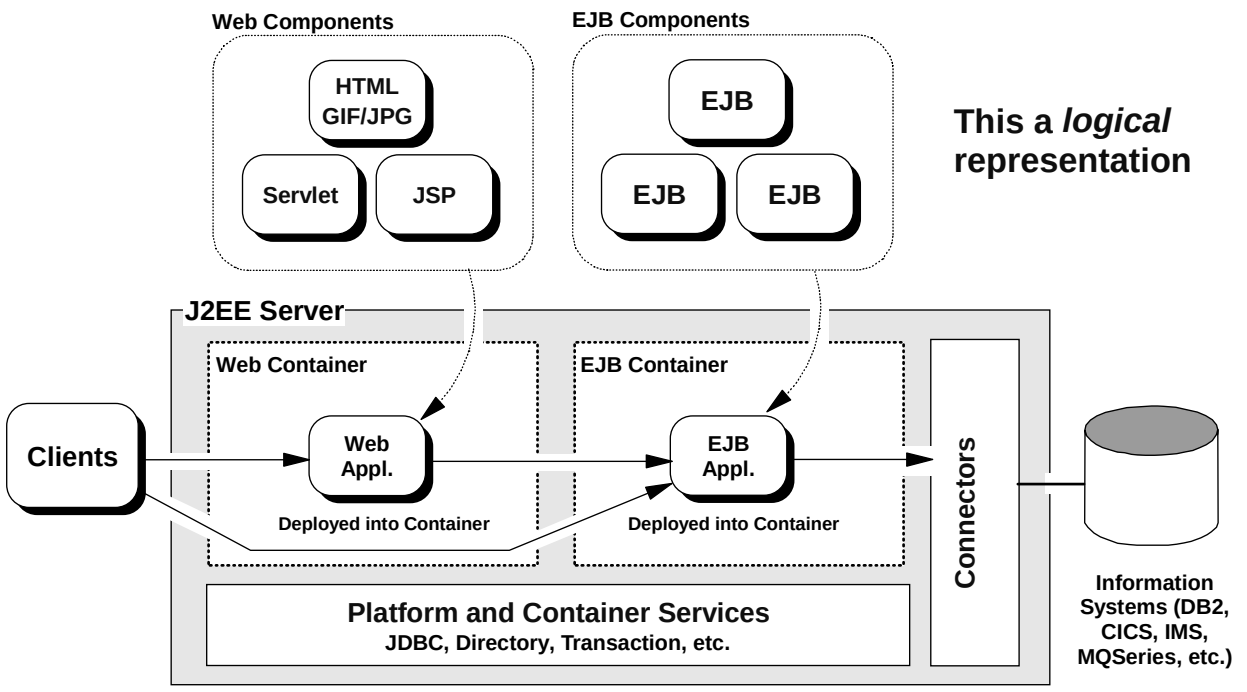
IBM developers took the J2EE specification and developed an IBM product called "WebSphere Application Server Version 4.01 for z/OS and OS/390." That product, known in shorthand as WAS390 or WAS 4.01, is the physical implementation of the J2EE specification.

The diagram to the right of the equal sign in the chart above is the logical layout of the physical implementation on the z/Series platform. You will see this picture quite a bit throughout this class. The picture represents the base infrastructure components required by WAS 4.01, as well as the address space regions that make up the WAS 4.01 "runtime environment." What each box in that diagram means and how it all relates to one another will be presented in the next section of this class.

The key message here is that the specification comes to life only when a vendor implements the functions and interfaces spelled out in the specification. IBM has done that for the z/Series platform (and for the AIX and Intel servers as well). The product is compliant to the J2EE specification to the applications deployed on the platform; the "gears and wheels" under the cover that makes it all go consists of many different IBM products tied together into a working system.

Now let's look at the J2EE architecture itself.

Basics of J2EE Architecture



The starting point for this discussion is a brief discussion of the two kinds of "components" that go into a J2EE solution:

- ***Web Components*** -- standard Web stuff you're probably already familiar with: Servlets, JSPs, HTML page and image files.
- ***EJB Components*** -- An EJB is a special kind of Java program. EJB stands for "Enterprise JavaBeans" and they make up the primary building blocks of the application logic and data. We have a few more charts on what an EJB is, so for now just know that such a thing as an EJB exists.

These components are "deployed into" the J2EE server for execution. The act of "deploying" an application and its components involves using the tools associated with the platform. We'll cover that later in this presentation (at a conceptual level), and then in more detail in a later presentation.

The components are deployed into what are known as ***containers*** within the J2EE server. Containers are logical things within the J2EE server -- they are software structures but not actual MVS started tasks or anything you can easily put your fingers on. But they serve to provide a standardized execution environment for the components, and they insulate or shield the components from the underlying complexity of the server platform. There are two kinds of containers: Web Containers (for web components, naturally), and EJB Containers for EJBs.

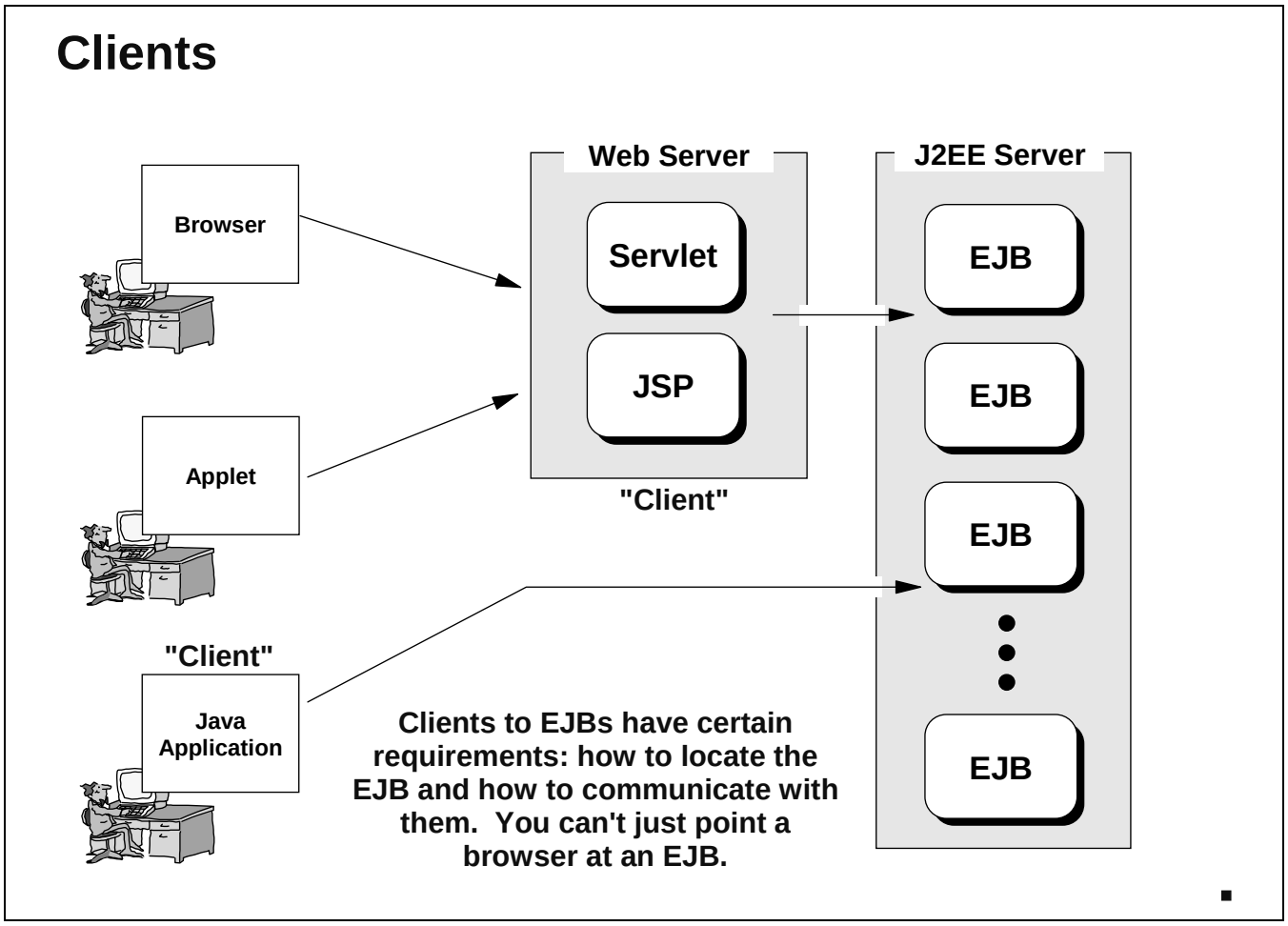
To drive the components you need a ***client*** to the server portion of this equation. Clients are software programs that are written to understand how to locate, communicate and execute the methods of the EJBs in the solution. Clients can exist in several different forms in several different environments, and those variations are covered in a later chart.

Underneath the components and the containers in which they run exists the ***services*** available for use. These services are defined in the J2EE specification and are accessible through standard interfaces. How the vendor chose to implement the service is up to them as long as the behavior is according to the specification and the interface as seen by the components is standard.

Most J2EE applications will require data that resides in backend systems outside the J2EE server environment. These backend systems are things like CICS, IMS and DB2. To access those backend systems another part of the J2EE specification is brought into play: ***connectors***. Connectors provide a way for EJBs to access these backend systems in a standardized way. We'll cover these connectors in concept later in this presentation, and then in much greater detail later in this class.

What's presented in this chart is a *logical* layout of the J2EE structure. The *physical* layout of the WAS V4.0 runtime environment looks different and will be covered in the next presentation.

Let's go a bit deeper, starting with the clients ...

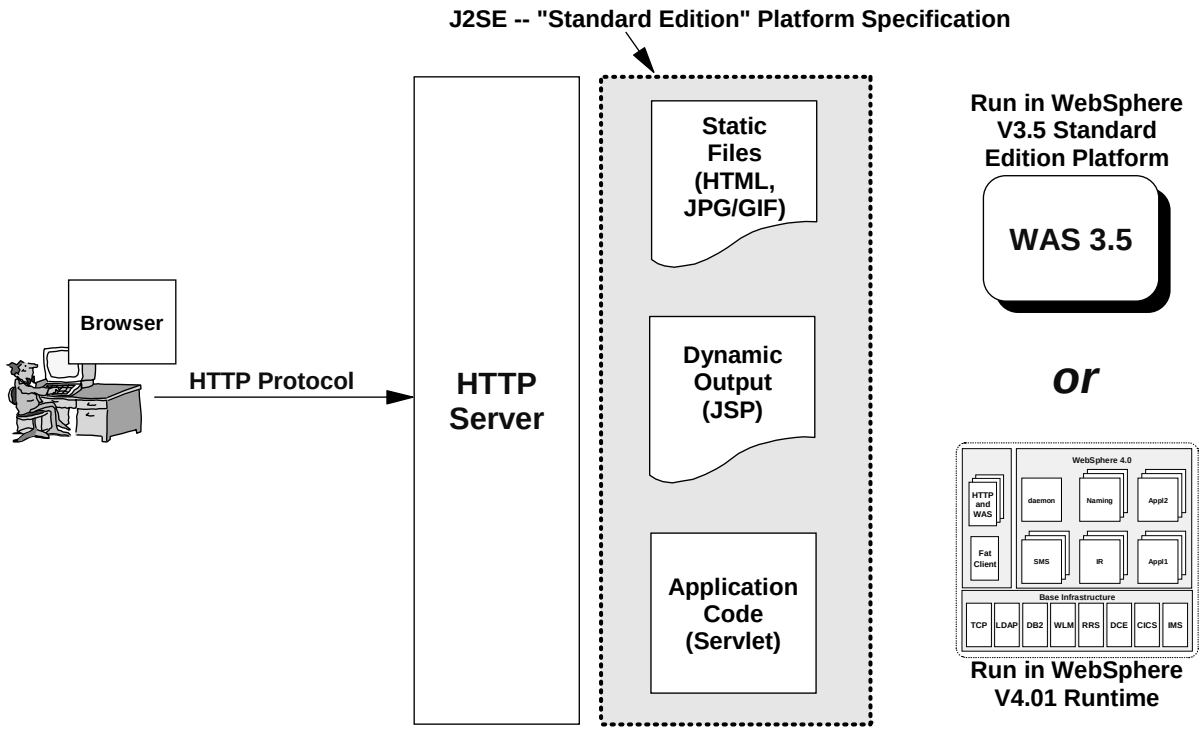


As mentioned earlier, a client is needed to make things happen on the server side of this equation. Clients are software programs that are programmed to know how to look up (using the Java Native Directory Interface -- JNDI) the location of the EJBs, and how to communicate with the EJBs. Clients are *not* just browsers; a browser can't talk directly with an EJB. For a browser (or applet, for that matter) to drive an EJB, they must go through either a servlet or JSP. If that's the method of access, the "client" is actually the servlet or JSP.

If the client is a Java program, it's often referred to as a "fat client" because the client has this additional function to know how to interact with the EJB. A fat client can reside and execute in a number of different places: in the OMVS environment on S/390, either on the same box as the WAS 4.0 runtime or another MVS machine; on another distributed platform like AIX or Windows2000, or even running in a DOS environment.

For this class we'll be using a "fat client" for some of the labs to verify things work properly, and web clients (servlets) in other labs.

Web Components



These are the standard web components you may already be familiar with. They can be run and served from the J2EE platform as well.

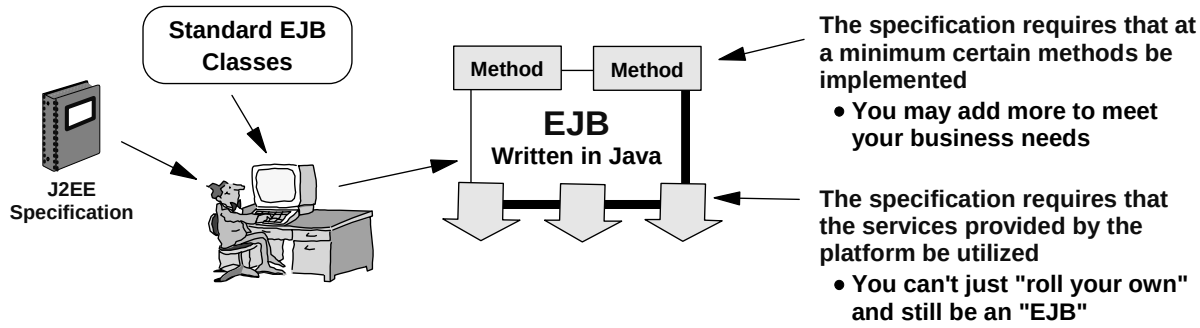
Now let's talk about web components. They consist of static files (HTML, JPG, GIF), files used to generate dynamic HTML (JSP), and application code in the form of servlets. These web components are not new to WAS 4.01: they've been around for a few years.

The key here is this: these web components may be run on the WebSphere Application Server Standard Edition (WAS 3.5) platform (which is a "plugin" to the Webserver on the S/390 box), or they may be run inside the J2EE Server. When web components are deployed into the web container of a J2EE server, the web container provides an execution environment that adheres to the Java 2 Standard Edition (J2SE) specification just like the WAS 3.5 runtime provides. This means your servlets and JSPs do not need to be rewritten (provided the Servlet and JSP specification levels of your components comply with what the web container provides, which is Servlet spec 2.2 and JSP spec .91, 1.0 and 1.1).

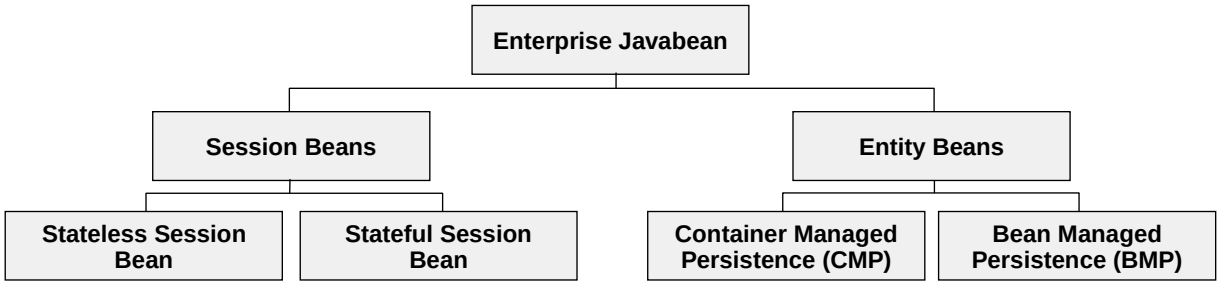
Standard stuff ... and the point is this: just because we have a new server environment (a "J2EE" server), it doesn't mean the work you've invested in servlets and JSPs and HTML is wasted. That investment can be deployed again in the J2EE environment. More on how to do this later in the class.

Enterprise Javabean (EJB) Components

In order to be considered an EJB and to run on a J2EE server, the program must abide by certain standards:



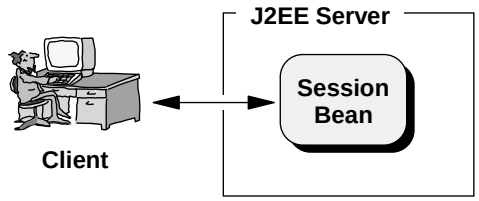
Here's how EJB's are categorized:



EJBs are Java programs, and they are used in the J2EE environment to provide the implementation of the business logic and the representation of the business data. They are Java programs, yes, but they are special in that they must be written to certain standards about how they are structured and how they behave. The specification calls for EJBs to include (or extend) certain key standard EJB classes, and the EJBs must implement certain methods so the instances of the object can be created, managed, and removed. The EJBs must also make use of the underlying services provided by the platform. A developer can't write their own special handling of these services and deploy it into a J2EE environment and call it an EJB. (There's an exception to this, and it has to do with managing the permanent "persistence" of data in a data source. The standard allows some flexibility there.)

EJBs come in several different flavors, and the chart above shows the way they are organized. We'll cover each of those in the next several charts.

Session Beans

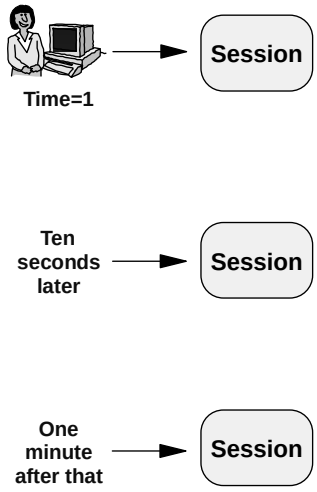


A "session bean" is Java code that maintains non-permanent information about an EJB client.

A bank application example might be "LoanApprover" bean, "CarLoanCreator" bean, "Deposit" bean, "Payment" bean -- activities related to a banking application.

A J2EE application might consist of several different session beans.

Stateless Session Beans



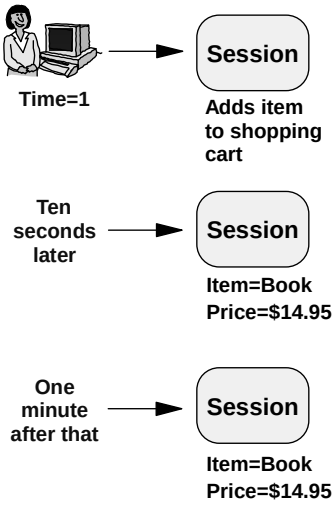
Each time the person connects to the Session bean, the information contained by the bean is the same.

The "state" of the information is *not* maintained between connections

Person my not get same instance of object on return clicks

It is state*less*

Stateful Session Beans



Information provided by the client to the bean is maintained in the bean.

Each return click results in client using the same instance of object

The "state" of the information *is* kept between connections

It is state*ful*

■

One category of EJBs is what's called a "Session Bean." The name is somewhat misleading in that the purpose of the bean is not *just* to control the session between client and server. Session beans will typically contain business logic for various pieces of your overall application solution. The key to understanding session beans is to realize that the data held in the beans is *non-permanent*. The data is held in memory, but not written out to a database (Entity Beans serve that function, which we discuss next). If the instance of the object is lost, the person on the other side of the client will have to re-type the information.

An application solution will consist of perhaps many different session beans, each representing a different piece of the overall solution. An example would be a bank application which might consist of several different session beans: a "LoanApprover" session bean that has the logic in it to allow a person to review and approve a pending bank loan; a "CarLoanCreator" session bean with logic to allow a person to create a car loan application for a customer; a "Deposit" session bean that allows a bank teller to initiate and process a deposit; and a "Payment" session bean that contains code to process a customer's payment of loans, utility bills and municipal taxes.

The non-permanent nature of session beans is what is difficult to grasp at first. How can you possible execute a deposit action if the deposit information isn't permanent? The key to understanding this is to accept -- for now; we cover this on the next chart -- that the session bean will invoke the methods on an "Entity Bean" to handle the "making permanent" the information entered by the client. If a bank teller was interacting with the "Deposit" session bean and had typed in the customer's bank account number and deposit amount *but had not yet committed the deposit*, and a server glitch occurred, they would need to re-key that information when they got back into the system. The information in the session bean is non-permanent until an entity bean is brought into play to make the data permanent.

There are two types of session beans: ***stateless*** session beans and ***stateful*** session beans. The difference between the two is somewhat subtle, and it has to do with whether a person driving some client code gets the *same* session bean each time they return to the server with a mouse click. We'll start with stateful session beans.

- State*ful* Session Beans

  We said that the data in a session bean is non-permanent, but doesn't mean the data instantly vaporizes; for stateful session beans the data is maintained (in memory, but maintained nevertheless) as long as that instance of the bean object exists in the server environment. Imagine you're working with a "ShoppingCart" session bean, and you're happily clicking on things you want to buy. The bean is maintaining information on the products you desire, but the information has not been committed to a database; it's just held in memory for now.

  Each time you click on a product to add to your shopping cart, your client is connected back to the *same* session bean. (The mechanism by which the client flow knows to get back to *that* bean is beyond the scope of this presentation. For now, just trust that the instance of the bean you're working with has a tag of information which is used to route your next mouse-click flow back to this copy of the bean, and not someone else's.)

  Because the information is maintained in the bean, and because each return trip to the server results in your connecting to the same instance of the bean, it is said that bean maintains its state, or it is stateful. The shopping cart concept is often used to illustrate this, but that concept is somewhat clumsy in that most shopping cart implementations in the real world do maintain the information in a database. But we hope the concept of the data being maintained across time is apparent.

  Note: Without going into too much detail, other considerations come into play when the duration of time between mouse clicks gets longer and longer. Various connection timeout values will expire if, for example, you add a product to your shopping cart and then go take a four hour nap. In that case the instance of the bean would be removed and your data would be lost.

- State*less* Session Beans

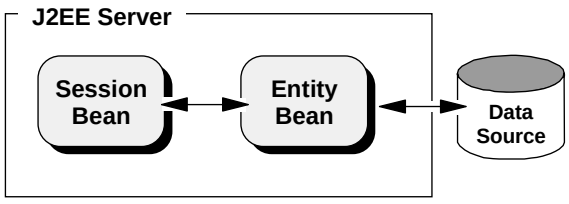
  Imagine a scenario where a business activity is very short and very simple. The simple bank deposit example is a good one to work with. The teller invokes the "Deposit" function of their client system, and all they enter is the bank account number and amount of the deposit and then hit "enter." If the bank customer then walks away from the teller session and 30 seconds later another customer walks up to make a deposit, the teller will again invoke the "Deposit" function. Will the teller be connected back to the *same* session bean? Does it matter?

  The answer is "probably not" to the first question, and "no" to the second. The nature of the deposit activity -- from the session bean's perspective; remember, a whole bunch more goes on behind the scenes after the session bean passes the deposit information along to the entity bean for permanent storage in the database, but the session bean isn't involved with that -- is very short lived. The data entered by the teller -- account number and deposit amount -- is passed in by the client, the session bean is activated, the data is received and passed back to an entity bean, and the need for the session bean to maintain that data is over. The session bean can be removed, or set back into a pool for some other teller to come in and use.

  Because the data is not maintained between connections, the state of the session bean is considered to be reset after each use. It is therefore considered to be a stateless session bean.

Several references to "Entity Beans" have been made in that description of session beans. Let us now go look at those.

Entity Beans

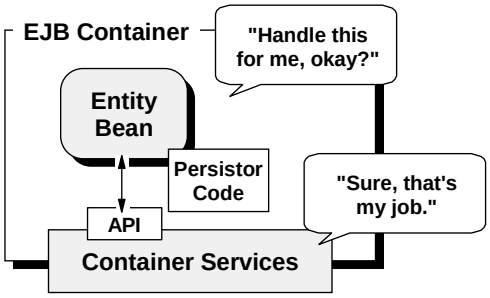


An "Entity Bean" is Java code that represents a piece of "permanent" data, such as a row from a database table.

Entity Beans interact with a data source that physically stores the data; this is "persistence"

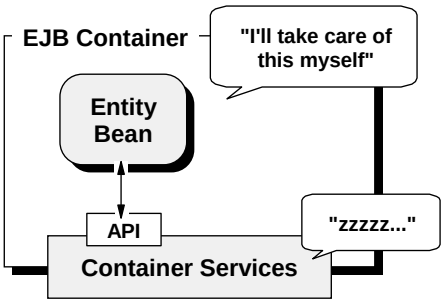
Entity Beans are typically accessed and acted upon by Session Beans

Container Managed Persistence (CMP)



The low-level work required to connect to the data source and store or retrieve the information is handled by the Container. The Bean simply calls the API and asks for the service.

Bean Managed Persistence (BMP)



The application developer writes the code in the Bean to manage the persistence, including SQL statements. Use only when you have compelling reason to override container services.

An "Entity Bean" is Java code that represents permanent data, and that data is made permanent by it being stored in some storage system like a relational database or other backend system. A typical way of thinking of these things is to consider a row from a database table, which has, let's say, three fields with a customer's name, bank account number, and current balance. The entity bean's responsibility is to hold that information, and to make sure the data it has in its variables is synchronized with the information committed to the database. In the previous chart's description we talked about a stateless session bean interacting with an entity bean and passing deposit information. The session bean invoke the "create" method of the an entity bean, which would create an instance of the entity bean to hold the customer's balance information. Upon receiving the deposit amount, the entity bean would update the balance value which is then written to the database.

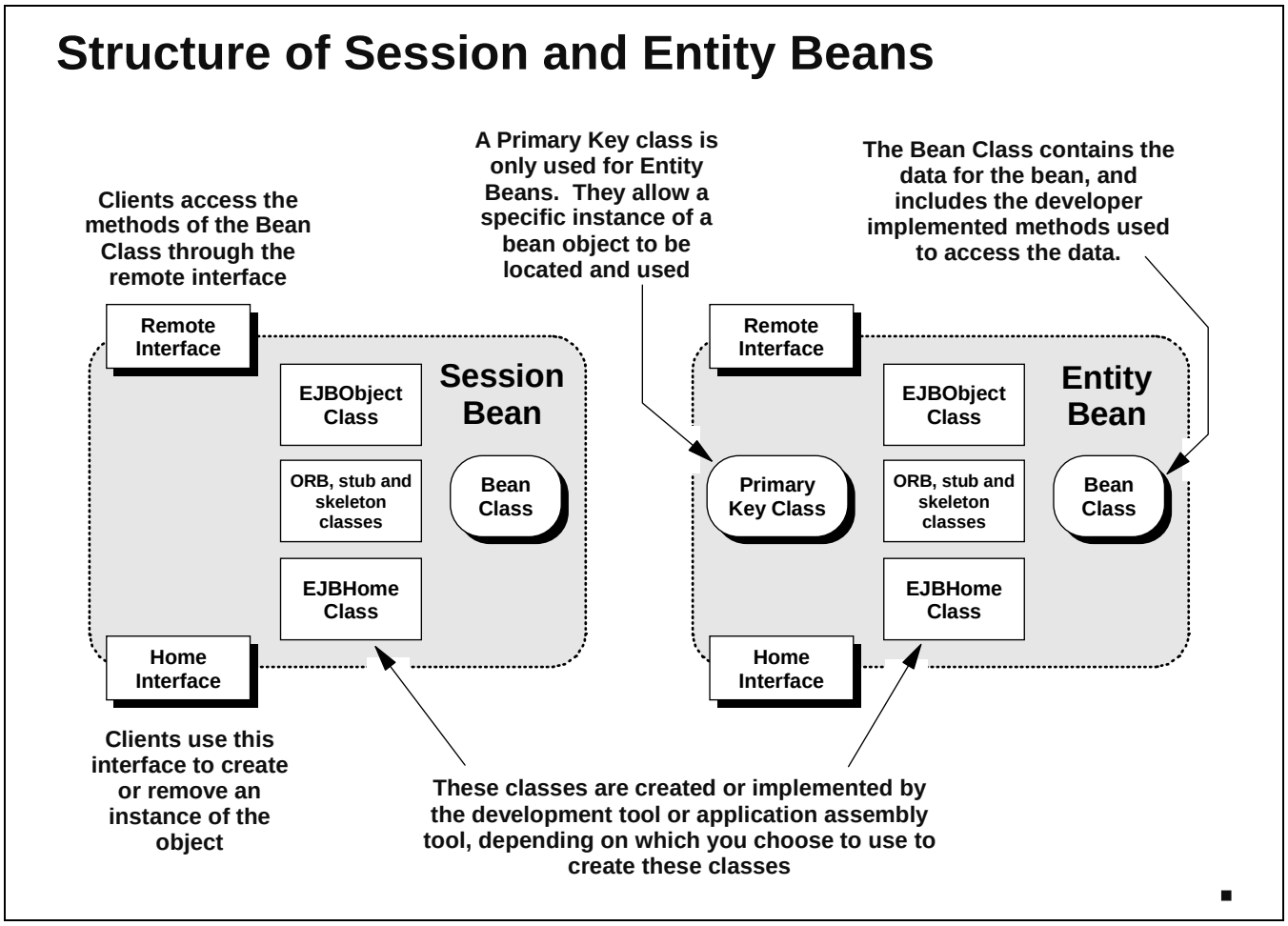
Note: These examples are intentionally simple in nature to get across a point. In a real banking application more than just an update to the balance variable would take place. For example, the deposit amount, along with the date and time and probably the branch location would be captured as well so that a transaction history could be maintained.

There are two kinds of entity beans: Container Managed Persistence (CMP) entity beans and Bean Managed Persistence (CMP) entity beans. The difference between the two is who does the grunt work to get the data into or out of the database.

In a ***CMP bean***, the application developer writes his or her code to ask the "container" (the logical structure inside the J2EE environment and in which the EJB is running) to handle that work. That is one of the "services" the container provides. The application developer doesn't code the bean to manage the persistence, they code the bean to have the *container* manage the persistence. Hence the name *Container Managed* Persistence (CMP). The code that handles the interaction with the data source is called the "persistor code," and it is generated by the tools (VisualAge or the AAT tool in the

case of WAS 4.0). The application developer doesn't even have to code any SQL; the container (which has been provided information about the database tables and fields to which the bean's data variables are associated by the person who "deployed" the application into the container) does the SQL grunt work under the covers.

In the case of a ***BMP bean***, the application developer has chosen to code that grunt work into the logic of the bean itself. This is permitted by the J2EE specification because the designers of that specification knew the container couldn't be given infinite flexibility for all issues of persistence. If the bean is coded as a BMP, the container will step back and let the bean do the hard work, providing other services to the bean but not managing the persistence of the data as closely as would be the case with a CMP.



Here's where the discussion gets a little deeper into the makeup of the bean itself. This level of detail is being presented because some of the concept here will serve you later when the issues of the "naming server," "JNDI," "LDAP" and application assembly and deployment come into play.

- ***Bean Class***

  This is the Java code written by the application developer. Interestingly, the methods and interfaces of this code are *not* exposed to the clients (the "Remote Interface" serves that role). The bean class contains the business logic the developer has written as part of the total application solution.

- ***Home Interface***

  The Home Interface is used by clients to locate, create and remove instances of the object. When a client wishes to invoke the various methods of the bean class, it'll use the "remote interface." The "home interface" is used to manage what's called the "life cycle" of the object: its creation, its in-use state, and its removal. Information about the interface is provided by the deployer of the application; the actual class file that implements the interface is called the EJBObject class, and that class file is generated by the deployment tools when the application is deployed.

- ***Remote Interface***

  The Remote Interface is used by clients when they want to get at the methods coded into the bean class held inside the EJB. Clients never directly access methods of a bean class. Like the home interface, information about the remote interface is provided by the deployer of the application and the actual class file that implements the interface (EJBHome class) is generated by the deployment tools.

- ***Primary Key Class***

  This class is unique to entity beans. It provides a unique identifier for the instance of the entity bean object. Remember that entity beans represent data, such as a person's bank information and balance. One instance of the bean object might be for "Mr. Jones' account" and another instance of the bean object is "Ms. Smith's account." Something has to keep them separated, and the primary key serves this function. In the case of a bank account record like this, the account number (presumably unique) would probably serve as the primary key information contained in the Primary Key Class.

- ***EJBObject Class***

  Earlier we stated that information about the remote interface is provided by the deployer of the application, but the actual class file (the honest-to-gosh compiled bytecode for the interface) is generated by either VisualAge for Java (as an example of a development tool) or by the application assembly tool The EJBObject Class is that the actual class file that implements the functions of the "remote interface."

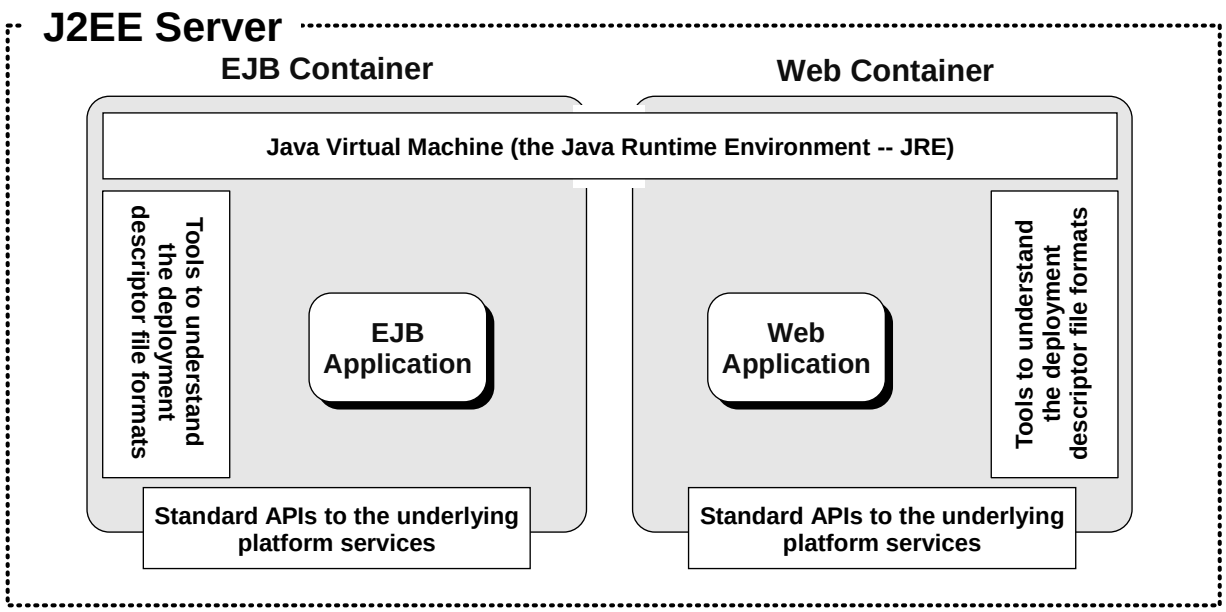
- ***EJBHome Class***

  This is the compiled Java class file that implements the functions of the "home interface." Like the EJBObject Class, this is generated by the development or application assembly tool.

- ***ORB, stub and skeleton classes*** (and finder helper and persistor classes for entity beans)

  Finally, the development tools generate a handful of other class files that comlete the EJB picture. The ORB is the "Object Request Broker," and is code used when one object wishes to communicate with another object. The stub and skeleton classes are needed for Remote Method Invocation (RMI -- the ability to drive a method of an object when that object resides in a different execution environment). The finder helper and persistor classes are implemented for entity beans to aid in the locating of specific entity beans and to help with the interaction with the persistance service.

EJB and Web Containers



- **Containers are logical structures in your J2EE server.**
- **Each J2EE Server will have two containers: "EJB" and "Web"**
- **Your EJB Applications and Web Applications are deployed into containers**
- **You'll learn later that these are created for you at "Server Instance Creation"**

The EJB and Web components discussed earlier are deployed into, and run inside, things called "container." These containers are logical structures within the J2EE server, not "things" you can put your finger on. These containers provide the execution environment for the components, and they shield, or insulate, the underlying platform implementation from the components.

There are two kinds of containers: "EJB Containers" and "Web Containers." The type of component that runs in each is implied by the name. Any given J2EE server has one of each. The *creation* of the containers is something done for you when you create the J2EE server; however, there is additional *configuration* of the containers that you do after the server has been created.

These containers provide three primary things to the components inside them:

- ***A Java Runtime Environment (JRE)***

  EJB components and Web components are quite different from one another, but both require a Java environment to execute. The specification says each container must have a JRE so Java programs may run.

  Note: The chart above shows a single box representing the JVM spanning both containers. That's because both containers are going to share the same JVM. There is one JVM per J2EE application server, and that one JVM is controlled by a single `jvm.properties` file. Therefore, you can tune your JVM down to the application server level, but not down to the container or application level. If you have an application that requires a unique JVM environment, you may define a separate application server for the application and then tune that server's jvm.properties file specific to the application. More on how to do that later.

- ***Standard APIs***

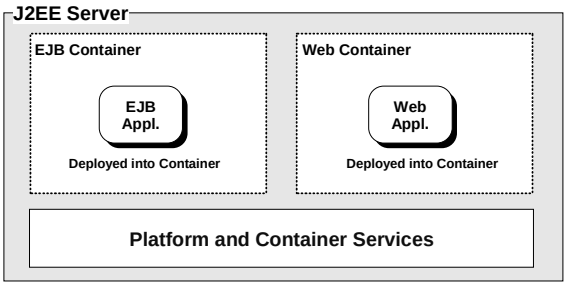
  Developers of the components require a standard environment to write code against, otherwise they would have to code differently for each vendor's J2EE implementation. The specification calls for the interfaces to the underyling services be according to the standards. Developers can be assured that if they write code to the standard interface, their code will run on any vendor's J2EE platform.

- ***Tools to Understand the Deployment Descriptors***

  Applications are deployed into the containers with files called "deployment descriptors." These are XML-based files that contain information about how the application is to behave and run. The containers have the ability to read and understand the deployment descriptors. This provides those who deploy the applications the ability to focus on a standard process for creating and coding these XML files, without regard to the platform in which the application will be deployed.

Platform and Container Services

WAS 4.01 code plus other S/390 products used to provide these services



Deployment Service
Provides services to deploy applications into runtime and generate required additional class files

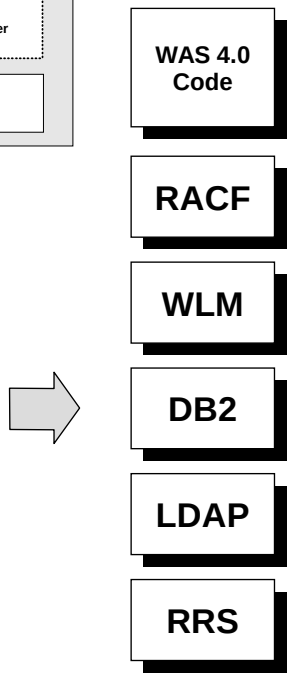
Security Service
Provides authentication and authorization services for EJBs

Workload Management
Provides the ability to scale the solution by running the running multiple instances of the server.

Persistence Service
Provides services that will make permanent (write to DB) the values of entity bean values

Naming Service
Provides a directory service so that objects and their methods can be recorded and retrieved programmatically

Transaction Service
Provides a way to identify components of an application as belonging to a transaction, and a way to treat the collection of activities as an atomic unit, including rollback

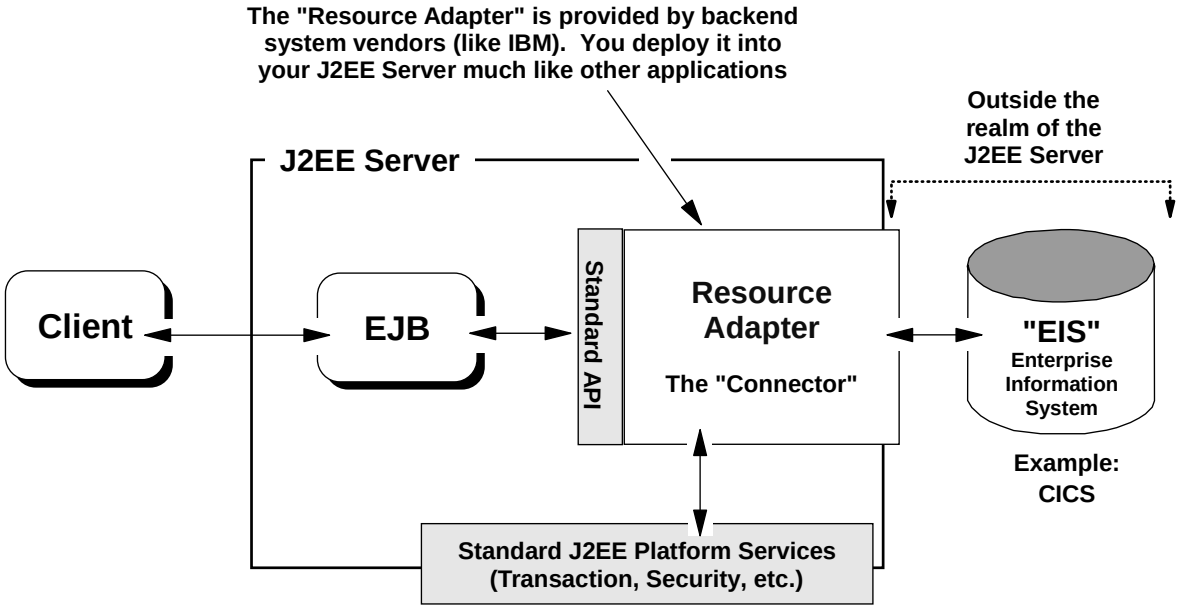


This presentation has made many references to the "underlying services of the J2EE server." These services are part of the J2EE specification, and any vendor looking to implement the specification into a real product (as IBM did with WAS 4.01 for z/OS and OS/390) must implement the services. The services are shown and described above.

Now, the specification does not say *how* the services are to be implemented, only that they be provided and access to them be through standard interfaces. The gears and wheels under the interfaces may be implemented in any manner the vendor feels works to their advantage. The developers of WAS 4.01 for z/OS and OS/390 took a look at the services and decided many of them can be implemented using existing OS/390 products. That is why, as this class goes on, you'll see these other products brought into play while creating servers and deploying applications.

The physical structure of the WAS 4.01 structure will be presented in the next presentation. For now, just be aware that the services have *standard interfaces*, but IBM has used existing components, in conjunction with new WAS 4.01 code, to implement the services.

Connectors



The Connector Architecture provides a standard way to access traditional backend systems such as CICS and others. Vendors supply a "Resource Adapter" which deploys into J2EE server. You code your EJBs to access API of connector, and it does the "behind the scenes" work.

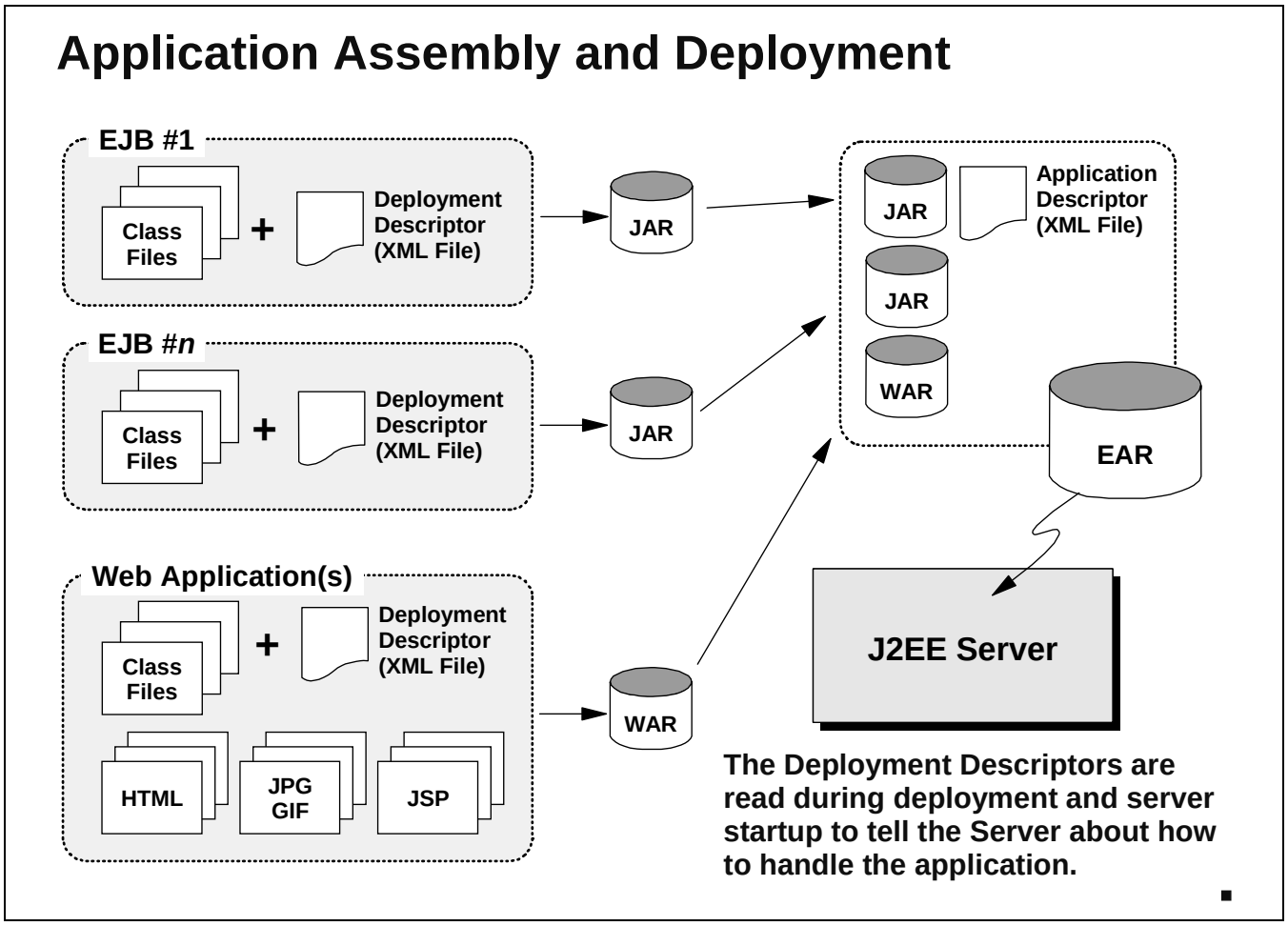
■

The designers of the J2EE specification provided a standard service for access to relational databases (this is the JDBC service). However, they also knew that a great deal of important data is stored in other backend systems other than relational database systems. These other systems, known as "Enterprise Information Systems" (EIS), are things like CICS, IMS, SAP, etc. The designers did not want to make access to every known EIS a requirement for every instance of a J2EE server, so they provided an architecture that defines something called a "Resource Adapter." These adapters, or "connectors," are software components deployed into your enviornment much like other applications, but their purpose is to provide access to the EIS.

The connector has a standard API on the J2EE-side of the picture. This allows developers of J2EE components the ability to write to a known set of standards, and be confident their code will be portable to other platforms if required. On the "backend" of the connector is code that does whatever is necessary to make connections to the EIS in question. In addition, the connectors make use of the standard services of the J2EE platform for things like transaction management and scalability.

Vendors of the EIS will be the ones who write and deliver the connectors. If you have a J2EE server installed at your location and you have an EIS such as CICS to which you need to connect, you acquire the connector from IBM and deploy it into your environment. If later you decide you wish to give J2EE components access to, say, a PeopleSoft server, and the folks at PeopleSoft have a connector available, you would deploy that connector.

The subject of connectors will be explored in greater detail later in this class.



We have described two kinds of components in the J2EE world: EJB components and Web components. Some number of components make up an "application," and the act of installing these applications on your J2EE server is known as "deploying" the application.

The J2EE specification defines the structure of the *packaging* for the application. The packaging involves different files and files called "deployment descriptors." Let's look at how this fits together:

- ***Packaging EJBs***

  EJBs are Java programs with a requirment to adhere to certain standards as described earlier in this presentation. Java programs -- and EJBs -- are typically developed using a development tool like IBM's VisualAge for Java. When the EJB is fully developed, the packaging for it is something known as a JAR file. JAR files have a ZIP format and contain the various class files required by the EJB (see "Structure of Session and Entity Beans").

  JAR files contain EJBs, but JAR files themselves can't be deployed into the J2EE server. The JARs need to be incorporated into another file type called an "EAR File." That's described in a bit.

- ***Packaging Web Applications***

  Web applications consist of not only Java class files, but other static components like HTML pages, JSP pages and GIF/JPG image files. Web applications are often generated with a tool such as IBM's WebSphere Studio. When the application is ready to go, the files are placed in a WAR file. WAR stands for "Web ARchive." WAR files have a ZIP format just like JAR files, but contain web applications and not just Java class files. The WAR file has a "deployment descriptor" XML file as well. That deployment descriptor is read at deployment time and at server startup time to provide the server with knowledge of how to handle the web application.

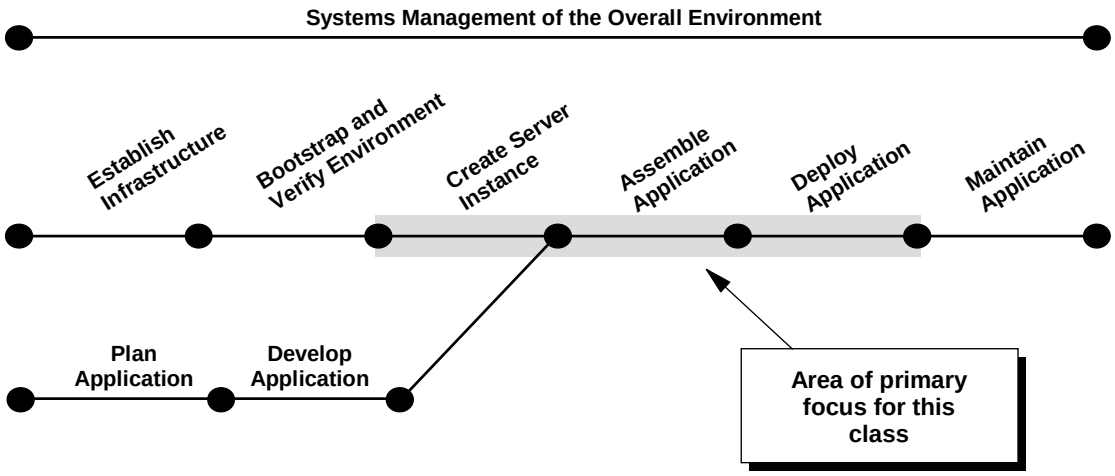
WAR files can't be deployed directly into a J2EE server, just like JAR files can't. You need to package the WAR (and JARs) into an "EAR File."

- ***Packaging J2EE Applications***

  When you have the JARs and WARs ready to go, you bundle them up into what's called an EAR file. EAR files are ZIP format files as well, and they too have a "deployment descriptor" XML file. The tool used to generate EAR files for the WAS 4.0 for z/OS platform is called the "Application Assembly Tool" (AAT), and it will create the EAR file as well as generate class files described in the chart "Structure of Session and Entity Beans" earlier in this presentation. The output from the AAT is a file with an extension of EAR, and that file is what you deploy into the J2EE environment.

The act of deploying the application into the J2EE environment involves using the management interface tool of the J2EE server platform. In that process you will point to the EAR file to be deployed, provide a few bits of additional information, then tell the tool to deploy it into the server. That process will be discussed in greater detail later in this course.

Activities and Roles



The J2EE specification also defines various "roles" that people will perform as part of the overall operation of your server environment. This picture expands on those defined roles and includes the activities related to seting up the environment, as well as maintaining it.

- ***Establish Infrastructure***

  For the WAS 4.01 for z/OS environment, the establishment of the infrastructure involves not only installing the product, but running through a series of jobs that defines things like the WLM application enviornments, RRS log steams, RACF definitions, DB2 table creation, etc.

- ***Bootstrap and Verify Environment***

  Once the infrastructure is in place, the WAS 4.01 environment needs to be initialized. This process is called "bootstrapping the server," and that involves running several jobs that populates the DB2 tables and LDAP naming space with information about the server. Once "bootstrapped" the environment is ready for the creation of server instances.

- ***Create Server Instance***

  The WAS 4.01 environment supports the creation and running of multiple server "instances" (comprised of a "control region" and some number of "server regions" ... more on this later). Applications are deployed into server instances, so at least one needs to be created. This involves using the "Systems Management End User Interface" (a PC-based graphical tool supplied with the product that connects to the Systems Management server region and allows you to execute functions on the S/390 box from your PC) to set the definitions for a server instance, and then running a few jobs to create RACF IDs and WLM application environment in support of the server instance.

- ***Assemble Application***

  The act of assembling the application involves taking the various piece-parts of the application and packaging them up in to the EAR file. The tool you use to do this is the Application Assembly Tool, a PC-based application provided with the WAS 4.01 product. Once the EAR file is created, the next step is "deploying" the application.

- ***Deploy Application***

  Deploying the application involves taking the EAR file, bringing it into the Systems Management End User Interface, and then having the Systems Management tool FTP the EAR file up to the S/390 box and performing the steps it does to put the application into the HFS, update the DB2 tables and register the application in the LDAP naming space. All that sounds complicated, but much of it is done by the tool and server and requires little more than clicking of the mouse.

- ***Maintain Application***

  Once an application is deployed and is in operation, the standard efforts to maintain the application come into play. This phase has little to do specifically with the J2EE environment, and more to do with what has traditionally been the activities involved with maintaining an application: monitoring its operation, performing problem determination and developing fixes as needed, maintaing some kind of change-control mechanism for the application, fielding suggestions and complaints from the users, and performing periodic maintenance.

- ***Plan Application***

  When a J2EE application is still in the planning phase, the application designers and developers get together and architect the application, mapping out the sources of input data, the structure of saved data, the flow of the user-interface environment, and the flow of the internal logic. This requires some knowledge of the EJB development environment, but is not so much platform dependent.

- ***Develop Application***

  At some point the developers have to put their heads down and start coding.

- ***Systems Management of the Overall Environment***

  When all is said and done, you'll have a S/390 environment running this new thing called WAS 4.01 for z/OS and OS/390. It'll also have all the other things you've grown accustomed to maintaining: RACF, DB2, CICS, etc., etc. This phase involves things like backup and recovery planning, maintenance planning, resource usage monitoring, etc. All the things you have done for years with your S/390 system continues when the WAS 4.01 server is also part of the mix.

End of Document