4 : Overview of JEE

IT 4206 – Enterprise Application Development

Level II - Semester 4

Overview

- Identify key features of application servers and describe the Java EE server architecture.
- Identify various types of containers and server profiles.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Describe what is an application server
  - Packaging and deploying a Java EE application

List of sub topics

1.1 Application server

- 1.1.1 Introduction
- 1.1.2 Enterprise Application Platform(EAP)
- 1.1.3 Containers
- 1.1.4 Java EE 7 profiles

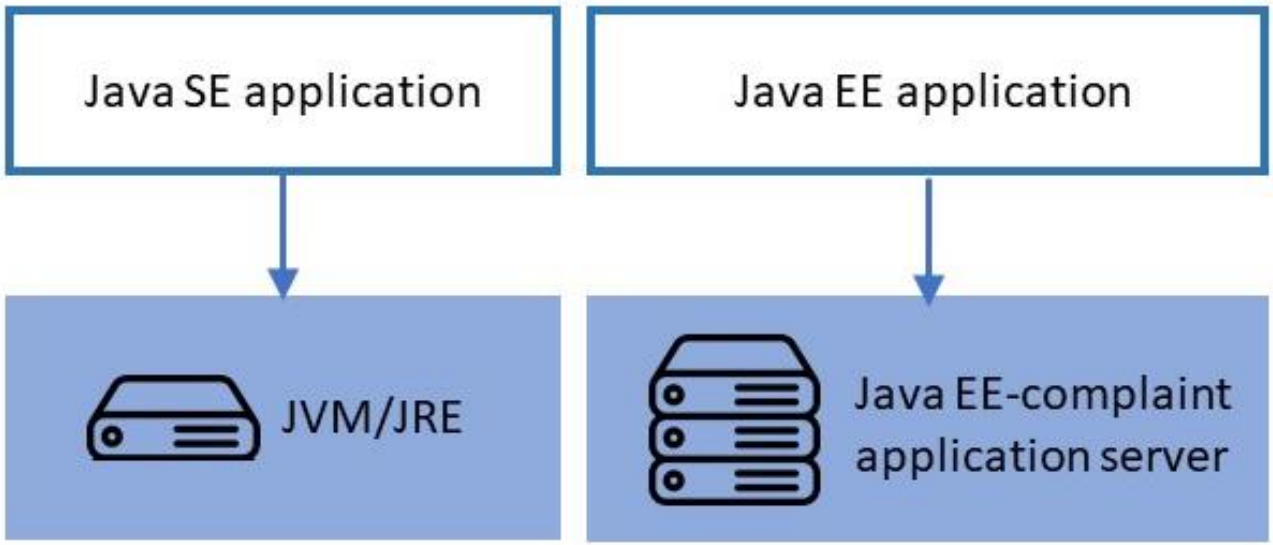
1.2 Packaging and deploying a Java EE application

- 1.2.1 Introduction
- 1.2.2 JAR Files
- 1.2.2 WAR Files
- 1.2.2 EAR Files
- 1.2.2 Packaging and deploying java EE applications to EAP

1.1 Application server

- An *application server* is a software component that provides the necessary runtime environment and infrastructure to host and manage Java EE enterprise applications.
- The application server provides features such as concurrency, distributed component architecture, portability to multiple platforms, transaction management, web services, object relational mapping for databases (ORM), asynchronous messaging, and security for enterprise applications.
- In java SE developer has to implement them manually.

- Comparison of Application server in Java SE and Java EE



ENTERPRISE APPLICATION PLATFORM (EAP)

- EAP is an application server to host and manage Java EE applications.
- It is a centralized management of multiple server instances and physical hosts.
- EAP makes developing enterprise applications easier because it provides Java EE APIs for accessing databases, authentication, and messaging.

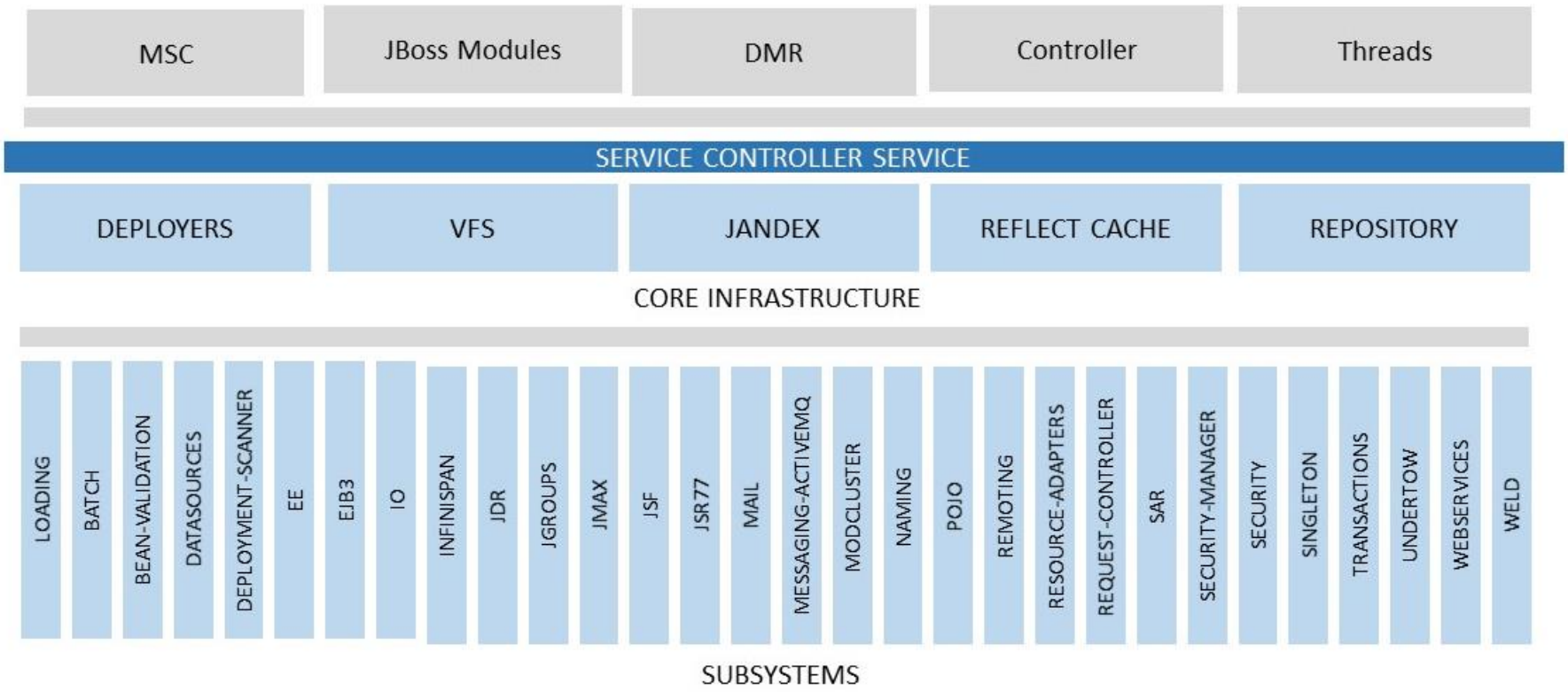
ENTERPRISE APPLICATION PLATFORM (EAP)

- EAP has a modular architecture with a simple core infrastructure that controls the basic application server life cycle and provides management capabilities.
- The core infrastructure is responsible for loading and unloading modules.
- Modules implement the bulk of the Java EE 7 APIs.
- Each Java EE component API module is implemented as a subsystem, which can be configured, added, or removed as required through EAP's configuration file or management interface.

EAP offers features such as:

- High availability clustering
- Distributed caching
- Messaging
- Transactions
- A full web services stack

EAP Architecture



Containers

- A container is a logical component within an application server that provides a runtime context for applications deployed on the application server.
- A container acts as an interface between the application components and the low-level infrastructure services provided by the application server.
- Containers are responsible for security, transactions, JNDI lookups, and remote connectivity and more.
- Containers can also manage runtime services, such as EJB and web component life cycles, data source pooling, data persistence, and JMS messaging.

There are two main types of containers within a Java EE application server:

- **Web containers:** Deploy and configure web components such as Servlets, JSP, JSF, and other web-related assets.
- **EJB containers:** Deploy and configure EJB, JPA, and JMS-related components. These types of deployments are described in detail in later chapters.

Where will the container look for things in the web app?

```
<servlet>
        <servlet-name>One</servlet-name>
        <servlet-class>foo.DeployTestOne</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>One</servlet-name>
        <url-pattern>*.do</url-pattern>
     </servlet-mapping>
     <servlet>
        <servlet-name>Two</servlet-name>
        <servlet-class>foo.DeployTestTwo</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Two</servlet-name>
        <url-pattern>/fooStuff/bar</url-pattern>
     </servlet-mapping>
        <servlet>
        <servlet-name>Three</servlet-name>
        <servlet-class>foo.DeployTestThree</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Three</servlet-name>
        <url-pattern>/fooStuff/*</url-pattern>
     </servlet-mapping>
```

Requests:

http://localhost:8080/MapTest/blue.do
Container choice:

http://localhost:8080/MapTest/fooStuff/bar
Container choice:

http://localhost:8080/MapTest/fooStuff/bar/blue.do
Container choice:

http://localhost:8080/MapTest/fooStuff/blue.do
Container choice:

http://localhost:8080/MapTest/fred/bar/blue.do
Container choice:

http://localhost:8080/MapTest/fooStuff
Container choice:

http://localhost:8080/MapTest/fooStuff/bar/foo.fo
Container choice:

http://localhost:8080/MapTest/fred/blue.fo
Container choice:

Key rules about servlet mappings

1) The Container looks first for an exact match. If it can't find an exact match, it looks for a directory match. If it can't find a directory match, it looks for an extension match.

2) If a request matches more than one directory <url-pattern>, the Container chooses the longest mapping. In other words, a request for /foo/car/myStuff.do will map to the <url-pattern> /foo/car/* even though it also matches the <url-pattern> /foo/*. The most specific match always wins.

- The answer to the above question would look like the one given in the below slide.

- **Mappings:**

```
<servlet>
        <servlet-name>One</servlet-name>
        <servlet-class>foo.DeployTestOne</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>One</servlet-name>
        <url-pattern>*.do</url-pattern>
</servlet-mapping>
<servlet>
        <servlet-name>Two</servlet-name>
        <servlet-class>foo.DeployTestTwo</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>Two</servlet-name>
        <url-pattern>/fooStuff/bar</url-pattern>
</servlet-mapping>
<servlet>
        <servlet-name>Three</servlet-name>
        <servlet-class>foo.DeployTestThree</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>Three</servlet-name>
        <url-pattern>/fooStuff/*</url-pattern>
</servlet-mapping>
```

Requests:

http://localhost:8080/MapTest/blue.do
Container choice: Deploy TestOne
(matched the *.do extension pattern)

http://localhost:8080/MapTest/fooStuff/bar
Container choice: Deploy TestTwo
(exact match with /fooStuff/bar pattern)

http://localhost:8080/MapTest/fooStuff/bar/blue.do
Container choice: Deploy TestThree
(matched the /fooStuff/* directory pattern)

http://localhost:8080/MapTest/fooStuff/blue.do
Container choice: Deploy TestThree
(matched the /fooStuff/* directory pattern)

http://localhost:8080/MapTest/fred/bar/blue.do
Container choice: Deploy TestOne
(matched the *.do extension pattern)

http://localhost:8080/MapTest/fooStuff
Container choice: Deploy TestThree
(matched the /fooStuff/* directory pattern)

http://localhost:8080/MapTest/fooStuff/bar/foo.fo
Container choice: Deploy TestThree
(matched the /fooStuff/* directory pattern)

http://localhost:8080/MapTest/fred/blue.fo
Container choice: 404 NOT FOUND
(doesn't match ANYTHING)

How the container chooses files?

- ***Refer to topic 3 for more information about Containers.***

JAVA EE 7 PROFILES

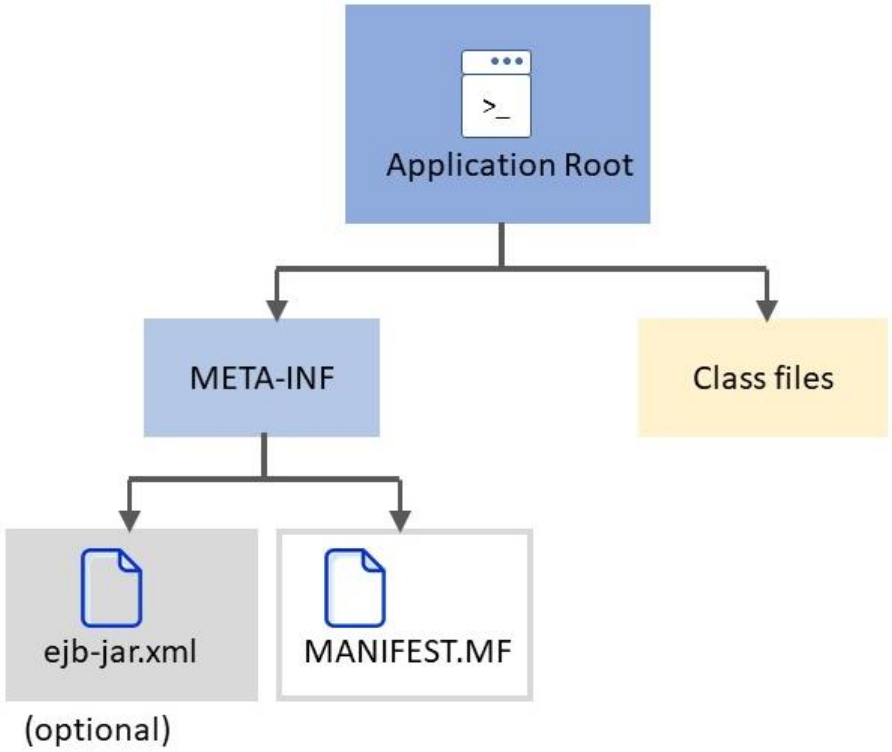
A profile in the context of a Java EE application server is a set of component APIs that target a specific application type. There are currently two profiles defined in Java EE 7:

- **Full Profile:** Contains all Java EE technologies, including all APIs in the web profile as well as others.
- **Web Profile:** Contains a full stack of Java EE APIs for developing dynamic web applications.

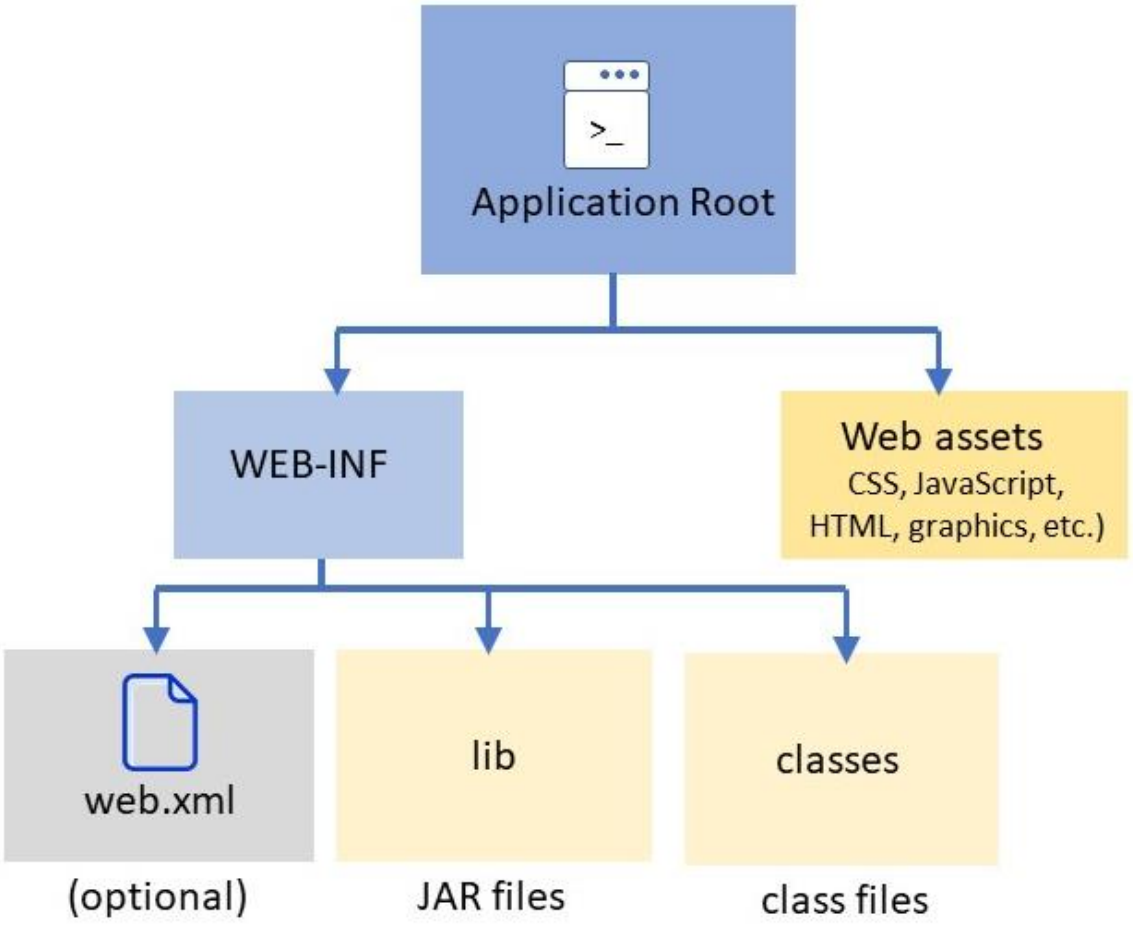
1.2 Packaging and Deploying a Java EE Application

- Java EE applications can be packaged in different ways for deployment to a compliant application server.
- Depending on the application type and the components it contains, applications can be packaged into different deployment types such as compressed archive files containing classes, application assets, and **XML deployment descriptors**.
- ***Refer to topic 3 to refresh your knowledge on the deployment descriptor.***
- The three most common deployment types are listed in upcoming slides:

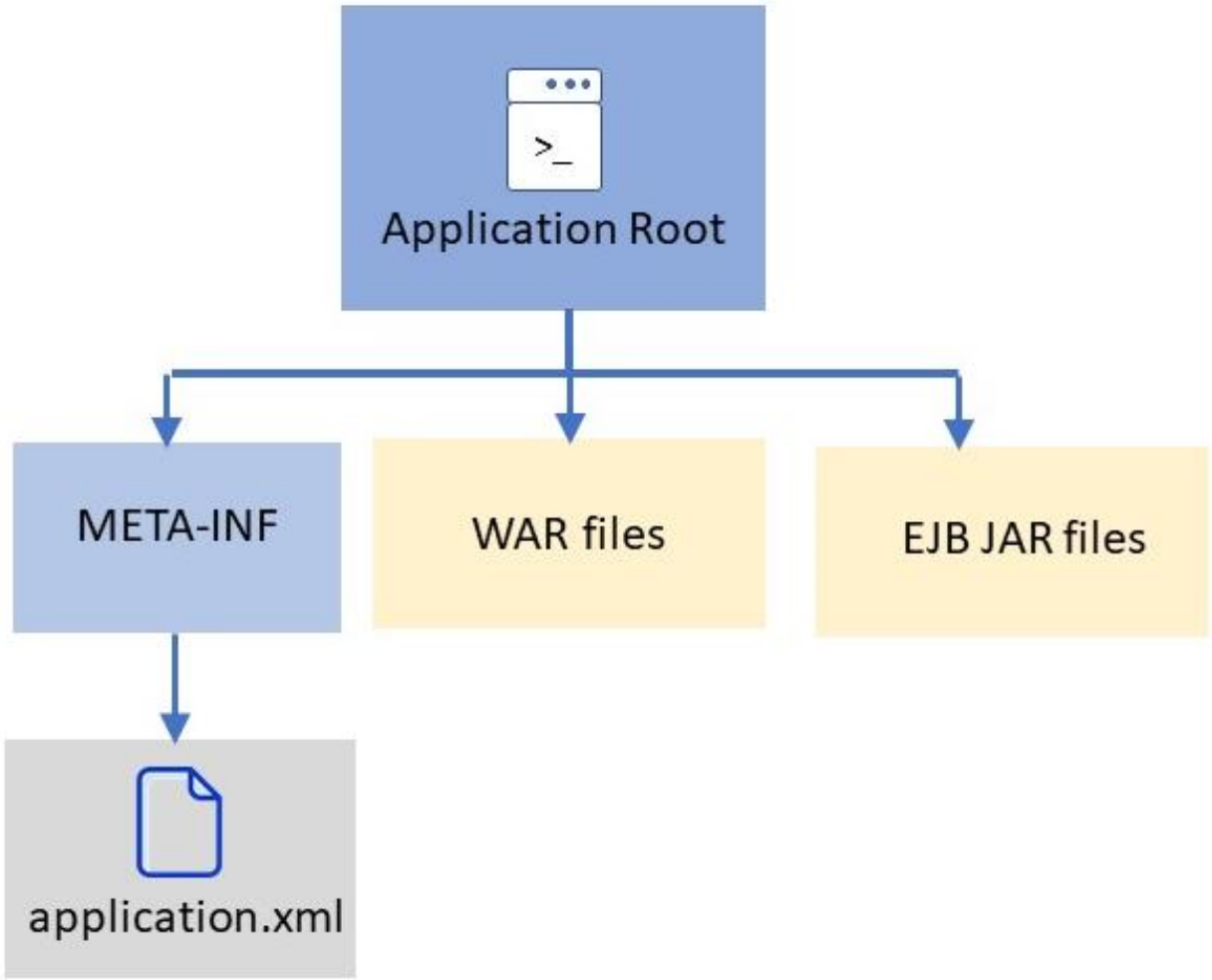
- **JAR files:** JAR files can contain Plain Old Java Object (POJO) classes, JPA Entity Beans, utility Java classes, EJBs, and MDBs.
- When deployed into an application server, depending on the type of components inside the JAR file, the application server looks for XML deployment descriptors, or code-level annotations, and deploys each component accordingly.



- **WAR files**: A WAR file is used for packaging web applications. It can contain one or more JAR files, as well as XML deployment descriptor files under the WEB-INF or WEB-INF/classes/META-INF folders.



- **EAR files:** An EAR file contains multiple JAR and WAR files, as well as XML deployment descriptors in the **META-INF** folder.



The ejb deploy command

- Before you can successfully run your enterprise application, you need to generate deployment code for the enterprise beans.

- Use the following command and the optional parameters, when the schema and map are provided in the input EAR or JAR file:

ejbdeploy input_EAR_name|input_JAR_name working_directory output_EAR_name|output_JAR_name [-bindear "options"] [-cp classpath] [-codegen] [-debug] [-keep] [-ignoreErrors] [-quiet] [-nowarn] [-noinform] [-rmic "options"][-trace] [-sqlj] [-outer] [-complianceLevel "1.4"|"5.0"|"6.0"|"7.0"][-DgenInheritancePerfEnhancement]

- Use the following command and the optional parameters, when the schema and map are not available in the input EAR or JAR file, and a top-down mapping approach is needed:

ejbdeploy input_EAR_name|input_JAR_name working_directory output_EAR_name|output_JAR_name [-bindear "options"] [-cp classpath] [-codegen] [-dbschema "name"] [-dbvendor name] [-debug] [-keep] [-ignoreErrors] [-quiet] [-nowarn] [-noinform] [-rmic "options"][-trace] [-sqlj][-OCCColumn] [-outer] [-complianceLevel "1.4"|"5.0"|"6.0"] [-DgenInheritancePerfEnhancement]

Parameters

- **ejbdeploy -** The command to generate deployment code.
- **input_JAR_name or input_EAR_name -** The fully qualified name of the input JAR or EAR file that contains the enterprise beans for which you want to generate deployment code.
- **working_directory -** The name of the directory where temporary files that are required for code generation are stored.
- **output_JAR_name or output_EAR_name -** The fully qualified name of the output JAR or EAR file that is created by the ejbdeploy command and that contains the generated classes required for deployment.
- **cp classpath -** If you intend to run the ejbdeploy command against JAR or EAR files that have dependencies on other .zip files or JAR files, you can use the -cp option to specify the class path of the other JAR or .zip files.

- **codegen** - Restricts the ejbdeploy command to the following tasks:
  - Importing code from the input JAR or EAR file
  - Generating the deployment code
  - Exporting code to the output JAR or EAR file
- **bindear "options" -** Enables you to populate an EAR file with bindings.
- **dbschema "name" -** The name of the schema you want to create.
- **dbvendor name -** The name of the database vendor, which is used to determine database column types, mapping information, Table.ddl, and other information.
- **debug -** Specifies that deployment code is compiled with debug information.

- **keep -** Controls the disposition of the temporary files that are created when the ejbdeploy command has run.
- **ignoreErrors -** Specifies that processing should continue even if validation errors are detected.
- **quiet -** During validation, suppresses status messages.
- **nowarn -** During validation, suppresses warning and informational messages.
- **noinform -** During validation, suppresses informational messages.
- **rmic "options" -** Enables you to pass RMIC options to RMIC.
- **trace -** Generates additional progress messages to the console.
- **sqlj -** Enables you to use SQLJ instead of JDBC to make calls to a DB2 database.

- **OCCColumn -** Enables you to add a column to your relational database table for collision detection.
- **outer -** This is an optional parameter and is only supported for deploying J2EE 1.3 applications.
- **complianceLevel "1.4 " | "5.0" | "6.0" -** Specify the Java developer kit (JDK) compiler compliance level to either 1.4, 5.0, or 6.0, if you have included application source files for compilation.
- **DgenInheritancePerfEnhancement true or false -** For EJB 1.x CMP beans using inheritance, use this parameter to optimize the number of SQL queries that the EJB container executes at runtime during bean hydration to improve the runtime performance of the application.

Example

- ejbdeploy AccessEmployee.ear d:\deploydir AccessEmployee_sqlj.ear -dbvendor DB2UDB_V95 -keep -sqlj -cp "e:\sqllib\java\sqlj.zip"

Packaging and deploying java EE applications to EAP

- A Java EE application is delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file.
- Using JAR, WAR, and EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components.
- No extra coding is needed.
- it is only a matter of assembling (or packaging) various Java EE modules into Java EE JAR, WAR, or EAR files.

The basic format of the command for creating a JAR file is:

- jar cf jar-file input-file(s)

- To create war file, you need to use jar tool of JDK. You need to use -c switch of jar, to create the war file. Go inside the project directory of your project (outside the WEB-INF), then write the following command:

    - jar -cvf projectname.war *

- Package the EJB's .jar file into an .ear for your application. To do this use a jar command similar to the following:

  - jar -cvf myApplication.ear myService.jar