

Application Client Guide

JEUS 9.1

TMAXSOFT

Copyright

Copyright 2025. TmaxSoft Co., Ltd. All Rights Reserved.

Company Information

TmaxSoft Co., Ltd.

TmaxSoft Tower, 45, Jeongjail-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, South Korea

Website: <https://www.tmaxsoft.com/en/>

Restricted Rights Legend

All TmaxSoft Software (JEUS®) and documents are protected by copyright laws and international convention. TmaxSoft software and documents are made available under the terms of the TmaxSoft License Agreement and this document may only be distributed or copied in accordance with the terms of this agreement. No part of this document may be transmitted, copied, deployed, or reproduced in any form or by any means, electronic, mechanical, or optical, without the prior written consent of TmaxSoft Co., Ltd. Nothing in this software document and agreement constitutes a transfer of intellectual property rights regardless of whether or not such rights are registered) or any rights to TmaxSoft trademarks, logos, or any other brand features.

This document is for information purposes only. The company assumes no direct or indirect responsibilities for the contents of this document, and does not guarantee that the information contained in this document satisfies certain legal or commercial conditions. The information contained in this document is subject to change without prior notice due to product upgrades or updates. The company assumes no liability for any errors in this document.

Trademarks

JEUS® is registered trademark of TmaxSoft Co., Ltd.

Java, Solaris are registered trademarks of Oracle Corporation and its subsidiaries and affiliates.

Microsoft, Windows, Windows NT are registered trademarks or trademarks of Microsoft Corporation.

HP-UX is a registered trademark of Hewlett Packard Enterprise Company.

AIX is a registered trademark of International Business Machines Corporation.

UNIX is a registered trademark of X/Open Company, Ltd.

Linux is a registered trademark of Linus Torvalds.

Noto is a trademark of Google Inc. Noto fonts are open source. All Noto fonts are published under the SIL Open Font License, Version 1.1. (<https://www.google.com/get/noto/>)

Other products and company names are trademarks or registered trademarks of their respective

owners.

The names of companies, systems, and products mentioned in this manual may not necessarily be indicated with a trademark symbol (™, ®).

Open Source Software Notice

Some modules or files of this product are subject to the terms of the following licenses: APACHE2.0, CDDL1.0, EDL1.0, OPEN SYMPHONY SOFTWARE1.1, TRILEAD-SSH2, Bouncy Castle, BSD, MIT, SIL OPEN FONT1.1

Detailed Information related to the license can be found in the following directory:
\${INSTALL_PATH}/license/oss_licenses

Document History

Product Version	Guide Version	Date	Remarks
JEUS 9.1	3.3.1	2025-12-10	-
JEUS 9	3.1.2	2025-03-24	-
JEUS 9	3.1.1	2024-12-24	-

Table of Contents

1. Application Clients	1
1.1. Overview	1
1.2. Programming	1
1.2.1. Program Architecture	1
1.2.2. Example	2
1.3. Deployment Descriptor(DD)	2
1.3.1. Writing a DD	3
1.3.2. Creating a DD	3
1.4. Packaging	4
1.5. Deployment	5
1.6. Execution	5
1.6.1. JEUS Libraries	5
1.6.2. Executing in Console	6
2. Advanced Application Clients	7
2.1. Overview	7
2.2. Dependency Injection	7
2.2.1. EJB Injection	8
2.2.2. Resource Injection	9
2.2.3. Other Injections	10
2.3. Clients Not Using Dependency Injection	10
2.4. Security Setting	11
2.5. Transaction	12
3. Client Applets	13
3.1. Overview	13
3.2. Programming	13
3.2.1. Example	13
3.2.2. HTML Example	14
3.3. Deployment	15
3.4. Execution	15
3.4.1. Executing in Web Browser	15
3.4.2. Executing in AppletViewer	15

1. Application Clients

This chapter discusses application clients that are executed on a separate JVM from the JEUS server (engine container).

1.1. Overview

In general, an application client module running in a JEUS client container is used to call a Jakarta EE application or be provided with Jakarta EE service.

An application client is a stand-alone client that runs in a Jakarta EE environment. It operates within the application client container, which is defined in Jakarta EE Specification. An application client module, a type of JEUS clients, is useful for client/server systems, testing, or debugging.

JEUS application client can use JEUS services, including naming service, scheduler, and security, by using a client container. Without using a client container, JNDI and security services can be used through the JEUS client library. However, dependency injection and JEUS scheduler cannot be used.



1. For more details about Jakarta EE based application client, refer to Jakarta EE Specifications.
2. For details about JEUS XML schema, refer to "jeus-client-dd.xml" in *JEUS XML Reference*.

1.2. Programming

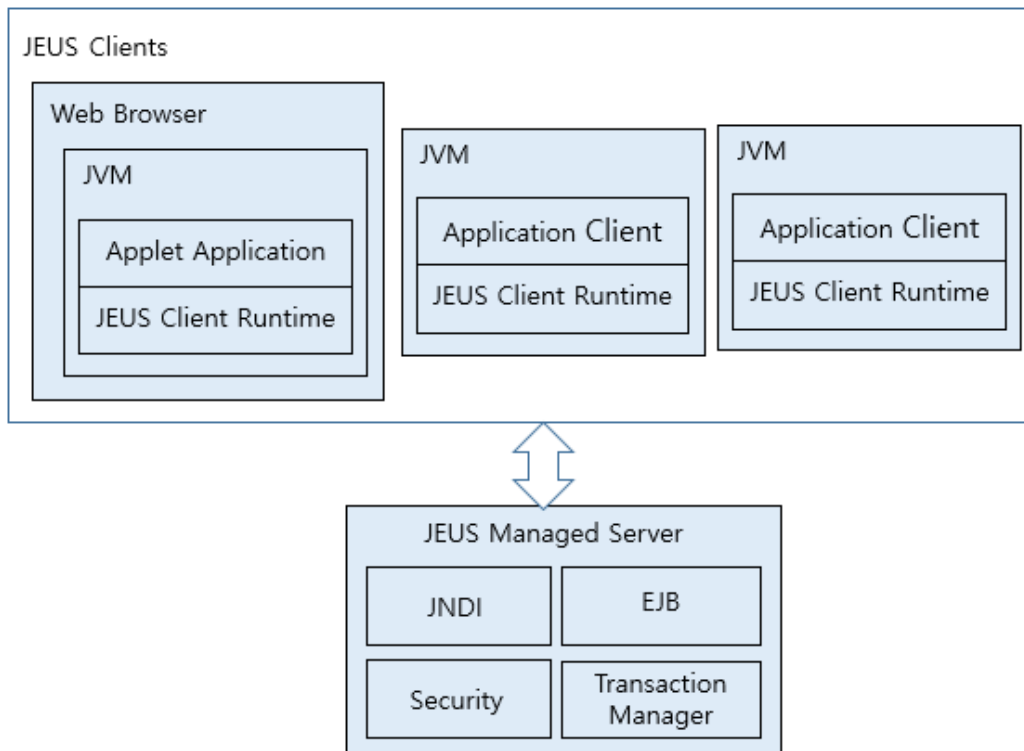
This section describes JEUS client and server architectures, and describes some examples using sample codes.

1.2.1. Program Architecture

An application client is a client program that runs on a JVM that is different from the server.

An application client calls and executes the `main()` method and runs until the JVM terminates. Like other Jakarta EE application components, an application client also runs in a client container that provides system services, but the container uses less system resources than other Jakarta EE containers. It provides the environment in which Jakarta EE services offered by JEUS can be provided for a general client application.

The services include scheduler, security, JNDI, and other services with which users can use the application components (EJB) and system resources (JDBC data source, JMS connection factory, etc.) that are bound to JEUS via JNDI.



Application Client Architecture

1.2.2. Example

Like other Java programs, an application client must have the main method that is declared in the public class.

The following example calls EJB via injection. (refer to the example of JEUSHOME/samples/client/hello/hello-client directory).

Application Client: <HelloClient.java>

```
package helloejb;

import java.io.*;
import jakarta.ejb.EJB;

public class HelloClient {
    @EJB(mappedName="helloejb.Hello")
    private static Hello hello;

    public static void main(String[] args) {
        System.out.println("EJB output : " + hello.sayHello());
    }
}
```

1.3. Deployment Descriptor(DD)

This section describes how to create a deployment descriptor (DD).

1.3.1. Writing a DD

A DD can be written in the following two ways:

- **Jakarta EE DD**

Jakarta EE Specification defines DD for client modules, and uses standard settings regardless of WAS.

Jakarta EE 5 and above do not require a standard DD file, and necessary settings can be done through annotations. Therefore, the previous example, [Application Client: <HelloClient.java>](#) can be run without any separate application-client.xml file.

For more information about standard XML descriptor, refer to Jakarta EE Specifications.

- **JEUS DD**

Information about the client module is required for a server to communicate with an application client. DD is an XML document that contains such information.

DD filename for a client is META-INF\jeus-client-dd.xml. Using a deployment descriptor, a user can decide which services should be provided when deploying each application client to the client container. DD can be modified without changes to the program.

In general, a deployment descriptor is used to configure a resource used by the client. However, the previous example [Application Client: <HelloClient.java>](#) does not require the DD file, thus it is omitted.



For more information about jeus-client-dd.xml, refer to *JEUS XML Reference*.

1.3.2. Creating a DD

If it is necessary to create the jeus-client-dd.xml file, you need to create it before JEUS deploys the client module.

The following example shows an XML file that creates a DD.

The XML file contains environment variables used by the client, a name to which the EJB application is bound, a name of the bound JDBC datasource, and a JNDI binding name for the JMS queue. Except for the environment variables, specify **<export-name>** as the JNDI name which is actually bound to **<ref-name>** in the application-client.xml file.

Creating a DD: <jeus-client-dd.xml>

```
<jeus-client-dd>
  <env>
    <name>year</name>
    <type>java.lang.Integer</type>
    <value>2002</value>
  </env>
```

```

<ejb-ref>
  <jndi-info>
    <ref-name>count</ref-name>
    <export-name>count_bean</export-name>
  </jndi-info>
</ejb-ref>
<res-ref>
  <jndi-info>
    <ref-name>datasource</ref-name>
    <export-name>Oracle_DataSource</export-name>
  </jndi-info>
</res-ref>
<res-env-ref>
  <jndi-info>
    <ref-name>jms/SalaryInfo</ref-name>
    <export-name>jms/salary_info_queue1</export-name>
  </jndi-info>
</res-env-ref>
</jeus-client-dd>

```



For a description of each element in the jeus-client-dd.xml file, refer to *JEUS XML Reference*.

1.4. Packaging

Client module packaging can be done by manual packaging or by packaging with IDE.

• Manual Packaging

If an XML file is necessary for a DD, you can create a DD XML file by using the XML editor or text editor that is installed on the user's computer. Then gather all the necessary files and create a JAR file for the client module by using the JAR tool provided by Java.

To package an application client, the application class files, and, in some cases, the **application-client.xml** and **jeus-client-dd.xml** files are required.

Use the **jar** command to create a JAR file for the client module in the console. By standard usage, it is possible to specify the main class, which is to be used when executing the JAR file, by using the main-class attribute in the MANIFEST.MF of the JAR file. In such cases, the JEUS client container automatically executes the class.

The following example shows how to create a JAR file using the jar command.

```
jar cvf hello-client.jar *
```

• Packaging with IDE

You can create using the IDE tool that supports Jakarta EE environment such as Eclipse,

NetBeans, or IntelliJ IDEA. Refer to each IDE's help for details.

1.5. Deployment

An application client module can be deployed "manually".

In the module, depending on the need, there are the Jakarta EE standard descriptor files, namely the application-client.xml file, and the jeus-client-dd.xml file, which is provided by JEUS. First, create a module file for the application client, and then move the file to a preferred location.

If there is a function that is controlled by the web service client, perform additional deployment via a console tool named jeusadmin, or use **appcompiler** to create a web service stub.



For more information about deployment, refer to *JEUS Applications & Deployment Guide*.

1.6. Execution

This section outlines JEUS libraries that are additionally required by each service and describes how to execute modules in the console.

1.6.1. JEUS Libraries

To use a web service by using an application client, additional libraries besides JEUS_HOME/lib/client/clientcontainer.jar (the default library) are required. Most of the libraries are located in JEUS_HOME/lib/system (SYSTEM_LIB_DIR).

The following is the list of JEUS libraries required for each service.

Service	JEUS Library
JMS(Java Message Service)	◦ SYSTEM_LIB_DIR/jms.jar
Web Service	◦ SYSTEM_LIB_DIR/jakarta.mail-2.0.1.jar ◦ JEUS_HOME/lib/shared/wsit-3.0/jeus-ws.jar ◦ JEUS_HOME/lib/shared/wsit-3.0/webservices-rt-3.0.1.jar ◦ SYSTEM_LIB_DIR/resolver.jar
JMX(Java Management eXtensions)	◦ SYSTEM_LIB_DIR/jmxremote.jar



For more information about the web service, refer to "JEUS Web Service Guide".

1.6.2. Executing in Console

To execute an application client module in the console, you can use the **appclient** command. The appclient script is located under the JEUS_HOME\bin directory, and executes an application client module through a client container.

The following are the command lines for JEUS client container.

- Usage

```
appclient -client client_jar_path
          [-main main_class]
          [-cp classpath]
          application_arguments...
```

The following describes the command options.

Option	Description
-client <i>client_jar_path</i>	Specifies the path of the application client to be executed.
[-main <i>main_class</i>]	Specifies the main class of the application client. This option is not required when main-class is already specified in the META-INF\MANIFEST.MF, located in the client's class path.
[-cp <i>classpath</i>]	Specifies a class path for executing the client, if needed.

- Example

If the sample command lines are executed, the following is displayed. For the application client to be successfully executed, the Hello EJB should be deployed first.

```
JEUS_HOME/samples/client/hello/hello-client/dist$ appclient -client hello-client.jar
[2016.08.03 14:44:46][0] [t-1] [CLIENT-0050] Starting the Application Client Container - JEUS 9.1
EJB output : Hello EJB!
```

If you want to turn off logging, use the following command:

```
-Djeus.log.level=OFF
```



For details about log setting, refer to "Logging" in *JEUS Server Guide*.

2. Advanced Application Clients

This chapter introduces advanced modules of an application client.

2.1. Overview

A JEUS client can be executed as simply as a Java class is executed without using a client container described in the Jakarta EE specifications. This chapter discusses the types and JNDI default binding names of resources, for which a client container performs dependency injection. It also describes security and transaction services that can be used by application clients and clients that run without a client container.

If you execute a JEUS client without using a client container, the client cannot use the dependency injection service of the client container. With security settings, you can execute JEUS's various services. With transaction functions, you can manage the resources and applications as global transactions.

2.2. Dependency Injection

This section describes injection details common to application clients, web applications, and EJB applications.

Injection can be performed for resources such as an environment variable, which can be mapped to an EJB object. In general, a resource's name can be found in `java:comp/env` context, a JNDI context of the application component. Because the actual resource is bound to the JNDI global context, you need to know the global binding name.

Resources have their own JNDI global binding names. The JNDI global binding name of an EJB application should be specified by one of the following methods:

- `<export-name>` of `jeus-ejb-dd.xml`
- `<mapped-name>` of `ejb-jar.xml` in the standard
- `mappedName` of an annotation specified in the EJB application
- EJB is bound to the default JNDI name of JEUS specified in *JEUS EJB Guide*.

To obtain an EJB through JNDI without a client container, it is necessary to understand the rules for setting the default JNDI name. Because it is hard for a developer to know to which name the EJB will be bound to, it is recommended to specify a JNDI binding name using either of the suggested ways.

To inject a resource, specify a JNDI global binding name in `jeus-ejb-dd.xml`, `jeus-web-dd.xml`, and `jeus-client-dd.xml`, which are JEUS DD files. You can also map the resource to the value specified by `mapped-name` in `ejb-jar.xml`, `web.xml` or `application-client.xml`, or to the `mappedName` in the annotation. If nothing is specified, use a name according to the basic rules of JEUS.

In the actual development, it is recommended to use XML in specifying the JNDI global binding

name, rather than using the mappedName of the annotation. If the application will be executed in many places, XML should be used since it needs to use a global name according to the environment. Injection is normally done for annotated setter methods and variables. However, it is possible to perform injection for non-annotated setter methods and variables, if they are specified by XML descriptor.



1. For more information about injection, refer to [Jakarta EE 9 Platform](#), and see "5. Resources, Naming and Injection" section of this guide about resources that can be injected.
2. For EJBContext injection for EJB applications, refer to "JEUS EJB Guide".

The following example shows how a client injects an EJB application using a mappedName of the annotation. Since statelessEJB1 application uses 'MyEJB1' as its JNDI global binding name, the client specifies the same name for mappedName of the @EJB annotation. If the client uses JNDI Lookup instead of Injection, it is possible to directly use the JNDI global binding name for lookup. If the client runs in a client container, it is possible to use the name, java:comp/env/ejb/sless1, from the application context using the application-client.xml file.

If the client uses JNDI Lookup instead of Injection, it is possible to directly use the JNDI global binding name for lookup. If the client runs in a client container, it is possible to use the name, java:comp/env/ejb/sless1, from the application context using the application-client.xml file.

EJB Reference Injection

```
import ejb1.RemoteSession;

@Stateless(name="StatelessEJB1", mappedName="MyEJB1")
public class StatelessEJB1 implements RemoteSession, LocalSession {...
}
...

@EJB(name="sless1", beanName="StatelessEJB1", mappedName="MyEJB1")
private RemoteSession sless1;
...
RemoteSession session = context.lookup("MyEJB1");
...
// with client container and application-client.xml descriptor
RemoteSession session = context.lookup("java:comp/env/ejb/sless1");
```

2.2.1. EJB Injection

For EJB reference injection, the following binding names are used.

- **If the variable is a business interface:**

If a mappedName is specified, the global name used for lookup should be the following format:

```
mappedName + "#" + Business_Interface_Name
```

In the previous example, the name will be set to MyEJB1 or MyEJB1#RemoteSession. If deployed as EAR or EJB JAR file and if ejb-link is given to the ejb-jar.xml file or if there is a beanName in the annotation, find an EJB in the same application and use its mappedName as the global name for lookup. Otherwise, find an EJB in the application with a business interface name, and use the EJB's mappedName as the global name.

Finally, perform JNDI Lookup with the business interface name. In the previous example, the name, 'java:global/<module-name>/MyEJB1' or 'java:global/<module-name>/MyEJB1#ejb1.RemoteSession,' is used for Lookup. In this way, use a default binding name when there is no mappedName for EJB deployment.

- **If the variable is a sub interface of EJBHome/EJBObject interface:**

If there is a mappedName, use it as the global name. If deployed as EAR or EJB JAR file and if ejb-link is given to the ejb-jar.xml file or if there is a beanName in the annotation, find an EJB in the same application and use its mappedName as the global name for lookup. Otherwise, find an EJB in the application with a variable type interface name, and use the EJB's mappedName as the global name.

Finally, perform JNDI Lookup with the variable type interface name.

2.2.2. Resource Injection

It is possible to use the @Resource annotation for resources.

- If a mappedName is specified, use it as the resource's JNDI global binding name for lookup.
- If not specified, use the name value of @Resource as the JNDI global binding name.

If no name value is specified, the following format is used according to the specifications.

Application class name + / + Variable or setter method's property name

In the following example, the name 'jdbc/DB2' is used for JNDI Lookup. If no name is specified, the name 'test.Client/myDataSource3' is used for lookup.

Resource injection

```
package test;

class Client {
    @Resource(name="jdbc/DB2") // default mapping if no mapped-name private
    javax.sql.DataSource myDataSource3;
    ...
}
```



When there is a name specified or a default value is specified for a name attribute that is not set, the value is mapped to the application context. However, for the

actual JNDI global binding name different rules apply for each vendor. Thus for compatibility reasons, the mappedName should be used.

2.2.3. Other Injections

In addition, it is possible to obtain a web service object, an EntityManager object, and an EntityManagerFactory object through Injection, by using annotations like @WebServiceRef, @PersistenceUnit and @PersistenceContext.



For further details, refer to [Jakarta EE Platform Specification](#).

2.3. Clients Not Using Dependency Injection

You can use JEUS resources and applications through JEUS JNDI, without using an application container. To do so, execute a client program whose class path is set to the jclient.jar file in the JEUS_HOME\lib\client directory.

Because it is impossible to use dependency injection in this case, you need to modify the sources as shown in the following example. In the example, a binding name of the EJB application should follow JEUS's default name binding rule. The client can run on a client container. This means that all clients which do not use a client container can run on a client container.

Clients Not Using Dependency Injection: <HelloClient.java>

```
package helloejb;

import java.io.*;
import jakarta.ejb.EJB;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

/**
 * HelloEJB application client
 */
public class HelloClient {
    private static Hello hello;

    public static void main(String[] args) {
        try {
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY, "jeus.jndi.JNSContextFactory");
            Context context = new InitialContext(env);
            hello = (Hello) context.lookup("helloejb.Hello");

            System.out.println("EJB output : " + hello.sayHello());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

```
}
```

Client packaging is not performed in a client container that processes XML files. Thus, you need to create a JAR file without the standard or JEUS XML file. For deployment, copy the JAR file to any location. For execution, execute the previous client class just like executing a general Java class.

The result is as follows:

```
$ java -cp /jeus/lib/client/jclient.jar;./hello-client.jar helloejb.HelloClient
[2012.05.23 22:45:51][2] [t-1] [Network-0405] [Endpoint] exporting Endpoint (0:192.168.0.16:9756:-
1:0x79F24F28)
[2012.05.23 22:45:51][2] [t-1] [Network-0002] [Acceptor] start to listen NonBlockingChannelAcceptor:
/192.168.0.16:9756
EJB output : Hello EJB!
```

2.4. Security Setting

To use JEUS's Jakarta EE services, a client has to provide its username and password to the client runtime in order to verify authorization for using the services. The services include EJB applications and JMS resources. The following are three ways to specify a username and password:

- **Using jeus-client-dd.xml**

If **<security-info>** in the jeus-client-dd.xml file is specified, a user has to sign in with the specified username and password before starting an application in the client container. After this, the application attempts authentication using this username when using JEUS services. For more information about the jeus-client-dd.xml settings, refer to *JEUS XML Reference*.

Security Setting: <jeus-client-dd.xml>

```
<jeus-client-dd>
...
  <security-info>
    <provider-node-name>jeusNode</provider-node-name>
    <user>user1</user>
    <passwd>password1</passwd>
  </security-info>
  ...
</jeus-client-dd>
```

- **Using the JNDI context**

When a client creates the JNDI context for an application, the client can use the JNDI properties to sign in with a username and password, which are used for authentication. This method of providing account information is available even without a client container. For the detailed information about the JNDI settings, refer to "JNDI Naming Server" in *JEUS Server Guide*.

Security Setting: <Client.java>

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "jeus.jndi.JNSContextFactory");
env.put(Context.PROVIDER_URL, "192.168.0.16:9736");
env.put(Context.SECURITY_PRINCIPAL, "user1");
env.put(Context.SECURITY_CREDENTIALS, "password1");
Context context = new InitialContext(env);
```

- **Using JEUS Security API**

You can log in using JEUS's Security API. For more information about the Security API, refer to *JEUS Security Guide*.

2.5. Transaction

UserTransaction is used to utilize EJB client applications, JDBC DataSource, JMS connection factory and destination, and more as one global transaction or an XA transaction.

The transaction manager used by a client is either a server transaction manager or client transaction manager depending on whether it has direct control over resources.



1. For detailed description of UserTransaction, refer to [Jakarta Transaction 2.0 Specification](#).
2. For detailed explanation of the transaction manager, refer to "Transaction Manager" in *JEUS Server Guide*.

3. Client Applets

This chapter describes how to create, configure, and execute a JEUS applet program.

3.1. Overview

An applet is a Java application that is executed in a web browser.

To use a Jakarta EE service in an applet, an applet container should be used. Because lightweight client containers are not provided yet, an applet that cannot directly access JEUS libraries is allowed to use a Jakarta EE service without a container.

With an applet, a user can execute Java applications in the web browser. An applet can be run as a JEUS client as well.

3.2. Programming

Files required to run an applet exist within a web application. Since the HTML `JAVA_CODEBASE` is the URL directory containing the code file, JAR files should exist in the same directory where the HTML document of the web application will be deployed to.

This section shows sample examples.

3.2.1. Example

An applet should inherit the `Applet` or `JApplet` class and implement the `start()` method.

The following example shows a client without a client container. In this case, the client does not use dependency injection but JNDI API to look up an EJB.

In the example, the same EJB as the client container in the previous chapter is used, with the interface name, 'helloejb.Hello,' as its binding name.

Applet Application: <HelloClient.java>

```
package helloejb;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.applet.Applet;
import java.util.Hashtable;

import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.event.*;

import javax.swing.*;
```

```

public class HelloClient extends JApplet {
    public void start() {
        try {
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY, "jeus.jndi.JNSContextFactory");
            Context context = new InitialContext(env);
            Hello hello = (Hello) context.lookup("helloejb.Hello");
            System.out.println("EJB output : " + hello.sayHello());

            JLabel label = new JLabel(hello.sayHello());
            label.setFont(new Font("Helvetica", Font.BOLD, 15));
            getContentPane().setLayout(new BorderLayout());
            getContentPane().add(label, BorderLayout.CENTER);
            setSize(500, 250);
            setVisible(true);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```



For more information about EJB JNDI binding name, refer to *JEUS EJB Guide*.

3.2.2. HTML Example

In an HTML document, users can call a certain applet and specify the location of the applet's classes.

In the following example, the application class and helloejb.Hello EJB interface in the previous example are included in the hello-client.jar file. The jclient.jar is a JEUS client library, and it exists in the JEUS_HOME\lib\client directory.

HTML Example: <index.html>

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Hello JakartaEE</title>
</head>
<body>
<center>
<h1>Hello JakartaEE Sample Applet!</h1>
<APPLET CODE = "helloejb.HelloClient" JAVA_CODEBASE = "."
    ARCHIVE = "hello-client.jar,jclient.jar"
    WIDTH = 300 HEIGHT = 300/>
</APPLET>
</center>
</body>
</html>

```



If an HTML document is used in any browser while JDK is not installed at the location where the browser is to be executed, you can prompt the user to install

JDK. To convert to an HTML document, use JDK's `htmlconverter`. For more information, refer to [Java Plug-in HTML Converter](#).

3.3. Deployment

Because an applet generally runs from HTML, running an applet in a browser requires a web application that passes an HTML document containing the applet to the browser. Therefore, it is necessary to complete the deployment of the web application and EJB before executing an applet.

- **Web Application Deployment**

Create a web application and add the HTML document and the JAR files specified in 'ARCHIVE'. For more information about creating and deploying a web application, see *JEUS Application & Deployment Guide*.

- **EJB Deploy**

After deploying the web application, deploy the EJB application.

3.4. Execution

Open the web browser, enter the URL of the HTML document, and run the applet. Following the Java Security model, an applet performs access control as specified in the `java.policy` file. Therefore, in the file, the classes that the applet uses must be given permission.



For information about the configuration settings in `java.policy`, refer to [Security Developer's Guide](#).

3.4.1. Executing in Web Browser

Access the web browser by using the `<applet>` tag in the HTML page. The applet in the example uses Swing, and so the following address is used to access the applet.

```
http://host1:8088/hello/index.html
```

3.4.2. Executing in AppletViewer

For testing, you can execute an applet by using the AppletViewer included in the JDK, rather than the browser. Executing an applet by using the AppletViewer is more convenient for testing during development because users can immediately check for exceptions.

appletviewer index.html