

Application Client 안내서

JEUS 9

TMAXSOFT

저작권 공지

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

회사 정보

(주)티맥스소프트

주소 : 경기도 성남시 분당구 황새울로258번길 29, 티맥스수내타워 8-9층

기술 서비스 센터: 1544-8629

홈페이지: <https://www.tmaxsoft.com>

제한된 권리

이 소프트웨어(JEUS®) 사용설명서와 프로그램은 저작권법과 국제 조약에 의해 보호됩니다. 사용설명서와 프로그램은 TmaxSoft Co., Ltd.와의 사용권 계약 하에서만 사용할 수 있으며, 사용설명서는 사용권 계약의 범위 내에서만 배포 또는 복제할 수 있습니다. 이 사용설명서의 전부 또는 일부를 TmaxSoft의 사전 서면 동의 없이 전자, 기계, 녹음 등의 수단으로 전송, 복제, 배포하거나 2차적 저작물을 작성할 수 없습니다.

이 소프트웨어 사용설명서와 프로그램의 사용권 계약은 어떠한 경우에도 사용설명서 및 프로그램과 관련된 지적 재산권(등록 여부를 불문)을 양도하는 것으로 해석되지 않으며, 브랜드나 로고, 상표 등을 사용할 권한을 부여하지 않습니다. 사용설명서는 오로지 정보 제공만을 목적으로 하며, 이로 인한 계약상의 직접적 또는 간접적 책임을 지지 않습니다. 또한 사용설명서 상의 내용이 법적 또는 상업적인 특정 조건을 만족시킬 것을 보장하지 않습니다. 사용설명서는 제품의 업그레이드나 수정에 따라 예고 없이 변경될 수 있으며, 내용상의 오류가 없음을 보장하지 않습니다.

상표 공지

JEUS®는 TmaxSoft Co., Ltd.의 등록 상표입니다.

Java, Solaris는 Oracle Corporation 및 그 자회사, 관계회사의 등록 상표입니다.

Microsoft, Windows, Windows NT는 Microsoft Corporation의 등록 상표 또는 상표입니다.

HP-UX는 Hewlett Packard Enterprise Company의 등록 상표입니다.

AIX는 International Business Machines Corporation의 등록 상표입니다.

UNIX는 X/Open Company, Ltd.의 등록 상표입니다.

Linux는 Linus Torvalds의 등록 상표입니다.

Noto는 Google Inc.의 상표입니다. Noto 글꼴은 오픈 소스입니다. 모든 Noto 글꼴은 SIL Open Font License, 버전 1.1에 따라 게시됩니다. (<https://www.google.com/get/noto/>)

기타 모든 제품들과 회사 이름은 각각 해당 소유주의 상호, 상표, 또는 등록 상표입니다.

본 사용설명서에 기재된 회사, 시스템, 제품 이름 등에 반드시 상표 표시(™, ®)를 하지는 않습니다.

오픈 소스 소프트웨어 공지

본 제품의 일부 파일 또는 모듈은 다음의 라이선스를 준수합니다.: APACHE2.0, CDDL1.0, EDL1.0, OPEN SYMPHONY SOFTWARE1.1, TRILEAD-SSH2, Bouncy Castle, BSD, MIT, SIL OPEN FONT1.1

관련 상세 정보는 제품의 다음 디렉터리에 기재된 사항을 참고하시기 바랍니다. :

`${INSTALL_PATH}/license/oss_licenses`

유지 보수

| 구분 | 지원항목 | 서비스 내용 |
|----------------|-----------------|--|
| 제품지원 | 패치 & 업그레이드 | 무상 패치 서비스 제공 |
| | | 메이저 버전 업그레이드 시 할인 혜택 |
| | | 웹 지원을 통한 패치 내역 제공 |
| 기술 지원 - 기본 서비스 | 장애 지원 | 장애 발생 시 원인 분석 및 조치 Service Desk팀 → 기술팀 → R&D의 3단계 장애 분석 및 조치 |
| | 일상 지원(온라인 지원) | E-mail, 전화, 원격, 웹 사이트 등 온라인 자원을 통한 질의 응답 서비스 |
| | 고객 맞춤 지원(방문 지원) | 고객의 요청으로 수행하는 방문 지원 서비스 |
| 기술 지원 - 옵션 서비스 | 예방 지원 | 정기 점검을 통한 시스템 운영현황 보고 및 장애 예방 ◦ 관리자 또는 운영자의 요구사항 수렴 ◦ 운영 현황(시스템, 엔진 운영) 보고서 제공 ◦ 필요 시 시스템 개선 권장 사항 보고 |
| 유지 보수 비용 및 기간 | 계약 시 별도 협의 | 계약 시 EOL/EOS 문서 제공 |

안내서 이력

| 제품 버전 | 안내서 버전 | 발행일 | 비고 |
|--------|--------|------------|----|
| JEUS 9 | 3.1.2 | 2025-01-03 | - |
| JEUS 9 | 3.1.1 | 2024-09-27 | - |

목차

| | |
|---|----|
| 1. 애플리케이션 클라이언트 | 1 |
| 1.1. 개요 | 1 |
| 1.2. 프로그램 작성 | 1 |
| 1.2.1. 프로그램 구조 | 1 |
| 1.2.2. 예제 | 2 |
| 1.3. Deployment Descriptor(DD) | 2 |
| 1.3.1. DD 작성 | 3 |
| 1.3.2. DD 생성 | 3 |
| 1.4. 패키징 | 4 |
| 1.5. Deploy | 4 |
| 1.6. 실행 | 5 |
| 1.6.1. JEUS 라이브러리 | 5 |
| 1.6.2. 콘솔에서 실행 | 5 |
| 2. 고급 애플리케이션 클라이언트 | 7 |
| 2.1. 개요 | 7 |
| 2.2. Dependency Injection | 7 |
| 2.2.1. EJB Injection | 8 |
| 2.2.2. 리소스 Injection | 9 |
| 2.2.3. 그 외의 Injection | 10 |
| 2.3. Dependency Injection을 사용할 수 없는 클라이언트 | 10 |
| 2.4. 보안 설정 | 11 |
| 2.5. 트랜잭션 | 12 |
| 3. 애플릿 클라이언트 | 13 |
| 3.1. 개요 | 13 |
| 3.2. 프로그램 작성 | 13 |
| 3.2.1. 애플릿 예제 | 13 |
| 3.2.2. HTML 예제 | 14 |
| 3.3. Deploy | 15 |
| 3.4. 실행 | 15 |
| 3.4.1. 웹 브라우저에서 실행 | 15 |
| 3.4.2. 애플릿 뷰어에서 실행 | 15 |

1. 애플리케이션 클라이언트

본 장에서는 JEUS 서버와는 별도의 JVM에서 수행되는 애플리케이션 클라이언트에 대해서 설명한다.

1.1. 개요

일반적으로 Jakarta EE 애플리케이션을 호출하거나 Jakarta EE의 서비스를 제공받기 위해서는 JEUS의 클라이언트 컨테이너에서 구동되는 애플리케이션 클라이언트(Application Client) 모듈을 사용한다.

애플리케이션 클라이언트는 Jakarta EE 환경을 사용하는 standalone 클라이언트로 Jakarta EE 스펙에 정의되어 있는 애플리케이션 클라이언트 컨테이너(Application Client Container)에서 구동되는 애플리케이션이다. 애플리케이션 클라이언트 모듈은 JEUS 클라이언트의 한 형태이다. 이 형태는 클라이언트 또는 서버 시스템이나 테스트 및 디버깅을 하는 경우 유용하게 사용할 수 있는 클라이언트이다.

JEUS 애플리케이션 클라이언트는 클라이언트 컨테이너를 사용해서 Naming Service, Scheduler, Security 등과 같은 JEUS 서비스를 사용할 수 있다. 클라이언트 컨테이너를 사용하지 않더라도 JEUS 클라이언트 라이브러리를 사용하면 JNDI, Security 등의 일부 서비스를 사용할 수는 있지만 Dependency Injection, JEUS Scheduler 등에 해당하는 서비스는 사용할 수 없다.



1. Jakarta EE 기반의 애플리케이션 클라이언트의 자세한 내용은 Jakarta EE 스펙을 참고한다.
2. JEUS XML 스키마의 자세한 내용은 "JEUS XML Reference"의 "jeus-client-dd.xml"을 참고한다.

1.2. 프로그램 작성

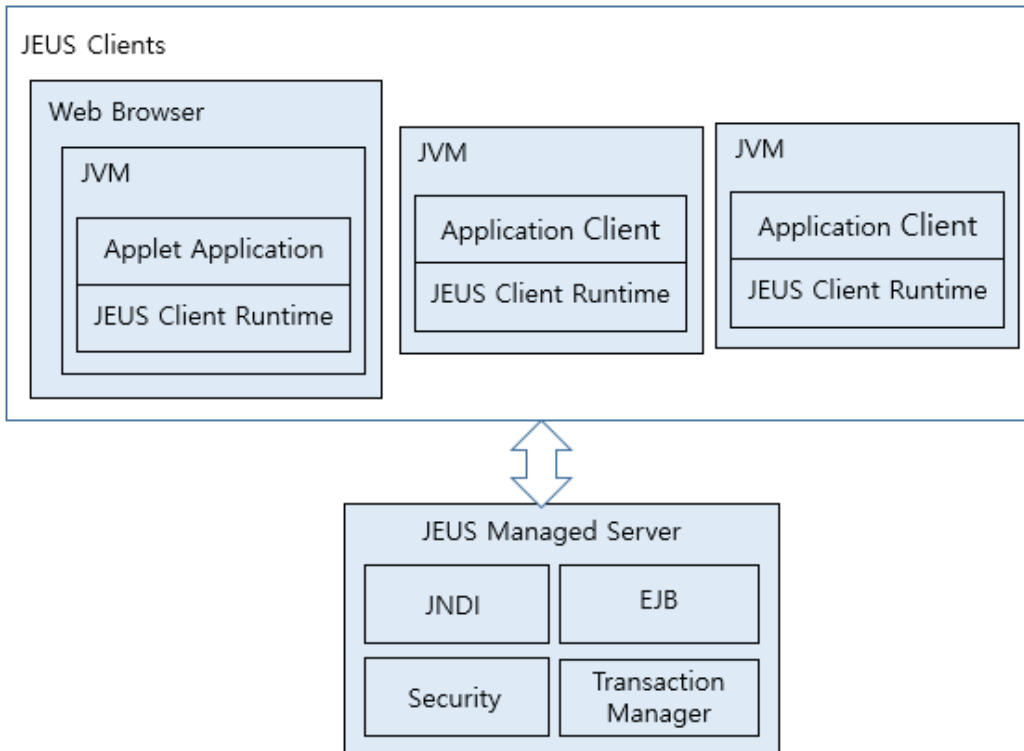
본 절에서는 JEUS 클라이언트와 서버 환경의 아키텍처에 대해 알아보고, 간단한 샘플 코드를 예로 들어 설명한다.

1.2.1. 프로그램 구조

애플리케이션 클라이언트는 서버와는 다른 JVM에서 실행되는 클라이언트 프로그램이다.

애플리케이션 클라이언트는 main() 메소드를 호출해서 실행하고, 가상 머신이 종료되면 실행을 마친다. 다른 Jakarta EE 애플리케이션 컴포넌트처럼 애플리케이션 클라이언트는 시스템 서비스를 제공하는 클라이언트 컨테이너에서 동작한다. 클라이언트 컨테이너는 다른 Jakarta EE 컨테이너에 비해서 적은 양의 시스템 리소스를 사용한다. 클라이언트 컨테이너는 일반 애플리케이션 클라이언트에서 JEUS가 제공하는 Jakarta EE 서비스를 사용할 수 있는 환경을 제공한다.

이 서비스에는 JNDI, Scheduler, Security 그리고 JNDI 서비스를 통해서 JEUS에 바인딩되어 있는 애플리케이션 컴포넌트(EJB) 및 시스템 리소스들(JDBC DataSource, JMS Connection Factory 등)에 대한 사용이 포함된다.



애플리케이션 클라이언트 아키텍처

1.2.2. 예제

애플리케이션 클라이언트는 일반적인 Java 프로그램과 마찬가지로 반드시 public으로 선언된 main() 메소드를 가지고 있어야 한다.

다음 예제는 Injection을 통해 EJB를 호출한다(JEUS_HOME/samples/client/hello/hello-client 예제를 참고).

애플리케이션 클라이언트 : <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)

본 절에서는 Deployment Descriptor(이하 DD)를 작성하고 생성하는 방법에 대해 설명한다.

1.3.1. DD 작성

DD 작성 방법은 다음과 같이 2가지로 나뉘어진다.

- Jakarta EE DD

Jakarta EE 스펙에서는 클라이언트 모듈에 대한 DD를 정의하고 있는데, 이는 Web Application Server(WAS)에 관계없이 표준적인 설정을 담고 있다.

표준 DD 파일 대신, Annotation을 통해 설정할 수 있다. 따라서 위의 [애플리케이션 클라이언트 : <HelloClient.java>](#)도 별도의 application-client.xml이 없이도 동작한다.

표준 XML Descriptor에 대한 자세한 설명은 Jakarta EE 스펙을 참고한다.

- JEUS DD

서버와 애플리케이션 클라이언트가 통신할 때 클라이언트 모듈에 대한 정보가 필요한데, DD는 이러한 정보들을 가지고 있는 XML 설정이다.

클라이언트를 위한 DD는 META-INF\jeus-client-dd.xml이다. DD를 사용하여 각각의 애플리케이션 클라이언트를 클라이언트 컨테이너에 디플로이할 때에 어떤 서비스를 사용할지 결정할 수 있다. 또한 실행하는 경우에도 프로그램의 별도의 수정 없이 DD만 수정하여 해당하는 애플리케이션 클라이언트에게 적용할 수 있다.

일반적으로 클라이언트가 사용하는 리소스 등을 지정하기 위해 사용하는데, 위의 [애플리케이션 클라이언트 : <HelloClient.java>](#)에 대해서는 이 DD를 사용하지 않아도 되므로 생략한다.



jeus-client-dd.xml에 대한 자세한 설명은 "JEUS XML Reference"를 참고한다.

1.3.2. DD 생성

DD를 생성할 때 필요하다면 JEUS에서 클라이언트 모듈을 디플로이하기 전에 jeus-client-dd.xml 파일을 작성한다.

다음은 DD를 작성한 XML 파일의 예이다.

XML에서 클라이언트가 사용하는 환경변수, EJB 애플리케이션이 바인딩된 이름, 바인딩된 JDBC DataSource의 이름, JMS Queue의 JNDI 바인딩 이름을 지정하고 있다. 환경변수를 제외하고는 application-client.xml의 **<ref-name>**에 실제 바인딩된 JNDI 이름을 **<export-name>**으로 지정한다.

DD 생성 : <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>
</jeus-client-dd>
```

```

</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>

```



jeus-client-dd.xml 파일의 각 element에 대한 설명은 "JEUS XML Reference"를 참고한다.

1.4. 패키징

클라이언트 모듈의 패키징 방식은 수동 패키징 방식과 IDE를 사용한 패키징 방식으로 나뉜다.

• 수동 패키징

필요하다면 사용자의 컴퓨터에 설치되어 있는 텍스트 에디터나 XML 에디터를 사용해서 DD XML 파일을 생성하고, 필요한 파일을 모아서 Java에서 제공하는 JAR 툴을 사용하여 클라이언트 모듈에 대한 JAR 파일을 생성한다.

애플리케이션 클라이언트를 패키징하려면 애플리케이션을 구성하고 있는 클래스 파일과 필요하다면 **application-client.xml**, **jeus-client-dd.xml** 파일이 포함되어 있어야 한다.

콘솔에서는 **jar** 명령어를 사용하여 클라이언트 모듈에 대한 JAR 파일을 생성한다. 표준에서는 JAR 파일의 MANIFEST.MF에 Main-Class 속성으로 이 JAR 파일이 실행될 때 사용할 메인 클래스를 지정할 수 있다. 이 경우 JEUS 클라이언트 컨테이너는 메인 클래스를 알려주지 않아도 자동으로 이 클래스를 실행한다.

다음은 jar 명령어를 사용하여 JAR 파일을 생성하는 예이다.

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

• IDE를 사용한 패키징

Eclipse나 NetBeans, IntelliJ IDEA와 같이 Jakarta EE 환경을 지원하는 IDE 툴에서 생성한다. 이 방법은 각 IDE의 도움말을 참고한다.

1.5. Deploy

애플리케이션 클라이언트 모듈은 수동으로 직접 deploy를 진행한다.

모듈에는 필요에 따라 Jakarta EE 표준 DD 파일인 application-client.xml과 JEUS에서 제공하는 jeus-client-dd.xml이 있다. 애플리케이션 클라이언트에 대한 모듈 파일을 생성한 후 해당 파일을 원하는 위치로 이동한다.

만약 웹 서비스 클라이언트로 동작하는 기능이 있다면 추가로 콘솔 툴(jeusadmin)을 통해 deploy하거나 **appcompiler** 명령어를 사용해야 웹 서비스 Stub이 생성된다.



Deploy에 대한 자세한 내용은 "JEUS Applications & Deployment 안내서"를 참고한다.

1.6. 실행

본 절에서는 각 서비스별로 추가적으로 필요한 JEUS 라이브러리와 콘솔에서 모듈을 실행하는 방법에 대해 설명한다.

1.6.1. JEUS 라이브러리

애플리케이션 클라이언트로 웹 서비스를 사용할 경우 라이브러리 실행 단계에서 기본 라이브러리인 JEUS_HOME/lib/client/clientcontainer.jar 외에 추가적으로 필요한 라이브러리들이 존재한다. 이들은 대부분 JEUS_HOME/lib/system(이하 SYSTEM_LIB_DIR) 아래에 위치한다.

다음은 서비스별 추가적으로 필요한 JEUS 라이브러리 목록이다.

| 서비스 | JEUS 라이브러리 |
|---------------------------------|---|
| 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 |



웹 서비스에 대한 자세한 내용은 "JEUS Web Service 안내서"를 참고한다.

1.6.2. 콘솔에서 실행

콘솔에서 모듈을 실행하기 위해서 **appclient** 명령어를 사용한다. appclient는 JEUS_HOME\bin에 존재하는 스크립트로 클라이언트 컨테이너를 통해 애플리케이션 클라이언트 모듈을 실행한다.

다음은 JEUS에서 제공하는 클라이언트 컨테이너의 command line 형식이다.

- 사용법

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

다음은 명령어 옵션에 대한 설명이다.

| 옵션 | 설명 |
|--------------------------------|---|
| -client <i>client_jar_path</i> | 실행할 애플리케이션 클라이언트의 패스를 지정한다. |
| [-main <i>main_class</i>] | 애플리케이션 클라이언트의 Main-Class를 지정한다. 클라이언트의 클래스 패스가 지정된 경로에 META-INF\MANIFEST.MF의 설정 정보에 Main-Class가 지정되어 있다면 이 옵션이 필요하지 않다. |
| [-cp <i>classpath</i>] | 필요한 경우 클라이언트 실행에 필요한 클래스 패스를 지정한다. |

- 예제

위의 예제를 실행하면 다음과 같다. 애플리케이션 클라이언트가 정상적으로 실행되기 위해서 Hello EJB가 deploy되어 있어야 한다.

```
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
Fix#0
EJB output : Hello EJB!
```

위의 내용 중 JEUS 로그를 출력하지 않으려면 다음을 추가로 설정한다.

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



로그 설정에 대한 자세한 설명은 JEUS Server 안내서의 "Logging"을 참고한다.

2. 고급 애플리케이션 클라이언트

본 장에서는 이전 장에서 설명한 애플리케이션 클라이언트의 설명보다 좀 더 고급 모듈 구현에 대해서 설명한다.

2.1. 개요

Jakarta EE 스펙에 제시된 클라이언트 컨테이너를 사용하지 않고도 간편하게 일반 Java 클래스를 실행하듯이 JEUS 클라이언트를 실행할 수 있다. 이를 위해서 클라이언트 컨테이너가 Dependency Injection을 하는 리소스의 종류와 이들의 JNDI 디폴트 바인딩 이름을 살펴본다. 또한 클라이언트 컨테이너 없이 동작하는 클라이언트와 애플리케이션 클라이언트가 사용할 수 있는 서비스(보안 설정, 트랜잭션)에 대해 자세히 설명한다.

클라이언트 컨테이너를 사용하지 않고 JEUS 클라이언트를 실행하는 경우에는 컨테이너의 Dependency Injection 서비스를 받을 수 없음에 유의해야 한다. 또한 보안 설정을 통해 JEUS가 제공하는 다양한 서비스를 실행할 수 있고, 트랜잭션 기능을 사용하면 클라이언트가 사용한 애플리케이션과 리소스들을 글로벌 트랜잭션으로 관리할 수 있다.

2.2. Dependency Injection

본 절에서 설명하는 Injection의 내용은 애플리케이션 클라이언트를 포함하여 웹 애플리케이션, EJB 애플리케이션 등에 모두 적용되는 내용이다.

Injection이 가능한 리소스는 EJB 객체와 JNDI로 매핑이 가능한 Environment Variable 등이 있으며 기본적으로 애플리케이션 컴포넌트의 JNDI 컨텍스트인 `java:comp/env` context에서 지정된 이름을 찾는다. 실제 리소스는 JNDI 글로벌 컨텍스트에 바인딩되어 있으므로 이 글로벌 바인딩 이름을 알아야 한다.

리소스들은 자신의 JNDI 글로벌 바인딩 이름을 갖고 있다. EJB 애플리케이션을 예로 들면 이 이름은 다음 중에 한 가지 방법으로 설정되어야 한다.

- JEUS에서 인식하는 `jeus-ejb-dd.xml`의 `<export-name>`을 사용
- 표준에 있는 `ejb-jar.xml`의 `<mapped-name>`을 사용
- EJB 애플리케이션에 지정된 Annotation의 `mappedName`을 사용
- "JEUS EJB 안내서"에 제시되어 있는 JEUS의 기본 JNDI 이름으로 EJB가 바인딩

클라이언트 컨테이너가 없는 환경 등에서 JNDI를 직접 사용하여 EJB를 얻을 경우에는 이런 기본 JNDI 이름이 정해지는 규칙을 알아야 한다. 이렇게 어떤 이름으로 바인딩될지 개발자가 알기 힘들기 때문에 실제 개발에서는 위의 방법 중 어느 하나의 방법으로 JNDI 바인딩 이름을 명시하는 것이 일반적이다.

리소스를 Injection하는 쪽에서는 JEUS 자체의 DD인 `jeus-ejb-dd.xml`, `jeus-web-dd.xml`, `jeus-client-dd.xml` 등에서 Injection에 사용할 JNDI 글로벌 바인딩 이름을 지정하거나 `ejb-jar.xml`, `web.xml`, `application-client.xml` 등의 표준 DD의 `mapped-name` 또는 Annotation의 `mappedName`에 지정된 값을 사용하여 매핑한다. 이 모든 값이 지정되어 있지 않는 경우에는 JEUS의 기본 규칙에 따른 이름으로 Injection을 시도한다.

실제 개발에서는 Annotation의 `mappedName`으로 JNDI 글로벌 바인딩 이름을 지정하기보다는 XML을 사용하여 지정하는 것이 좋다. 특히 여러 곳에서 운영할 애플리케이션이라면 그 환경에 맞는 글로벌 이름을 사용해야 하므로 XML을 사용해야 한다. Injection은 Annotation을 사용한 변수와 setter 메소드에 대해서 이루어지지만 Annotation을 사용하지 않아도 XML Descriptor에서 지정한 변수와 setter 메소드에 대해 injection이 가능하다.



1. Injection의 자세한 설명은 [Jakarta EE 9 Platform](#)을 참고하고, Injection이 가능한 리소스의 자세한 설명은 해당 안내서의 "5. Resources, Naming and Injection"을 참고한다.
2. EJB 애플리케이션에 대한 EJBContext injection 등은 "JEUS EJB 안내서"를 참고한다.

다음은 Annotation의 mappedName을 사용한 EJB 애플리케이션을 클라이언트에서 Injection하는 예제이다. StatelessEJB1 애플리케이션이 'MyEJB1'이란 이름을 JNDI 글로벌 바인딩 이름으로 사용하므로 클라이언트에서는 @EJB Annotation의 mappedName에 같은 이름을 지정한다.

또한 클라이언트에서 Injection 대신 JNDI Lookup을 사용한다면 JNDI 글로벌 바인딩 이름으로 바로 Lookup을 하거나 클라이언트 컨테이너 등에서 동작하는 클라이언트라면 application-client.xml 등을 사용하여 애플리케이션 컨텍스트에 등록된 이름인 java:comp/env/ejb/sless1를 사용할 수 있다.

EJB 참조 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

EJB 참조의 경우 Injection을 위해 다음과 같은 바인딩 이름을 사용한다.

- 변수 타입이 비즈니스 인터페이스인 경우

mappedName이 있는 경우에 Lookup할 때 사용할 글로벌 이름을 다음 형식으로 설정한다.

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

위 예제의 경우 'MyEJB1' 또는 'MyEJB1#ejb1.RemoteSession'을 사용한다. EAR이나 EJB JAR 등으로 deploy된 경우 ejb-jar.xml에 ejb-link가 주어졌거나 Annotation에 beanName이 있을 때에는 동일 애플리케이션 내의 EJB를 찾아 그 EJB의 mappedName을 글로벌 이름으로 사용하여 Lookup한다. 만약 이런 정보가 없는 경우에는 비즈니스 인터페이스의 이름으로 같은 애플리케이션 내에서 EJB를 찾아 그 EJB의 mappedName을 글로벌 이름으로 사용한다.

마지막으로 비즈니스 인터페이스 이름으로 JNDI에서 Lookup한다. 위의 예제의 경우 'java:global/<module-

name>/MyEJB1' 또는 'java:global/<module-name>/MyEJB1#ejb1.RemoteSession' 이름으로 Lookup한다. 이 방식은 EJB deploy의 경우 mappedName이 없을 때 기본 바인딩 이름을 사용하는 것이다.

- 변수 타입이 EJBHome/EJBObject 인터페이스의 하위 인터페이스인 경우

mappedName이 있는 경우에 **mappedName**을 글로벌 이름으로 사용한다. EAR이나 EJB JAR 등으로 deploy된 경우 ejb-jar.xml에 EJB-link가 주어졌거나 Annotation에 beanName이 있을 때에는 동일 애플리케이션 내의 EJB를 찾아 그 EJB의 mappedName을 사용하여 Lookup한다. 만약 이런 정보가 없는 경우에는 변수 타입의 인터페이스 이름으로 같은 애플리케이션 내에서 EJB를 찾아 그 EJB의 mappedName을 글로벌 이름으로 사용한다.

마지막으로 비즈니스 인터페이스 이름으로 JNDI에서 Lookup한다.

2.2.2. 리소스 Injection

리소스인 경우에는 @Resource Annotation을 사용할 수 있다.

- mappedName이 지정된 경우에는 이 이름을 리소스의 JNDI 글로벌 바인딩 이름으로 Lookup한다.
- mappedName이 지정되지 않은 경우에는 @Resource의 name 속성 값을 JNDI 글로벌 바인딩 이름으로 사용한다.

name 속성 값이 지정되지 않은 경우에는 스펙에 따라 다음의 형식이 사용된다.

애플리케이션 클래스의 이름 + / + 변수 또는 setter 메소드의 프로퍼티 이름

다음의 예제에서는 'jdbc/DB2' 이름으로 JNDI Lookup을 한다. 만약 name 속성이 지정되지 않았다면 'test.Client/myDataSource3'로 Lookup한다.

리소스 Injection

```
package test;

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



name이 있는 경우 또는 name 속성이 지정되지 않은 경우의 기본값은 스펙으로 정해져 있지만 이 값은 애플리케이션 컨텍스트에 매핑되는 이름이고, 실제 JNDI 글로벌 바인딩 이름은 벤더마다 다른 규칙을 갖고 있다. 따라서 호환성을 위해서는 mappedName 등을 사용하는 것이 좋다.

2.2.3. 그 외의 Injection

이외에도 @WebServiceRef, @PersistenceUnit, @PersistenceContext 등의 Annotation을 사용해서 각각 웹 서비스 객체, EntityManager 객체, EntityManagerFactory 객체 등을 Injection으로 얻을 수 있다.



그 외 자세한 설명은 [Jakarta EE Platform Specification](#)을 참고한다.

2.3. Dependency Injection을 사용할 수 없는 클라이언트

애플리케이션 컨테이너를 사용하지 않고도 JEUS의 JNDI 등을 통해 JEUS의 리소스와 애플리케이션을 사용할 수 있다. 이 경우에는 JEUS_HOME/lib/client의 jclient.jar 파일을 클래스 패스로 설정하고 작성한 클라이언트를 수행한다.

하지만 이 경우 Dependency Injection은 사용할 수 없으므로 다음과 같이 소스를 변경해서 사용해야 한다. 예제에서 사용한 EJB 애플리케이션의 바인딩 이름은 JEUS의 기본 바인딩 이름 규칙에 따른 이름을 사용한다. 이 클라이언트는 클라이언트 컨테이너 위에서도 동작한다. 즉, 클라이언트 컨테이너를 사용하지 않는 모든 클라이언트는 클라이언트 컨테이너 위에서도 동작한다.

Dependency Injection을 사용할 수 없는 클라이언트 : <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();
        }
    }
}
```

클라이언트 패키징은 XML 파일을 처리하는 클라이언트 컨테이너에서 실행하는 것이 아니므로 표준, JEUS XML 파일 없이 JAR 파일로 작성한다. deploy는 마찬가지로 자신이 원하는 위치에 JAR 파일을 복사한다. 실행할 때는 일반 Java 클래스를 실행하는 것과 같이 위의 클라이언트 클래스를 실행한다.

실행한 결과는 다음과 같다.

```
$ 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. 보안 설정

클라이언트에서는 JEUS가 제공하는 Jakarta EE의 서비스를 사용할 때 자신이 해당 서비스를 사용할 권한이 있는지를 확인할 수 있도록 클라이언트의 사용자명, 패스워드를 클라이언트 런타임에 지정해야 한다. 이런 서비스에는 EJB 애플리케이션, JMS 리소스 등이 있다. 이를 지정하는 방법은 다음의 3가지가 있다.

- **jeus-client-dd.xml을 사용하는 방법**

jeus-client-dd.xml의 **<security-info>**를 지정하면 클라이언트 컨테이너에서 애플리케이션을 수행하기 전에 주어진 사용자명, 패스워드로 로그인한다. 이후 애플리케이션에서는 JEUS 서비스를 사용할 때 이 사용자명으로 인증을 시도한다. jeus-client-dd.xml 설정에 대한 자세한 설명은 "JEUS XML Reference"을 참고한다.

보안 설정 : <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>
```

- **JNDI 컨텍스트를 사용하는 방법**

애플리케이션에서 JNDI 컨텍스트를 생성할 때 JNDI의 프로퍼티를 사용해서 원하는 사용자명, 패스워드로 로그인할 수 있다. 이후 로그인한 사용자명으로 인증이 이루어진다. 이 방법은 클라이언트 컨테이너를 사용하지 않을 경우에도 가능하다. JNDI 설정은 JEUS Server 안내서의 "JNDI Naming Server"를 참고한다.

보안 설정 : <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);
```

- **JEUS Security API를 직접 사용하는 방법**

JEUS의 Security API를 사용하여 로그인할 수 있다. Security API에 대한 자세한 내용은 "JEUS Security 안내서"를 참고한다.

2.5. 트랜잭션

클라이언트에서 사용하는 EJB 애플리케이션, JDBC DataSource, JMS Connection Factory와 Destination 등을 하나의 글로벌 트랜잭션 또는 XA 트랜잭션으로 사용하기 위해서는 UserTransaction을 사용한다.

클라이언트에서 사용하는 트랜잭션 매니저는 리소스를 직접 관리하는지의 여부에 따라 서버 트랜잭션 매니저와 클라이언트 트랜잭션 매니저로 나눌 수 있다.



1. UserTransaction의 자세한 설명은 [Jakarta Transaction 2.0 Specification](#)을 참고한다.
2. 트랜잭션 매니저의 자세한 설명은 JEUS Server 안내서의 "트랜잭션 매니저"를 참고한다.

3. 애플릿 클라이언트

본 장에서는 JEUS에서 애플릿 프로그램의 작성 및 설정하고 실행하는 방법에 대해 설명한다.

3.1. 개요

애플릿은 웹 브라우저에서 실행되는 Java 애플리케이션이다.

웹 브라우저에서 실행되는 애플릿에서 JEUS의 Jakarta EE 서비스를 사용하기 위해서는 애플릿 컨테이너를 사용한다. 기본적으로 애플릿은 애플리케이션 클라이언트이지만 JEUS에서는 아직 경량 클라이언트 컨테이너를 제공하지 않기 때문에 JEUS의 라이브러리들에 바로 접근할 수 없는 애플릿은 컨테이너 없이 바로 JEUS의 Jakarta EE 서비스를 사용하는 방법으로 구현한다.

애플릿을 사용하면 브라우저에서 Java 애플리케이션을 수행할 수 있다. 더불어 JEUS의 기능을 사용하는 클라이언트로 동작할 수 있다.

3.2. 프로그램 작성

애플릿이 구동되는 데 필요한 파일들은 웹 애플리케이션 내에 존재한다. HTML의 JAVA_CODEBASE가 "."으로 되어 있기 때문에 이 JAR 파일들은 HTML 문서가 deploy되는 웹 애플리케이션에서 HTML 문서와 같은 디렉터리에 존재해야 한다.

본 절에서는 사용자가 참고할 수 있는 샘플 예제를 설명한다.

3.2.1. 애플릿 예제

애플릿 애플리케이션은 Applet 또는 JApplet 클래스를 상속받고, start() 메소드를 구현해야 한다.

다음 예제는 클라이언트 컨테이너를 통해서가 아닌 독립적으로 수행되는 클라이언트의 예이다. 따라서, Dependency Injection을 사용하지 않고 JNDI API를 사용하여 해당 EJB를 직접 Lookup하게 되어 있다.

앞의 클라이언트 컨테이너와 동일한 EJB를 사용하고 이 경우 EJB는 helloejb.Hello라는 인터페이스 이름을 바인딩 이름으로 사용한다.

애플릿 애플리케이션 : <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();
        }
    }
}
```



EJB가 어떤 이름으로 JNDI에 바인딩되는가에 대한 자세한 내용은 "JEUS EJB 안내서"를 참고한다.

3.2.2. HTML 예제

HTML 문서에는 어떤 애플릿을 호출하고 이 애플릿의 클래스들의 위치를 지정한다.

예제에서는 위의 애플리케이션 클래스와 helloejb.Hello EJB 인터페이스가 hello-client.jar에 포함되어 있다고 가정한다. jclient.jar는 JEUS_HOME\lib\client에 존재하는 클라이언트용 JEUS 라이브러리이다.

HTML 예제 : <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>
```



HTML 문서를 브라우저에 관계없이 사용하고 또한 브라우저가 실행되는 곳에 JDK가 설치되어 있지 않다면 이를 설치하도록 유도할 수 있다. HTML 문서로 변환하기 위해서는 JDK의 `htmlconverter`를 사용한다. 자세한 내용은 [Java Plug-in HTML Converter 문서](#)를 참고한다.

3.3. Deploy

애플릿은 기본적으로 HTML 문서에서 접근하므로 이 HTML 문서를 웹으로 전송하기 위해서는 웹 애플리케이션이 필요하다. 애플릿을 실행하기 전에 웹 애플리케이션과 EJB의 `deploy`가 완료되어야 한다.

• 웹 애플리케이션 Deploy

웹 애플리케이션을 생성하고 HTML 문서 및 'ARCHIVE'에 지정된 JAR 파일들을 추가한다. 웹 애플리케이션의 생성과 `deploy`에 대한 자세한 내용은 "JEUS Application & Deployment 안내서"를 참고한다.

• EJB Deploy

웹 애플리케이션 `deploy`가 완료되면 EJB 애플리케이션을 `deploy`한다.

3.4. 실행

웹 브라우저를 실행하고 HTML 문서 URL을 입력하여 애플릿을 실행한다. 애플릿의 경우 Java Security 모델에 따라 `java.policy`에 설정한 대로 Access control을 한다. 따라서 `java.policy` 파일에 애플릿에서 사용하는 클래스에 대해 `permission`을 제공해야 애플릿이 문제없이 수행된다.



`java.policy` 설정은 [Security Developer's Guide](#)를 참고한다.

3.4.1. 웹 브라우저에서 실행

웹 브라우저는 HTML 페이지 내의 `<applet>` 태그로 접속한다. 예제의 애플릿은 Swing을 사용하므로 다음의 주소로 웹 브라우저를 통해서 접근할 수 있다.

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

3.4.2. 애플릿 뷰어에서 실행

테스트 과정에서는 브라우저를 통하지 않고 JDK에 포함된 애플릿 뷰어를 사용하여 애플릿을 수행할 수 있다. 이 경우 Exception 등을 바로 확인할 수 있어서 브라우저에서 테스트하는 것보다 쉽게 개발할 수 있다.

```
appletviewer index.html
```