

Web Engine 안내서

JEUS 9.1

TMAXSOFT

저작권 공지

Copyright 2026. TmaxSoft Co., Ltd. All Rights Reserved.

회사 정보

(주)티맥스소프트

주소 : 경기도 성남시 분당구 정자일로 45, 티맥스소프트타워

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 9.1.1	3.4.1	2026-06-08	Fix 릴리스
JEUS 9.1	3.3.1	2025-09-30	GA 버전
JEUS 9 Fix#1	3.2.2	2025-07-10	JDK 21 지원 내용 추가
JEUS 9 Fix#1	3.2.1	2025-06-30	RC 버전
JEUS 9	3.1.2	2025-01-03	-

제품 버전	안내서 버전	발행일	비고
JEUS 9	3.1.1	2024-09-27	-

목차

1. 웹 엔진	1
1.1. 개요	1
1.2. 구성 요소	1
1.3. 주요 기능	2
1.4. 관리 도구	4
1.4.1. 도구 종류	5
1.4.2. 제어와 모니터링	5
1.5. 디렉터리 구조	5
1.6. 환경설정	7
1.6.1. XML 설정 파일	8
1.6.2. 모니터링 설정	9
1.6.3. 기본 오류 페이지 설정	10
1.6.4. 오류 발생의 경우 Stack Trace 첨부 설정	11
1.6.5. 인코딩 설정	11
1.6.6. JSP 엔진 설정	18
1.6.7. 응답 헤더 설정	18
1.6.8. 쿠키 정책 설정	19
1.6.9. 세션 설정	20
1.6.10. 로그 설정	20
1.6.11. Async Servlet 타임아웃 처리 설정	26
1.6.12. 웹 엔진 레벨의 프로퍼티 설정	26
1.6.13. 웹 공격 대응 설정	26
1.6.14. 특정 URL 패턴의 HTTP 요청 방지	27
1.7. 웹 엔진 튜닝	28
2. 웹 커넥션 관리	29
2.1. 개요	29
2.2. 구성 요소	29
2.2.1. 리스너	30
2.2.2. 커넥터	31
2.2.3. Worker Thread Pool	32
2.3. 웹 커넥션 설정	32
2.3.1. 리스너 공통 설정	33
2.3.2. AJP 리스너 설정	36
2.3.3. HTTP 리스너 설정	37
2.3.4. TCP 리스너 설정	37
2.3.5. WebtoB 커넥터 설정	38
2.3.6. Tmax 커넥터 설정	39
2.4. 부하 분산을 위한 웹 서버 설정	41
2.4.1. 부하 분산 구조	41

2.4.2. Apache 웹 서버	42
2.4.3. IIS 웹 서버 및 Iplanet 웹 서버 설정	45
2.4.4. WebtoB 부하 분산 설정	46
2.5. TCP 리스너 사용	48
2.5.1. 맞춤 통신 프로토콜 정의	49
2.5.2. Dispatcher Config Class 구현	49
2.5.3. TCP 서블릿 구현	51
2.5.4. 맞춤 프로토콜 코드를 위한 TCP 리스너 설정	54
2.5.5. TCP 클라이언트 구현	54
2.5.6. TCP 클라이언트 컴파일과 실행	56
2.6. HTTP/2 사용	57
2.6.1. HTTP/2 사용 설정	57
2.6.2. Server Push	59
2.7. 리스너 튜닝	59
3. 웹 컨텍스트	61
3.1. 개요	61
3.2. 기본 구조	61
3.2.1. 웹 컨텍스트 콘텐츠	61
3.2.2. 웹 컨텍스트 내부 구조(WAR 파일 구조)	61
3.3. 웹 컨텍스트 Deploy	63
3.3.1. jeus-web-dd.xml 설정	63
3.3.2. 웹 컨텍스트 Redeploy(Graceful Redeploy)	67
3.3.3. Shared Library Reference 추가	67
3.3.4. 호환성을 위한 웹 컨텍스트 레벨의 옵션 설정	70
3.3.5. 부가 기능 사용을 위한 설정	71
3.3.6. 웹 보안 설정	72
3.3.7. 보안 Role 매핑	73
3.3.8. Symbolic Reference 매핑	74
3.4. 웹 컨텍스트 모니터링	76
3.5. 웹 컨텍스트 제어	76
3.5.1. 웹 컨텍스트 리로드	76
3.5.2. 웹 컨텍스트의 비동기 요청에 대한 Thread Pool 모니터링	77
3.5.3. 웹 컨텍스트 중지	78
3.5.4. 웹 컨텍스트 재개	78
3.6. 웹 컨텍스트 튜닝	79
3.7. 비동기식 처리 프로그래밍 가이드	79
3.7.1. Servlet 표준에 따른 주의사항	79
3.7.2. 기타 주의사항	83
3.8. 변경된 예외 처리 정책 안내	83
4. 스레드 풀	85
4.1. 개요	85

4.2. 기본 구조	85
4.3. 설정	86
4.4. 자동 Thread Pool 관리 설정(Thread 상태 통보)	88
4.5. 규칙	89
5. JSP 엔진	91
5.1. 개요	91
5.2. Apache Tomcat Jasper	92
5.3. JSP 엔진 기능	93
5.3.1. JSP Graceful Reloading	93
5.3.2. JSP 프리컴파일	94
5.3.3. 메모리에서의 JSP 컴파일 및 실행 기능	94
5.4. JSP 엔진 설정	94
5.4.1. 웹 엔진 레벨에서 설정	94
5.4.2. jeus-web-dd.xml 설정	96
5.4.3. JSP 하위 호환성을 위한 웹 컨텍스트 레벨의 옵션 설정	96
5.4.4. jarscan.properties 파일 설정	97
6. 가상 호스트	98
6.1. 개요	98
6.2. 웹 엔진과 가상 호스트	98
6.2.1. ServletContext 객체와 가상 호스트	99
6.3. 가상 호스트를 통한 웹 컨텍스트 요청	99
6.3.1. URL 매칭 예제	100
6.3.2. URL 매칭 순서	101
6.4. 가상 호스트 설정	101
6.4.1. 추가	102
6.4.2. 수정	102
6.4.3. 삭제	103
7. WebSocket 컨테이너	104
7.1. 개요	104
7.2. 제약 사항	104
7.3. WebSocket 컨테이너 기능	104
7.3.1. WebSocket UserProperties Failover	105
7.3.2. 기타	106
7.4. WebSocket 컨테이너 설정	107
7.5. Spring WebSocket 사용	109
8. JEUS WebCache	111
8.1. 개요	111
8.2. JSP Caching	111
8.2.1. 기본 내용	111
8.2.2. <cache> 태그	112
8.2.3. <jeus:cache> 사용 예	115

8.2.4. flush 기능	116
8.2.5. refresh 기능	116
8.3. HTTP Response Caching	117
8.3.1. 필터 설정	117
9. 클래스 동적 반영	120
9.1. 개요	120
9.2. 기본 설정 및 동작	120
9.2.1. 서버 설정	120
9.2.2. 애플리케이션 설정	121
9.2.3. JEUS HotSwap으로 지원 가능한 애플리케이션 및 변환	122
9.2.4. JEUS HotSwap 제약 사항	125
10. 밸브와 필터	127
10.1. 개요	127
10.2. 밸브의 개념	127
10.3. JEUS 제공 valve 설정	128
10.3.1. 메소드 제한 옵션 설정	128
10.3.2. remote host/adderss valve 설정	129
10.3.3. URL rewrite 옵션 설정	130
10.4. 사용자 정의 밸브	131
10.5. JEUS 제공 filter 설정	132
10.5.1. samesite 제한 옵션 설정	132
10.5.2. Forwarded Header 옵션 설정	133
11. 따라하기	135

1. 웹 엔진

본 장에서는 JEUS 웹 엔진의 구성 요소, 주요 기능, 기본적인 설정 방법에 대해 설명한다.

1.1. 개요

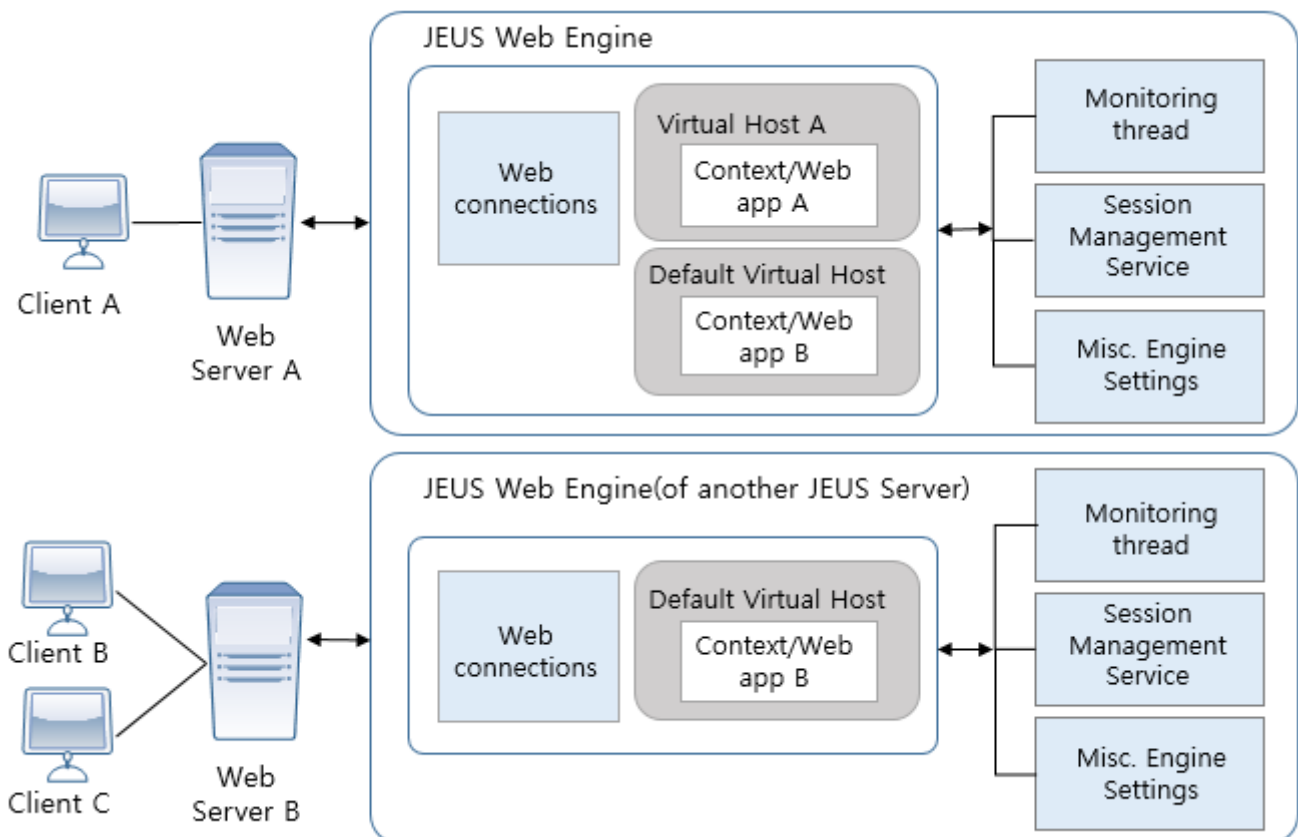
Jakarta EE 웹 애플리케이션(이하 웹 애플리케이션)은 사용자에게 동적으로 웹 콘텐츠를 제공하기 위한 서블릿(Servlet), JSP와 HTML 파일 같은 정적 리소스들로 구성되어 있다. JEUS는 이러한 웹 애플리케이션을 효율적으로 관리하는 웹 엔진을 제공한다.

본 장에서는 JEUS 웹 엔진에 대하여 소개하고, 운영 환경에서 제어하고 모니터링하는 방법을 설명한다.

1.2. 구성 요소

본 절에서는 웹 엔진의 구성 요소와 웹 엔진과 연관된 외부 요소들에 대해 설명한다.

웹 엔진은 Jakarta EE, 서블릿, JSP, EL 표준에 따르는 웹 애플리케이션을 관리하고 이를 서비스하는 개체이다. 서블릿과 JSP와 같은 동적 리소스뿐만 아니라 HTML과 같은 정적 리소스도 서비스할 수 있다.



웹 엔진의 구성 요소



웹 엔진은 웹 컨테이너(Web Container) 또는 서블릿 엔진(Servlet Engine)이라고 부르기도 한다.

주요 구성 요소들은 다음과 같다.

- **Web Connections**

Web Connections(이하 웹 커넥션)은 클라이언트, 웹 서버 또는 기타 다른 서버들과 웹 엔진이 연결되기 위한 요소이다.

여러 개의 웹 서버와 웹 엔진들이 서로 연결되어 성능과 안정성을 향상시킨 커다란 클러스터링으로 구성할 수 있다. 이에 대한 내용은 [웹 커넥션 관리](#)를 참고한다.

- **Virtual Host**

클라이언트가 서로 다른 호스트 이름을 지정해서 호출했을 때 각 호스트 이름에 매핑된 웹 애플리케이션을 호출할 수 있도록 제공하는 가상의 그룹이다. 웹 애플리케이션을 특정 가상 호스트를 지정해서 deploy할 수 있다. 자세한 내용은 [가상 호스트](#)를 참고한다.

- **Web Context(웹 애플리케이션)**

서블릿, JSP 표준을 따르는 웹 애플리케이션을 웹 엔진에 deploy하면 이에 대한 Web Context(이하 웹 컨텍스트)를 생성한다. 웹 애플리케이션은 일반적으로 WAR(Web ARchive) 형태로 패키징하며, 웹 엔진 내의 가상 호스트에 deploy된다. 자세한 내용은 [웹 컨텍스트](#)를 참고한다.

- **Monitoring Thread**

웹 엔진의 여러 구성 요소들을 감시하는 역할을 한다. 자세한 내용은 [주요 기능](#)의 모니터링을 참고한다.

- **Session Management Service**

분산된 환경에서 클라이언트의 세션을 관리하고 이를 여러 웹 엔진에서 사용 가능한 환경을 제공한다. 자세한 내용은 JEUS 세션 관리 안내서의 "세션 트래킹"을 참고한다.

- **기타 웹 엔진 설정**

웹 엔진에 공통적으로 적용되는 설정들이다. 기본 오류 페이지, 인코딩 설정, JSP 엔진 설정, 응답 헤더 설정 등이 이에 해당된다.

웹 엔진과 연관된 외부 요소들은 다음과 같다.

- **클라이언트**

JEUS 웹 엔진으로 서비스를 요청한다. HTTP 기반의 웹 브라우저가 일반적이다.

- **웹 서버**

클라이언트의 HTTP 요청을 받아서 필요한 경우에 JEUS 웹 엔진에 전달한다. JEUS 웹 엔진은 단독으로 HTTP 클라이언트의 요청을 처리하기 보다는 웹 서버(Web Server)와의 조합으로 사용되는 경우가 일반적이다. 자세한 내용은 [웹 커넥션 관리](#)를 참고한다.

1.3. 주요 기능

웹 엔진의 주요 기능은 다음과 같다.

• 웹 애플리케이션 관리

웹 엔진은 Jakarta EE 표준을 준수하는 웹 애플리케이션의 Deploy, Undeploy 및 일시 서비스 정지 기능을 제공한다. 그 외에도 서비스의 중단 없이 Redeploy하는 기능을 지원한다. 이에 관한 자세한 내용은 [웹 컨텍스트](#)를 참고한다.

• 모니터링

모니터링은 웹 엔진의 자원들의 상태를 감시하고 문제가 발생했을 때 대응하기 위한 기능이다. 크게 3가지 타입의 모니터링이 존재한다. 설정에 대한 자세한 내용은 [모니터링 설정](#)을 참고한다.

구분	설명
Thread Pool 모니터링	Web Connection, VirtualHost, Context 별로 가지고 있는 Thread Pool을 모니터링한다. 자세한 내용은 웹 커넥션 관리 를 참고한다.
클래스 로더 모니터링	웹 애플리케이션들의 서블릿 클래스들의 변경을 모니터링한다. 자세한 내용은 웹 컨텍스트 를 참고한다.
세션 서비스 모니터링	클라이언트 세션의 기한 만료 여부를 체크하고 만료된 세션을 제거한다. 세션 서비스 모니터링에 대한 자세한 내용은 JEUS 세션 관리 안내서의 "세션 트래킹"을 참고한다.

• 기본 오류 페이지 설정

웹 엔진은 해당 컨텍스트가 준비되지 않은 상태에서 보여줄 오류 페이지를 설정할 수 있다. 설정에 대한 자세한 내용은 [기본 오류 페이지 설정](#)을 참고한다.

• 오류 발생의 경우 Stack Trace 첨부 설정

웹 엔진은 오류가 발생했을 경우 오류가 발생한 지점의 Stack Trace를 기본 오류 페이지에 첨부할지 여부를 설정할 수 있다. 설정에 대한 자세한 내용은 [오류 발생의 경우 Stack Trace 첨부 설정](#)을 참고한다.

• 인코딩 설정

웹 엔진은 등록된 모든 컨텍스트에 의해 사용될 수 있는 인코딩 설정을 가지고 있다. 설정에 대한 자세한 내용은 [인코딩 설정](#)을 참고한다.

• JSP 엔진 설정

JSP 엔진은 각 웹 애플리케이션 별로 생성된다. JSP 엔진은 JSP 자원을 클라이언트가 요청했을 때 JSP 페이지들을 Java 파일로 변환 후 이 Java 파일을 컴파일하여 서블릿 바이트 코드로 생성하는 작업을 한다. 실행 결과로 하나의 JSP에 대해 Java 파일과 클래스 파일이 생성된다. 단, Java 파일은 jsp 작성시 문제점을 확인하기 위한 디버깅 용도로 사용하는 것이다. 설정에 대한 자세한 내용은 [JSP 엔진 설정](#)을 참고한다.

• 응답 헤더 설정

웹 엔진은 사용자 임의의 HTTP Response Header의 이름과 값의 짝을 설정할 수 있다. 사용자 임의의 응답 헤더를 설정할 경우 해당 웹 엔진에서 나가는 모든 응답에는 정의된 헤더가 항상 포함된다. 설정에 대한 자세한 내용은 [응답 헤더 설정](#)을 참고한다.

• 세션 관리

웹 엔진의 세션 관리에 대한 여러 가지 사항을 설정한다. 세션 클러스터링의 참여 여부, 세션 객체의 공유 여부,

세션 쿠키(Session Cookie) 설정, 타임아웃 등 웹 엔진의 세션과 관련된 모든 설정을 할 수 있다. 설정에 대한 자세한 내용은 [세션 설정](#)과 JEUS 세션 관리 안내서의 "세션 트래킹"을 참고한다.

• Event Logging

웹 엔진에서 남기는 로그는 서버 로그와 액세스 로그이다. 그리고 웹 애플리케이션에서 ServletContext API의 log 메소드로 남기는 유저 로그가 있다.

구분	설명
서버 로그	웹 엔진의 동작에 관련된 로그이다.
액세스 로그	웹 엔진에 요청 및 처리 결과에 대한 로그이다.
유저 로그	jakarta.servlet.ServletContext.log(String msg) 또는 jakarta.servlet.ServletContext.log(String msg, Throwable t) API를 사용하여 웹 애플리케이션 내에서 생성되는 메시지를 기록하는 로그이다.

JEUS 로그의 기본 위치는 [디렉터리 구조](#)를 참고하고, 로그 설정에 대한 자세한 내용은 [로그 설정](#)을 참고한다.

• Graceful Shutdown

관리자가 JEUS 서버를 Shutdown할 때 이미 진행 중이던 서비스 완료를 보장하는 기능이다. 실제로 관리자가 Shutdown 명령을 내리면 다음과 같은 순서로 동작한다.

1. 더 이상 새로운 서비스 요청을 받지 않는다.
2. 그 시점에 이미 진행 중이던 요청들의 완료를 보장한 뒤 다른 Shutdown 작업을 진행한다.

진행 중이던 서비스가 너무 오래 걸릴 경우 무한정 기다릴 수 없으므로 서버에 Shutdown 명령을 내릴 때 타임아웃 옵션을 설정할 수 있다. Graceful Shutdown에 대한 자세한 내용은 JEUS Server 안내서의 "Managed Server 종료"를 참고한다.



JEUS 6까지 제공하던 WEBMain.xml의 shutdown-timeout 설정은 더 이상 제공하지 않는다. 대신 콘솔 툴(jeusadmin)에서 shutdown timeout 옵션을 설정하면 동일한 기능을 수행한다.

• 웹 공격 대응

DoS(Denial of Service) 공격과 같은 웹 공격으로 인한 JEUS 서버의 부담을 덜기 위한 설정을 제공한다.

웹 공격을 방지하는 설정으로 POST 요청의 크기 제한 설정, GET/POST 요청에 포함된 파라미터들의 개수 제한, 하나의 요청에 포함된 헤더의 개수 제한, 하나의 요청 내에 포함된 헤더 하나의 크기 제한, 그리고 GET 요청의 Query String의 크기 제한을 설정할 수 있다. 이들에 대한 자세한 설정은 [웹 공격 대응 설정](#)을 참고한다.

1.4. 관리 도구

본 절에서는 웹 엔진의 제어 및 모니터링 기능을 제공하는 도구에 대해 설명한다.

1.4.1. 도구 종류

JEUS 웹 엔진을 운영하기 위한 도구는 다음과 같다.

- 콘솔 툴(jeusadmin)

OS 콘솔에서 웹 엔진을 제어할 수 있는 운영 도구이다. 기본적인 제어 기능과 모니터링 기능을 제공한다. 콘솔 툴의 사용법은 JEUS Server 안내서의 "JEUS 서버 제어 및 모니터링"과 JEUS Reference 안내서의 "웹 엔진 관련 명령어"를 참고한다.

1.4.2. 제어와 모니터링

운영 도구를 이용하여 웹 엔진의 제어 및 모니터링이 가능하다.

웹 엔진 제어

웹 엔진을 제어한다는 것은 웹 엔진의 시작과 종료를 제어한다는 의미와 같다. 이 2가지 작업은 콘솔 툴을 사용하여 가능하다.

- 콘솔 툴 사용

웹 엔진은 JEUS 서버에 포함되며, 서버와 별도로 제어할 수 있는 방법은 없다. 자세한 내용은 JEUS Server 안내서의 "JEUS 서버 제어 및 모니터링"의 서버 제어 설명을 참고한다.

웹 엔진 모니터링

콘솔 툴을 사용해서 웹 엔진을 모니터링할 수 있다.

```
show-web-statistics [-server <server-name>]
                    [-ctx,--context <context-name>]
                    [-r,--request | -s,--session | -t,--thread |
                    -m,--memory]
```

콘솔 툴에서는 웹 엔진에 대한 기초 정보를 조회할 수 있는 기능을 제공한다. 자세한 내용은 "JEUS Server 안내서"와 JEUS Reference 안내서의 "웹 엔진 관련 명령어"를 참고한다.

1.5. 디렉터리 구조

JEUS 디렉터리에서의 웹 엔진 구조는 다음과 같다.

```
{JEUS_HOME}
|--domains
|   |--domains1
|   |   |--config
|   |       |--[X]Domain.xml
|   |       |--Servlet
```

```

|--server1
|   |--[X]web.xml
|   |--[X]webcommon.xml
|--[X]web.xml
|--[X]webcommon.xml
|--servers
|   |--server1
|       |--logs
|           |--[T]jeusServer.log
|       |--servlet
|           |--vhost
|               |--[T]access.log
|               |--[T]access.log

```

* Legend

- [01]: binary or executable file
- [X] : XML document
- [J] : JAR file
- [T] : Text file
- [C] : Class file
- [V] : java source file
- [DD] : deployment descriptor

JEUS_HOME

JEUS 설치 홈 디렉터리이다.

domains/domain1/config/

JEUS_HOME의 domains 디렉터리 하위에는 각 도메인 이름을 기준으로 디렉터리가 존재한다. 이 중에서 웹 엔진의 설정과 관련되는 내용이 config 디렉터리 아래에 위치한다. *domain1*은 예시로 사용한 도메인 이름이다. **JEUS 7부터는 더 이상 WEBMain.xml을 사용하지 않고, 하나의 서버 JVM에는 하나의 웹 엔진만 존재하므로 엔진별 설정 디렉터리가 필요 없다.**

폴더 & 파일	설명
domain.xml	통합된 설정 파일이다. 해당 파일에 도메인에 속한 모든 서버의 설정이 있으며, 서버별로 웹 엔진을 설정할 수 있다. domain.xml에 관한 자세한 내용은 "JEUS Server 안내서"를 참고한다.
servlet/	<p>도메인에 속한 모든 서버들이 각 웹 애플리케이션을 deploy할 때 기본적으로 참조하는 XML 설정 파일이 위치한다. 이 디렉터리에 있는 web.xml과 webcommon.xml은 도메인에 속한 서버들이 부팅될 때 도메인 관리 서버로부터 받아오게 되므로 도메인의 서버들은 같은 파일을 유지하게 된다.</p> <ul style="list-style-type: none"> ◦ web.xml : Servlet 3.0부터는 web.xml이 기본적으로 없어도 된다. 그러나 JSP Parser인 Tomcat Jasper에서는 이 파일을 기반으로 웹 애플리케이션의 버전을 확인하므로 되도록 그대로 유지해야 한다. web.xml 파일을 가지고 있지 않은 웹 애플리케이션에서 이 파일을 참고한다. ◦ webcommon.xml : 모든 웹 애플리케이션들에 적용되는 공통 설정 파일이다. 이 파일의 형식은 web.xml 형식과 동일하다.

폴더 & 파일	설명
<div>servlet/server</div> <div>1/</div>	<p>servlet 설정 디렉터리 아래에 특정 서버 이름으로 디렉터리를 생성하고 그 아래에 webcommon.xml, web.xml을 위치시키면 특정 서버에 대해서만 적용할 수 있다. 이 디렉터리의 설정이 상위 디렉터리에 있는 설정에 우선한다.</p> <p>특정 서버 이름의 디렉터리에 있는 web.xml과 webcommon.xml은 JEUS가 자동으로 관리하지 않는다. 그러므로 이 설정 파일들은 서버 관리자가 수동으로 관리해야 한다.</p>

domains/domain1/servers/server1/logs

도메인에 속한 각 서버가 동작할 때 생성되는 파일들은 모두 servers 아래에 각 서버 이름으로 생성된 디렉터리 아래에 존재한다. 이에 관한 자세한 내용은 JEUS Server 안내서의 "Logging"을

로그 파일은 기본적으로 logs 디렉터리 아래에 있다. 웹 엔진에서 생성한 로그는 JeusServer.log에 저장된다. 단, 로그 로테이션 기능에 의해 파일 이름은 변경될 수 있지만 항상 JeusServer로 시작한다.

폴더 & 파일	설명
<div>servlet/</div>	<p>웹 애플리케이션 액세스 로그인 access.log를 저장한다. 만약 가상 호스트별로 액세스 로그를 설정했을 경우에는 기본적으로 이 디렉터리 아래에 가상 호스트 이름으로 디렉터리를 생성하고, 해당 가상 호스트에 deploy된 웹 애플리케이션들의 액세스 로그를 저장한다. 자세한 내용은 가상 호스트를 참고한다.</p> <p>또한, 이 디렉터리에는 웹 애플리케이션에서 ServletContext API의 log 메소드를 통해서 남기는 유저 로그인 user.log 또는 user_appname.log 파일을 저장한다. 유저 로그는 설정에 따라서 user.log 파일에 남을 수도 있고, 웹 애플리케이션 별도의 파일인 user_appname.log에 남을 수도 있다.</p>

1.6. 환경설정

본 절에서는 웹 엔진과 관련된 환경설정 파일과 웹 엔진 설정에 대해 설명한다.

웹 엔진에 대한 설정은 domain.xml을 사용해서 진행할 수 있다.

• 웹 엔진 설정

domain.xml에서 웹 엔진을 각각 설정할 수 있다.

```
<web-engine>
...
<jsp-engine>...</jsp-engine>
<web-connections>
  <http-listener>
    <name>SERVER-HTTP</name>
    <thread-pool>
      <min>10</min>
      <max>20</max>
      <max-idle-time>300000</max-idle-time>
      <max-queue>-1</max-queue>
    </thread-pool>
  </http-listener>
</web-connections>
</web-engine>
```

```

        <postdata-read-timeout>600000</postdata-read-timeout>
        <max-post-size>-1</max-post-size>
        <max-parameter-count>-1</max-parameter-count>
        <max-header-count>-1</max-header-count>
        <max-header-size>8192</max-header-size>
        <max-querystring-size>8192</max-querystring-size>
        <server-access-control>false</server-access-control>
        <allowed-server>...</allowed-server>
        <server-listener-ref>http-server</server-listener-ref>
    </http-listener>
</web-connections>
<monitoring>...</monitoring>
<access-log>...</access-log>
<ejb-engine>...</ejb-engine>
<jms-engine>...</jms-engine>
<system-logging>...</system-logging>
...
</web-engine>

```

1.6.1. XML 설정 파일

다음은 웹 엔진과 관련된 XML 설정 파일들에 대한 설명이다.

- **domain.xml** (jeus-domain.xsd)

위치	JEUS_HOME/domains/<domain-name>/config
목적	JEUS 서버별 웹 엔진 설정
상세 정보	"JEUS Server 안내서"

- **jeus-web-dd.xml** (jeus-web-dd.xsd)

위치	<packaged web application>/WEB-INF/
목적	JEUS 웹 모듈 Deployment Descriptor(이하 DD)
상세 정보	본 안내서

- **web.xml** (web-app_5_0.xsd)

위치	<packaged web application>/WEB-INF/
목적	Jakarta EE 표준 웹 모듈 DD
상세 정보	Servlet 5.0 표준

- **web.xml** (web-app_5.0.xsd)

위치	JEUS_HOME/domains/<domain-name>/config/servlet
목적	web.xml을 개별적으로 가지고 있지 않은 웹 모듈의 web.xml

상세 정보	Servlet 5.0 표준
-------	----------------

- **webcommon.xml** (web-app_5_0.xsd)

위치	JEUS_HOME/domains/<domain-name>/config/servlet
목적	웹 엔진의 모든 웹 모듈에 적용되는 공통 설정 파일
상세 정보	Servlet 5.0 표준

XML 스키마 파일들은 JEUS_HOME/lib/schemas/jeus/supportLocale/ko/ 디렉터리에 위치한다. 위 파일들의 내용은 반드시 JEUS에서 정의된 XML 헤더로 시작되어야 한다. 또한 Root Element는 JEUS XML 스키마의 namespace로 지정해야 한다.

각 파일에 사용되는 헤더는 다음과 같다. web.xml의 XML 헤더는 서블릿 표준에 포함되어 있다.

- domain.xml의 XML 헤더

```
<?xml version="1.0" encoding="UTF-8"?>
<domain xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
```

- jeus-web-dd.xml의 XML 헤더

```
<?xml version="1.0" encoding="UTF-8"?>
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
```

- web.xml의 XML 헤더

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="5.0" xmlns="https://jakarta.ee/xml/ns/jakartaee">
```



본 안내서에서 사용되는 모든 태그 순서는 XML 스키마의 설정 순서대로 작성되어 있다. 실제 사용할 때는 순서를 정확히 지키기가 쉽지 않으므로, JEUS에서는 태그 순서를 자동으로 정렬해주는 기능을 제공한다. 그러므로 XML 설정 파일을 작성할 때, 순서를 정확하게 지키지 않아도 무방하다. 태그 순서는 "JEUS XML Reference"를 참고한다.

위 파일들의 예제가 본 안내서에서 사용될 때에는 표준 헤더가 편의상 생략되어 사용되고 있다. 실제 XML 설정 파일에는 이 헤더들이 포함되어 있다는 것에 주의한다.

1.6.2. 모니터링 설정

웹 엔진은 내부 Thread와 클래스 변경사항, 세션의 변경사항을 모니터링할 수 있고, 이들의 모니터링 주기를 변경할 수 있다.

domain.xml 사용해서 모니터링 주기를 설정하는 방법은 다음과 같다.

```

<web-engine>
  ...
  <monitoring>
    <check-thread-pool>300000</check-thread-pool>
    <check-class-reload>300000</check-class-reload>
    <check-session>300000</check-session>
  </monitoring>
  ...
</web-engine>

```

다음은 모니터링 주요 항목에 대한 설명이다.

항목	설명
Check Thread Pool	요청 처리 스레드들의 상태를 점검하기 위한 시간 간격을 설정한다. (기본값: 300000(5분))
Check Class Reload	웹 컨텍스트를 자동으로 리로드하기 위해 클래스 변경 여부를 체크하는 주기를 설정한다. (기본값: 300000(5분)) 이 설정은 jeus-web-dd.xml에 <auto-reload><enable-reload> 기능을 설정해야 의미를 갖는다. <check-on-demand>를 true로 설정한 웹 컨텍스트의 경우에는 점검하지 않는다.
Check Session	세션의 타임아웃 상태 점검 주기를 설정한다. 세션의 타임아웃은 웹 엔진의 세션 설정이나 web.xml에 설정된다. (기본값: 300000(5분))

Thread Pool의 min/max 설정을 제외한 모든 모니터링 설정은 동적 반영이 되지 않으므로 서버를 재시작해야 설정 내용이 반영된다.

1.6.3. 기본 오류 페이지 설정

웹 요청은 웹 컨텍스트가 존재하는 경우에 오류가 발생하면 해당 웹 컨텍스트에서 오류를 처리한다. 그러나 웹 컨텍스트가 존재하지 않는 경우에는 웹 엔진이 내부적으로 에러 페이지를 보내주는 방법으로 오류를 처리해야 한다. 이 경우 오류 페이지가 사용자가 원하는 형식이 아닐 수 있으므로 이런 사용자의 요구사항을 처리할 수 있는 오류 페이지의 절대 경로를 지정할 수 있는 설정을 제공한다.

단, 이 경우에는 html 또는 htm 파일만 설정이 가능하다. 여기에 설정된 값은 HTTP 응답 내용으로만 사용될 뿐이며 forward 개념이 아니다.

domain.xml을 사용해서 기본 오류 페이지를 설정하는 방법은 다음과 같다.

'Default Error Page' 항목에 기본 오류 페이지와 관련된 절대 경로를 설정한다.

```

<web-engine>
  <default-error-page>/jeus/user/error.html</default-error-page>
</web-engine>

```

기본 오류 페이지 설정은 동적 반영이 되지 않으므로 서버를 재시작해야 설정 내용이 반영된다.

1.6.4. 오류 발생의 경우 Stack Trace 첨부 설정

웹 요청 처리 중 웹 엔진에 문제가 발생했을 때 오류의 상세 내역을 JEUS가 기본적으로 제공하는 오류 페이지에 표시할 것인지에 대한 설정이다. 이 메시지는 개발할 때는 유용하지만, 운영할 때는 제거하는 것이 바람직하다. 이에 대한 설정은 각 애플리케이션 `jeus-web-dd.xml`에서 설정 가능하다. `jeus-web-dd.xml`의 설정은 [jeus-web-dd.xml 설정](#)을 참고한다.

`domain.xml`을 사용하여 오류가 발생한 경우의 Stack Trace 첨부 여부를 설정하는 방법은 다음과 같다.

기본값은 `false`이고, `true`로 설정하면 기본 오류 페이지에 Stack Trace가 표시된다. 단, '**Default Error Page**' 항목이 설정된 경우에는 설정된 Error Page가 표시된다. 기본 오류 페이지 관련 설정에 대한 자세한 내용은 [기본 오류 페이지 설정](#)을 참고한다.

```
<web-engine>
  ...
  <attach-stacktrace-on-error>false</attach-stacktrace-on-error>
  ...
</web-engine>
```

Stack Trace 첨부 설정은 동적 반영이 되지 않으므로 서버를 재시작해야 설정 내용이 반영된다.

1.6.5. 인코딩 설정

웹 엔진은 엔진 내의 모든 컨텍스트에 의해 사용될 수 있는 3가지 인코딩 설정을 가지고 있다. 이는 모두 `jeus-web-dd.xml`을 통해서 각 애플리케이션별로 적용하거나, 가상 호스트별 또는 서버별로 설정 가능하다. 언급한 순서대로 xml 설정 우선순위가 적용된다. 관리자는 `domain.xml`에 인코딩 설정을 할 수 있다. **인코딩 설정은 웹 애플리케이션별로 다를 가능성이 매우 높으므로 `jeus-web-dd.xml`에 설정할 것을 권장한다.** (아래의 XML은 `jeus-web-dd.xml` 또는 `domain.xml` 설정이다.)

• Request Encoding

HTTP Request Header의 Query String, Cookie 및 바디에 사용되는 인코딩이다.

- HTTP Request Line의 Query String은 다음과 같은 우선순위로 인코딩이 적용되며, Servlet 또는 JSP에서 forward 또는 include 할 때 기술하는 Query String에도 적용된다.
 1. XML에 설정된 `<request-url-encoding>`의 "forced" 인코딩
 2. XML에 설정된 `<request-encoding>`의 "forced" 인코딩
 3. XML에 정의된 `<request-encoding>`의 "client-override" 인코딩
 4. XML에 정의된 `<request-url-encoding>`의 "default" 인코딩
 5. XML에 설정된 `<request-encoding>`안의 `<url-mapping>`의 "encoding" 인코딩
 6. XML에 설정된 `<request-encoding>`의 "default" 인코딩
 7. ISO-8859-1
- HTTP Request Header의 Cookie는 다음과 같은 우선순위로 인코딩이 적용된다.
 1. XML에 설정된 `<cookie-policy><write-value-on-header-policy>`의 "charset-encoding" 값

2. XML에 설정된 <request-encoding>의 "forced" 인코딩
3. XML에 정의된 <request-encoding>의 "client-override" 인코딩
4. XML에 설정된 <request-encoding>안의 <url-mapping>의 "encoding" 인코딩
5. XML에 설정된 <request-encoding>의 "default" 인코딩
6. ISO-8859-1

Cookie의 우선순위는 Response에서 Cookie을 내보낼 때도 함께 적용된다. Cookie 인코딩 설정을 일관성 있게 하려면 모든 웹 엔진에 1번 설정을 동일하게 적용하길 권장한다.

◦ HTTP 바디의 인코딩은 다음의 우선순위에 따라 적용된다.

1. XML에 정의된 <request-encoding>의 "forced" 인코딩
2. 애플리케이션(Servlet/JSP)에서의 설정(request.setCharacterEncoding())으로 설정한 인코딩)
3. XML에 정의된 <request-encoding>의 "client-override" 인코딩
4. HTTP 요청의 Content-Type의 charset에 의한 인코딩
5. XML에 설정된 <request-encoding>안의 <url-mapping>의 "encoding" 인코딩
6. XML에 정의된 <request-encoding>의 "default" 인코딩
7. web.xml에 설정된 <request-character-encoding>의 인코딩
8. ISO-8859-1

POST 요청이고 Content-Type이 application/x-www-form-urlencoded 인 경우, 웹 애플리케이션이 ServletRequest.getParameter()를 호출하면 웹 엔진이 바디를 읽어서 파라미터 처리를 하도록 되어 있다. 이 파라미터로 리턴되는 String 객체를 만들 때 HTTP 바디의 인코딩을 적용한다.

주의할 사항은 ServletRequest.getParameter()에서 URI Query String도 함께 처리하기 때문에 <request-url-encoding><forced> 설정된 인코딩과 <request-encoding> 설정된 인코딩이 다른 경우에는 서로 다른 인코딩이 적용된 String 객체가 리턴될 수 있다는 점이다. 따라서 클라이언트에서 파라미터를 전송할 때 Query String과 POST 바디의 인코딩은 일치시키는 것이 가장 바람직하다.

이외에도 바디 인코딩은 ServletRequest.getReader()를 호출할 때 적용된다.

이 부분은 JEUS 6와 7(Fix#1 이하)에 비교해서 변경 사항이 있으므로 기존 버전의 JEUS에 의존해서 HTTP Request Encoding을 처리한 애플리케이션은 위의 룰대로 수정하거나 호환성 옵션을 적용해야 한다.



JEUS 7 Fix#2부터 forced는 Servlet API보다 우선순위가 높다. JEUS 7 Fix#1 이하에서의 forced 설정을 그대로 유지해야 한다면 client-override 설정으로 치환해야 한다.

• Response Encoding

HTTP 응답 바디에 적용되는 인코딩이다.

Response Encoding은 ServletOutputStream 또는 PrintWriter의 print 메소드들을 Byte 배열로 변환할 때, HTTP 헤더의 "Content-Type:text/html;charset=XXX" 부분의 "XXX" 값을 설정할 때 등등 웹 컨테이너의 응답에 어떤 Character Encoding을 사용할지 결정한다. jeus-web-dd.xml에서도 설정이 가능하다.

◦ Response Encoding은 다음의 우선순위 목록에 따라 적용된다.

1. XML에 정의된 <response-encoding>의 "forced" 인코딩
2. API 호출에 의한 인코딩, JSP Response Character Encoding
3. XML에 정의된 <response-encoding>의 "default" 인코딩
4. web.xml에 설정된 <response-character-encoding>의 인코딩
5. ISO-8859-1

JSP Page Encoding은 JSP 파일 인코딩이다. 하지만 JSP Response Encoding과 구분해서 사용하는 경우가 거의 없을 것으로 예상되므로 <response-encoding> <forced>로 JSP 파일 인코딩을 치환한다. 단, JSP 파일에 BOM이 있는 경우에는 UTF-8으로 처리한다.



<response-encoding><forced>를 JSP Page Encoding에 적용하는 이유는 JSP 작성자가 page tag에 기술한 pageEncoding 값이 잘못되었고 이를 찾아내기 힘들 때 JEUS 설정만으로 바로잡을 수 있기 때문이다. 예를 들어 JSP 파일은 EUC-KR로 저장해놓고 pageEncoding 값은 UTF-8으로 잘못 적었더라도 <response-encoding><forced> 설정을 EUC-KR로 하면 글자가 깨지는 문제는 해결된다.

요약하면 JSP 파일을 읽을 때는 다음과 같은 우선순위가 성립한다.

BOM > <response-encoding><forced> > JSP Page Character Encoding > JSP Response Character Encoding > default



JSP Page Character Encoding 및 Response Character Encoding에 관한 자세한 사항은 JSP 표준을 참조한다.

domain.xml 사용

domain.xml을 사용하여 인코딩을 설정하는 방법은 다음과 같다.

<Encoding>에서 '**<request-encoding>**', '**<request-url-encoding>**', '**<response-encoding>**' 태그의 설정 값들을 입력한다. '**<request-url-encoding>**'을 제외하고 jeus-web-dd.xml에서도 설정이 가능하다. 각 항목별로 'Default', 'Forced' 또는 <request-encoding>의 경우 'Client Override' 중 하나를 선택하여 활성화한 후 설정할 인코딩명을 입력한다.

```
<web-engine>
...
<encoding>
  <request-encoding>
    <client-override>...</client-override>
  </request-encoding>
  <request-url-encoding>
    <default>...</default>
  </request-url-encoding>
  <response-encoding>
```

```

    <default>...</default>
  </response-encoding>
</encoding>
...
</web-engine>

```

설정값	설명
Default	어떤 인코딩도 없을 경우에 기본값으로 사용한다.
Client Override	지정된 Encoding이 없을 경우 클라이언트에서 보내는 Content-Type 헤더의 charset을 사용한다.
Forced	해당 인코딩을 모든 경우에 항상 강제적으로 사용한다.

문자 인코딩(Character Encoding) 설정 및 호환 가이드

JEUS는 애플리케이션 개발자에게 개발 편의성을 제공하고 애플리케이션 사용자 또는 서비스 관리자에게 관리 편의성 제공을 위해 XML을 통해 인코딩 설정을 제공해 왔다. 그러나 이에 대한 정책을 결정하고 구현해 나가는 과정에서 Request Encoding과 Response Encoding 간의 forced 설정 의미가 서로 일치하지 않거나 서블릿 표준과 맞지 않는 문제점들이 나타났다.



JEUS는 JEUS 7부터 정책을 일관성 있게 유지하고 바로 잡고자 하였다. 이로 인해서 기존에 작성한 애플리케이션들의 동작 호환성에 문제가 발생할 수 있다. 이 가이드를 통해서 애플리케이션을 수정하거나 호환성 옵션을 이용해서 동작을 유지하길 바란다. 호환성 옵션을 사용할 때는 jeus-web-dd.xml에 설정하기를 권장한다. domain.xml에 서버별로 설정하면 해당 서버에 디플로이하는 모든 웹 애플리케이션이 영향을 받는다.

• Request Url Encoding

- JEUS는 HTTP Request의 Accept-Language 헤더를 참조하지 않는다. 이를 위한 호환성 옵션은 없다.
- <request-url-encoding>의 forced 인코딩은 Request URL에 포함된 Query String을 파라미터로 처리할 때만 사용한다.
- 서블릿이 POST 요청(Content-Type: application/x-www-form-urlencoded)에 대해 ServletRequest.getParameter()를 호출했을 때, Query String을 HTTP 바디로 포함시켜서 처리할 것인지 아니면 원래 의미대로 HTTP 헤더로 볼 것인지에 따라서 <request-url-encoding>의 forced 인코딩 사용 여부를 결정해야 한다.

Query String을 무조건 HTTP 바디와 구분하는 경우 설정하고, 바디의 일부라고 해석한다면 설정하지 않으면 된다. 하지만 GET 요청의 경우에는 Request URL에 포함된 Query String만 파라미터로 처리하기 때문에 결국 <request-url-encoding>의 forced 인코딩 설정이 필요하다.

- 참고로 Query String을 파라미터로 만드는 시점은 서블릿이 최초로 ServletRequest의 Parameter API를 호출했을 때이다.

• Request Encoding

- Query String, Cookie, 요청 바디에 적용된다.
- Query String의 경우 <request-url-encoding><forced>를 설정했을 경우에는 영향받지 않는다.

- Cookie의 경우 <cookie-policy><write-value-on-header-policy>에 charset-encoding 설정이 최우선시된다. JEUS 6까지 사용하던 jeus.servlet.request.cookie.encoding 및 jeus.servlet.urldecode.cookie 프로퍼티는 더이상 제공하지 않는다.
- JEUS 7 Fix#2 이전 버전에서 <request-encoding><forced>를 사용했다면 <request-encoding><client-override>로 대체한다.
- JEUS 6는 request.setCharacterEncoding()의 적용 범위를 HTTP 바디가 아니라 Query String 및 Cookie까지 확대 해석한 문제점을 가지고 있다. 그러나 서블릿 표준은 명백하게 해당 API 적용 범위를 HTTP 바디로 한정하고 있다. JEUS 7부터는 이를 바로 잡았으나 만약 JEUS 6 동작을 유지해야 한다면 jeus.servlet.request.6CompatibleSetCharacterEncoding 프로퍼티를 true로 지정해야 한다. 단, 해당 애플리케이션의 jeus-web-dd.xml에 설정하기를 권장한다.
- **<request-encoding><forced>는 설정하지 않기를 권장한다.**

웹 애플리케이션 개발자는 client-override 또는 default를 설정하면 프로그램을 작성할 때마다 request.setCharacterEncoding()을 호출하지 않아도 된다. 참고로 웹 애플리케이션에서 아래와 같은 방식으로 new String()할 때 인코딩 값을 줘서 직접 String 객체를 생성하는 경우가 있다.

```
String param = request.getParameter();
String realParam = new String(param.getBytes("ISO-8859-1"), "EUC-KR");
```

위에 예제에서 웹 엔진은 기본 인코딩인 ISO-8859-1로 처리하도록 설정해야 param 객체가 ISO-8859-1로 생성된다. 이때 realParam String 객체는 웹 애플리케이션이 JVM을 통해 만든 것이며, 웹 엔진은 관여하지 않는다.

- **JEUS 7 Fix#4부터 jeus-web.dd.xml에 <request-encoding><url-mapping> 설정이 추가되었다.**

jeus-web.dd.xml에 <request-encoding> 설정하게 되면 domain.xml의 <request-encoding> 설정을 치환한다.

- jeus-web-dd.xml에 <url-mapping> 설정을 하더라도 <forced> 또는 <client-override> 설정을 같이 하게 되면 <url-mapping> 설정은 무시된다.
- <url-mapping> 설정만 하게 되면 <servlet-path>에 지정된 요청들만 매핑된 인코딩이 적용된다. <servlet-path>에 매핑 안 된 것들은 서블릿 표준 기본값인 ISO-8859-1이 적용된다.
- <url-mapping> 설정과 <default>를 같이 사용하면 <servlet-path>에 매핑 안 된 것들은 <default> 설정으로 적용된다.
- domain.xml의 <request-url-encoding><forced> 설정이 되어 있을 경우에는 POST 바디에만 이 설정이 적용된다.
- 클라이언트가 보낸 Content-Type 헤더의 charset보다 우선 순위가 낮다.

Request Encoding 설정 예제 : <jeus-web-dd.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<jeus-web-dd>
  <encoding>
    <request-encoding>
      <url-mapping>
        <servlet-path>/test/test1</servlet-path>
        <encoding>EUC-KR</encoding>
```

```

        </url-mapping>
        <url-mapping>
            <servlet-path>/test/*</servlet-path>
            <encoding>UTF-8</encoding>
        </url-mapping>
        <url-mapping>
            <servlet-path>*.jsp</servlet-path>
            <encoding>EUC-KR</encoding>
        </url-mapping>
    </request-encoding>
</encoding>
</jeus-web-dd>

```

• Response Encoding

- 응답 바디에 적용된다.
- Cookie의 경우 Response Encoding의 영향을 받지 않는다. <cookie-policy><write-value-on-header-policy>에 charset-encoding 설정이 최우선시된다.
- <response-encoding><forced>는 response.setCharacterEncoding(), setContentType(), setLocale()보다 우선순위가 높다. 단, JEUS 6 및 JEUS 7 Fix#1까지는 forced의 우선순위 적용 정책이 명확하지 않았다. 그로 인해서 구현이 잘못된 부분이 나타났는데 이러한 동작에 의존해서 작성한 애플리케이션을 유지하고자 한다면, jeus.servlet.response.6CompatibleForcedEncoding 프로퍼티를 true로 설정한다. 단, 해당 애플리케이션의 jeus-web-dd.xml에 설정하기를 권장한다.
- JEUS 6까지는 forced를 설정했더라도 response.setCharacterEncoding()은 우선순위가 가장 높았다. 이러한 동작을 유지하고자 한다면 jeus.servlet.response.6CompatibleSetCharacterEncoding 프로퍼티를 true로 설정한다. 단, 해당 애플리케이션의 jeus-web-dd.xml에 설정하기를 권장한다.
- **<response-encoding><forced>는 설정하지 않기를 권장한다.**

웹 애플리케이션 개발자의 경우 default 설정을 사용하면 매번 인코딩을 설정해야 하는 수고를 덜 수 있다. default 설정으로 커버가 안 될 경우에는 response API로 Response Encoding을 설정하는 것이 바람직하다.

• JSP

- JSP 표준에 의거하여 JSP는 두 가지 종류의 문자 인코딩이 존재한다. 바로 Page Character Encoding과 Response Character Encoding이다.
- Page Character Encoding은 JSP 파일에 대한 인코딩이다. 즉, 파일 시스템에서 그 파일을 읽을 때 사용할 인코딩이다. 이는 기본적으로 Response Character Encoding과는 명확하게 구분된다. 파일의 인코딩이 UTF-8이라고 하더라도 그 JSP를 수행해서 만든 HTTP 응답은 EUC-KR로 내보낼 수 있기 때문이다. 단, Page Character Encoding을 알 수 없을 경우에는 Response Character Encoding을 참조하도록 되어 있다. 만약 그것도 알 수 없는 경우에는 ISO-8859-1로 파일을 읽는다. 이러한 Page Character Encoding을 결정하는 우선순위에 관한 내용은 JSP 표준에 설명되어 있다. 대부분 아래 예제와 같이 JSP 파일마다 명확하게 설정하는 것이 바람직하다.

Page Character Encoding이 설정된 JSP 예제 일부 : <sample.jsp>

```
<%@ page pageEncoding="UTF-8"%>
```

- Response Character Encoding은 <page> 태그의 contentType 속성에 적힌 charset 값을 의미한다.

만약 이 값이 없는 경우에는 Page Character Encoding을 사용한다. Page Encoding도 알 수 없는 경우 JSP 표준에서 정한 기본값은 없다. 이는 웹 컨테이너 설정 또는 동작에 따라 결정된다.

Response Character Encoding이 설정된 JSP 예제 일부 : <sample2.jsp>

```
<%@ page contentType="text/html; charset=UTF-8"%>
```

- 위 두 값을 모두 설정하는 것이 바람직하며 일괄적으로 적용하고 싶은 경우에는 web.xml에 아래와 같은 설정을 추가한다.

Page 및 Response Character Encoding이 설정된 web.xml : <web.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
  metadata-complete="false"
  version="5.0">
  <jsp-config>
    <jsp-property-group>
      <url-pattern>*.jsp</url-pattern>
      <page-encoding>UTF-8</page-encoding>
      <default-content-type>text/html; UTF-8</default-content-type>
    </jsp-property-group>
  </jsp-config>
</web-app>
```

- <response-encoding>은 HTTP Response를 다루는 설정이며 JSP Page Character Encoding과는 명백하게 구분된다. 하지만 이러한 사항을 모두 이해하고 JSP를 작성하는 개발자는 거의 없을 것으로 판단된다. 이에 따라 <response-encoding><forced>는 **JSP Page Character Encoding과 Response Character Encoding에 대해 항상 우선한다.**

압축 인코딩

요청 및 응답 바디가 클 경우, 압축하여 주고 받을 수 있다. 이러한 압축 인코딩은 HTTP 스펙에 명시된 압축 협상을 통해 진행한다.

• 요청 압축 인코딩

요청 바디를 압축하여 전송받으려면, 서버는 응답 헤더에 Accept-Encoding을 추가하여 압축된 바디를 수신할 수 있음을 클라이언트에 알려야 한다. Keep-Alive 상태에서 클라이언트는 이후 요청부터 응답의 Accept-Encoding에 명시된 압축 인코딩으로 요청 바디를 압축하여 전송할 수 있다. 현재 JEUS는 gzip, deflate, identity 3가지 방식을 지원하며, 각 인코딩마다 quality 값을 0.0에서 1.0 사이로 지정할 수 있다. quality를 지정하지 않은 경우 가장 높은 우선순위를 가진다.

Accept Encoding이 설정된 jeus-web-dd.xml : <jeus-web-dd>

```
<?xml version="1.0" encoding="UTF-8"?>
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <accept-encodings>
    <accept-encoding>
```

```

        <compression>gzip</compression>
    </accept-encoding>
    <accept-encoding>
        <compression>deflate</compression>
        <quality>0.5</quality>
    </accept-encoding>
</accept-encodings>
</jeus-web-dd>

```

• 응답 압축 인코딩

응답 바디를 압축하여 전송하려면, 압축 협상에 따라 요청 헤더인 Accept-Encoding에 서버가 지원하는 압축 인코딩이 명시되어 있어야 한다. JEUS는 응답 크기에 따라 압축 여부를 결정할 수 있으며, mime-type에 따라 압축 인코딩 방식을 다르게 적용할 수 있다. 현재 JEUS는 gzip, deflate, identity 3가지 방식을 지원하며, mime-type에서 하나 이상의 subtype을 지정하려면 '*'를 사용할 수 있다. Accept-Encoding에 지정한 content encoding이 없거나, "identity"의 우선순위가 가장 높다면 압축을 진행하지 않는다. 압축 인코딩의 우선순위는 Accept-Encoding에서 압축 인코딩 존재 여부와 quality로 결정되며, quality가 동일할 경우에는 jeus-web-dd에서 상위에 정의된 압축 인코딩을 따른다.

Content Encoding이 설정된 jeus-web-dd.xml : <jeus-web-dd>

```

<?xml version="1.0" encoding="UTF-8"?>
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <content-encodings>
        <compression-min-size>8192</compression-min-size>
        <content-encoding>
            <mime-type>text/*</mime-type>
            <compression>deflate</compression>
            <compression>gzip</compression>
            <compression>identity</compression>
        </content-encoding>
        <content-encoding>
            <mime-type>image/png</mime-type>
            <compression>identity</compression>
        </content-encoding>
        <content-encoding>
            <mime-type>image/jpeg</mime-type>
            <compression>gzip</compression>
        </content-encoding>
    </content-encodings>
</jeus-web-dd>

```

1.6.6. JSP 엔진 설정

웹 엔진은 JSP 엔진을 포함하고 있다. 따라서 JSP에 관한 설정은 웹 엔진 또는 웹 애플리케이션별로 해야 한다. 이에 대한 자세한 내용은 [JSP 엔진](#)을 참고한다.

1.6.7. 응답 헤더 설정

사용자 임의의 HTTP Response Header를 이름과 값의 짝으로 정의할 수 있다.

domain.xml을 사용하여 응답에 기본적으로 포함될 사용자 정의 헤더를 설정하는 방법은 다음과 같다.

```
<web-engine>
...
<response-header>
  <custom-header>
    <header-field>
      <field-name>HeaderName1</field-name>
      <field-value>HeaderValue1</field-value>
    </header-field>
  </custom-header>
</response-header>
...
</web-engine>
```

항목	설명
Custom Header	HTTP 응답 메시지에 포함하기 위한 커스텀 필드를 정의한다. ◦ Header Field : 사용자 정의 헤더를 설정한다. 1개 이상의 응답 헤더를 설정할 경우에는 각 헤더들의 헤더명과 헤더값을 등호(=)로 연결하고, 라인별로 구분한다.

1. 저장에 정상적으로 처리되면 저장이 완료되었다는 결과가 나타난다. 응답 헤더 설정은 동적 반영이 되지 않으므로 서버를 재시작해야 설정 내용이 반영된다.

1.6.8. 쿠키 정책 설정

HTTP Request Header의 쿠키를 읽거나 애플리케이션이 생성한 새로운 쿠키를 HTTP 응답으로 보낼 때 적용하는 정책을 설정할 수 있다. 쿠키 정책 설정은 웹 엔진에 대하여 domain.xml을 통하여 설정할 수 있고, 또한 각 애플리케이션별로 jeus-web-dd.xml에서 설정 가능하다.

domain.xml을 사용하여 쿠키 정책을 설정하는 방법은 다음과 같다. 쿠키 정책으로 UTF-8 인코딩을 사용하여 쿠키 인코딩 수행을 설정한 예시이다.

```
<web-engine>
  <cookie-policy>
    <write-value-on-header-policy>
      <apply-url-encoding-rule>true</apply-url-encoding-rule>
      <charset-encoding>UTF-8</charset-encoding>
    </write-value-on-header-policy>
  </cookie-policy>
</web-engine>
```

항목	설명
Apply Url Encoding Rule	URL Encoding Rule의 적용 여부를 설정한다.
Charset Encoding	URL Encoding Rule을 적용 여부에 상관없이 쿠키를 응답에 쓰거나 해석할 때 사용하는 Character Encoding이다. 설정하지 않을 경우 Request Encoding의 값을 따른다.

쿠키 정책 설정은 동적 반영이 되지 않으므로 서버를 재시작해야 설정 내용이 반영된다.

1.6.9. 세션 설정

웹 엔진의 세션 관리를 위한 세션 객체의 공유 여부, 세션 쿠키 설정, 타임아웃 등 웹 엔진의 세션과 관련된 모든 사항을 설정한다.

- 웹 엔진 레벨

콘솔 툴을 사용해서 세션을 설정한다. 세션 설정에 대한 자세한 내용은 JEUS 세션 관리 안내서의 "세션 설정"을 참고한다.

- 컨텍스트 레벨

jeus-web-dd.xml의 <jeus-web-dd> 하위에 설정한다.

웹 엔진 레벨에서 세션이 설정되어 있다면, 컨텍스트 레벨에서 세션을 설정하지 않아도 공통적으로 웹 엔진의 세션 설정이 적용된다. 웹 엔진 레벨, 컨텍스트 레벨의 설정이 중복되었을 경우 하위 레벨의 세부적인 설정에 가장 높은 우선순위를 부여한다. 즉, 모두 설정이 되어있다면 **컨텍스트 레벨 > 웹 엔진 레벨** 순으로 설정값이 적용된다. 만약 하위 레벨에서 특정 설정을 하지 않았다면 상위 레벨의 설정값을 적용하고 모든 레벨에서 설정하지 않은 값은 엔진 내부적인 기본값으로 동작한다.



클러스터링된 환경에서의 세션 관련 사항에 대한 자세한 내용은 JEUS 세션 관리 안내서의 "세션 트래킹"을 참고한다.

1.6.10. 로그 설정

본 절에서는 각 로그들의 설정 방법에 대해 설명한다.

- 서버 로그 설정
- 액세스 로그 기본 설정
- 가상 호스트별 액세스 로그 설정
- 액세스 로그 포맷 설정
- 액세스 로그 필터 설정
- 유저 로그 설정

서버 로그 설정

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

액세스 로그 기본 설정

웹 엔진 액세스 로그는 콘솔을 통해 설정한다. 특별히 설정하지 않아도 기본적으로 액세스 로그를 파일로 남기도록

설정되어 있다.



JEUS 6 Fix#8부터는 액세스 로그도 기본적으로 로그 로테이션을 적용한다.

로그 로테이션이란 JEUS에서 제공하는 기능으로 JEUS가 시작될 때 또는 JEUS 운영 중에 설정한 시간이 되거나 파일 사이즈를 넘으면 자동으로 이전에 Logging하던 파일의 이름은 바꾸고 새로 출력할 로그들은 원래의 로그 파일에 계속 Logging할 수 있는 기능이다.

domain.xml을 사용한 액세스 로그 기본 설정 방법은 다음과 같다.

Access log의 xml 설정 : <domain.xml>

```
<domain xmlns="...">
  ...
  <servers>
    <server>
      <web-engine>
        <access-log>
          <handler>
            <file-handler>
              <name>accessLogFileHandler</name>
              <level>FINEST</level>
              <enable-rotation>true</enable-rotation>
              <valid-day>1</valid-day>
              <buffer-size>1024</buffer-size>
              <append>true</append>
            </file-handler>
          </handler>
          <enable>true</enable>
          <format>default</format>
          <enable-host-name-lookup>false</enable-host-name-lookup>
        </access-log>
      </web-engine>
    </server>
  </servers>
  ...
</domain>
```

기본 설정을 사용할 경우에는 특별히 수정할 필요가 없다. 주로 액세스 로그의 포맷을 변경할 경우나 특정 확장자의 접근을 액세스 로그에 포함시키지 않을 경우에 필요한 설정을 할 수 있다. 기본적으로 존재하는 파일 핸들러 이외에 다른 핸들러를 추가하려면 **<handler>**에서 추가한다. 핸들러에 대한 자세한 내용은 JEUS Server 안내서의 "Logging"을 참고한다.

다음은 **고급 선택사항**의 항목에 대한 설명이다.

항목	설명
Enable Host Name Lookup	%h 포맷에 대해 IP 주소 대신 호스트 이름을 로깅할 것인지 결정한다. 이값을 true로 할 경우 DNS lookup으로 인한 오버헤드가 발생할 수 있다.
Exclude Ext	Access 로그를 남기지 않는 확장자들을 설정한다. 여러 개를 입력하고 싶은 경우 콤마(,)로 구분한다.

항목	설명
Filter Class	Logger에 지정할 필터 클래스의 이름을 설정한다. 동적 반영이 되지 않으므로 서버를 재시작해야 설정 내용이 반영된다.

가상 호스트별 액세스 로그 설정

가상 호스트별로 액세스 로그를 남기려면 각 가상 호스트 설정에 액세스 로그를 설정해야 한다. 이에 대한 자세한 내용은 [가상 호스트 설정](#)을 참고한다.

액세스 로그 포맷 설정

domain.xml을 사용해서 액세스 로그의 포맷 Alias를 설정할 수 있다.

```
<access-log>
  ...
  <enable>true</enable>
  <format>default</format>
</access-log>
```

포맷에는 기본적으로 '%' 기호를 이용해서 데이터를 나타내며, 임의의 문자열도 가능하다.



- JEUS 9는 Common Log Format(이하 CLF)을 따른다. CLF에 대한 자세한 사항은 <http://httpd.apache.org/docs/2.4/logs.html>을 참고한다.
- 콘솔 툴을 이용한 포맷 설정 변경은 JEUS Reference 안내서의 "modify-web-engine-configuration"을 참고한다.

웹 엔진은 다음과 같은 포맷 Alias를 제공한다.

- default

JEUS에서 기본적으로 제공하는 포맷이다. CLF에서 가장 많이 사용하는 common 포맷에서 처리 시간(%D)를 추가한 포맷이다. 성능상의 이유로 항상 '-'가 나오는 %i과 세션에서 값을 읽어오는 %u를 삭제하였다.

```
%h %t \"%r\" %s %b %z %D
```

- common

CLF에서 가장 많이 사용하는 포맷으로 아래의 포맷을 Alias한다.

```
%h %l %u %t \"%r\" %s %b
```

- combined

HTTP 헤더에서 Referer와 User-agent를 찍어주는 포맷으로 아래의 포맷을 Alias한다.

```
%h %l %u %t \"%r\" %s %b \"%{Referer}i\" \"%{User-agent}i\"
```

- debug

default 포맷에 %S(세션 ID)와 %I(요청 처리 스레드 이름)을 추가한 포맷이다.

```
%h %t \"%r\" %s %b %z %D %u %S \"%I\"
```

위의 포맷 Alias를 이용해서 다음과 같은 형식으로 포맷을 지정할 수도 있다.

```
default %S
```

이 경우에 웹 엔진에서 common을 치환해서 다음과 같이 등록한다.

```
%h %l %u %t \"%r\" %s %b
```

이 포맷 설정은 서비스 도중에도 변경이 가능하며, 그 변경 사항이 적용된다. 따라서 현재 제공하는 서비스에 문제가 있는 경우 debug 포맷을 설정해서 세션 ID, 응답 처리 Thread 이름을 액세스 로그에 남길 수 있다.

액세스 로그 필터 설정

액세스 로그의 필터 기능은 액세스 로그의 양이 지나치게 많을 경우 특정 형식 또는 조건을 만족하는 내용만 로그로 기록하고 싶을 경우를 위해 제공되는 기능이다. <exclude-ext>를 통해서 특정 확장자로 끝나는 요청에 대해서 필터링을 지원하고 있다.



JEUS 7부터는 jeus.servlet.util 패키지에서 jeus.servlet.logger 패키지로 변경되었다.

JEUS에서는 jeus.servlet.logger.AccessLoggerFilter 인터페이스 및 jeus.servlet.logger.AbstractAccessLoggerFilter 추상 클래스를 제공하고 있다.

jeus.servlet.logger.AccessLoggerFilter 인터페이스의 상세 내용 : <AccessLoggerFilter.java>

```
package jeus.servlet.logger;

import java.util.logging.Filter;
import java.util.logging.LogRecord;

/**
 * Access Logger의 내용을 필터링하기 위해 필요한 정보들을 제공하는 인터페이스.
 * {@link Filter}를 상속한 필터 인터페이스로서 부수적으로 제공되는 인터페이스를 통해
 * {@link Filter#isLoggable(java.util.logging.LogRecord)}안에서 다양하게 필터링 정책을 정하도록
 * 가이드한다.
 */
public interface AccessLoggerFilter extends Filter {
```

```

/**
 * 서버에 접속한 remote client의 address 정보를 반환한다.
 *
 * @param record a LogRecord
 * @return remote client의 address. 만약 올바른 값을 얻지 못할 경우 null을 반환한다.
 */
public String getRemoteAddr( LogRecord record );

/**
 * 요청의 메소드를 반환한다. ex) GET, POST, PUT, etc...
 *
 * @param record a LogRecord
 * @return request method. 만약 올바른 값을 얻지 못할 경우 null을 반환한다.
 */
public String getMethod( LogRecord record );

/**
 * 요청의 uri를 반환한다.
 *
 * @param record a LogRecord
 * @return request uri. 만약 올바른 값을 얻지 못할 경우 null을 반환한다.
 */
public String getRequestURI( LogRecord record );

/**
 * 응답의 status값을 반환한다. ex) 200, 404
 *
 * @param record a LogRecord
 * @return status.
 */
public int getStatus( LogRecord record );

/**
 * 요청에 따른 처리시간을 ms단위로 반환한다.
 *
 * @param record a LogRecord
 * @return processing time. 만약 올바른 값을 얻지 못할 경우 -1을 반환한다.
 */
public long getProcessingTimeMillis( LogRecord record );
}

```

사용자가 액세스 로그의 특정 패턴에 대해 필터링하는 경우 위의 AccessLoggerFilter 인터페이스 및 AbstractAccessLoggerFilter 추상 클래스를 이용하여 쉽게 필터를 구현하고 적용할 수 있다.

사용자의 필터 클래스는 jeus.servlet.logger.AbstractAccessLoggerFilter를 상속하고, java.util.logging.Filter의 isLoggable() 메소드를 구현해야 한다. isLoggable() 메소드를 구현할 때, 위의 jeus.servlet.logger.AccessLoggerFilter의 API 중에서 사용자가 필요한 정보를 적절히 선택하여 이용 및 구현한다.

다음은 요청 확장자가 .gif인 경우 액세스 로그로 기록하지 않는 필터 클래스를 정의한 예이다.

필터 클래스 정의 예제 : <SimpleAccessLoggerFilter>

```

package sample;

import jeus.servlet.logger.AbstractAccessLoggerFilter;
import java.util.logging.*;

```



```
public class SimpleAccessLoggerFilter extends AbstractAccessLoggerFilter {
    public boolean isLoggable(LogRecord record) {
        String requestURI = getRequestURI(record);
        return requestURI != null && !requestURI.endsWith(".gif");
    }
}
```

- sample.SimpleAccessLoggerFilter는 사용자가 정의한 클래스이며 jeus.servlet.logger.AbstractAccessLoggerFilter를 extends한다.
- java.util.logging.Filter#isLoggable() 메소드를 구현하였으며, 클라이언트의 요청 URI를 얻기 위해 jeus.servlet.util.AccessLoggerFilter#getRequestURI() API를 이용한다.

클래스 정의가 완료되면 해당 클래스를 컴파일한다. 그 다음 JAR 파일 형태로

JEUS_HOME/domains/DOMAIN_HOME/lib/application 디렉터리에 포함시키고, 웹 엔진 액세스 로그 설정에 해당 필터를 설정한다. 액세스 로그 설정 방법은 "[액세스 로그 기본 설정](#)"을 참고한다.

유저 로그 설정

아무런 설정을 하지 않을 경우에는 서버 로그 파일(*JeusServer.log*)에 로그가 남는다. 만약 별도의 로그 파일을 남기고 싶은 경우에는 유저 로그를 설정한다. 유저 로그의 기본 위치 및 파일은 [디렉터리 구조](#)를 참고한다.

유저 로그는 서버 로그와 마찬가지로 JEUS 서버 레벨에서 등록할 수 있고, 웹 애플리케이션의 jeus-web-dd.xml에 등록할 수도 있다. 또한 각 웹 애플리케이션별로 남길 수도 있고, 특정 로그 파일에 모아서 남길 수도 있다.

domain.xml을 사용한 유저 로그 설정 방법은 다음과 같다.

<user-logging>의 하위 태그에서 설정한다.

다음은 webapp이라는 이름의 웹 애플리케이션만의 로그를 생성하기 위해 '**<name>**' 항목을 설정한 예이다.

webapp 이름 없이 'jeus.systemuser'를 입력하면 모든 웹 애플리케이션에 유저 로그가 적용된다. 고급 선택사항의 각 항목에 대한 설명은 "[액세스 로그 기본 설정](#)"을 참고한다. 그 외에 '**<level>**' 항목 등은 필요한 경우 설정한다.

<handler>에서 핸들러를 추가할 수 있다. 추가할 핸들러의 태그를 입력한 후 유저 로그 핸들러를 추가한다. 각 핸들러의 설정 방법은 JEUS Server 안내서의 "Logging"을 참고한다. 본 예제에서는 파일 핸들러 추가를 위해 **<file-handler>** 태그를 추가한다.

```
<server>
...
<user-logging>
    <name>jeus.systemuser.webuser.app1</name>
    <level>INFO</level>
    <use-parent-handlers>true</use-parent-handlers>
    <filter-class>...</filter-class>
    <formatter-pattern></formatter-pattern>
    <handler>
        <file-handler>
            <name>fileHandler</name>
            <level>FINEST</level>
        </file-handler>
    </handler>
```

```
</user-logging>
...
</server>
```

1.6.11. Async Servlet 타임아웃 처리 설정

Servlet 3.0부터 추가된 Async Servlet의 타임아웃 처리를 위해서 필요한 Thread 개수를 설정해야 한다.

domain.xml을 사용한 Async Servlet 타임아웃 처리 설정 방법은 다음과 같다.

```
<web-engine>
  <async-timeout-min-threads>10</async-timeout-min-threads>
</web-engine>
```

Async Servlet 타임아웃 처리 설정은 동적 반영이 되지 않으므로 서버를 재시작해야 설정 내용이 반영된다.

1.6.12. 웹 엔진 레벨의 프로퍼티 설정

domain.xml을 사용해서 JEUS에서 정의한 프로퍼티들을 설정할 수 있다. 웹 엔진 프로퍼티에 대한 자세한 내용은 JEUS Reference 안내서의 "웹 엔진 프로퍼티"를 참고한다.

domain.xml을 사용한 프로퍼티 설정 방법은 다음과 같다.

'<properties>' 태그를 설정한다. 프로퍼티 이름은 <key>, 값을 <value>로 입력한다.

```
<web-engine>
  <properties>
    <property>
      <key>jeus.servlet.request.enableDns=false</key>
      <value>false</value>
    </property>
  </properties>
</web-engine>
```



프로퍼티 설정은 '<properties>' 태그를 통해 설정하는 것을 권장한다. domain.xml에서 '<jvm-option>' 태그를 통한 설정도 가능하다.

1.6.13. 웹 공격 대응 설정

사용자가 악의적인 목적으로 JEUS에 요청 처리 부담을 주기 위해 일정 크기 이상의 요청들을 대량으로 보내는 공격을 받으면 정상적인 요청들의 응답이 늦어지거나 요청을 처리하지 못하게 되는 현상이 발생한다.

이런 현상을 방지하기 위하여 JEUS 관리자가 JEUS 서버를 기준으로 악의적인 요청에 대한 기준을 설정하여, 이들의 요청 처리로 인한 서버의 부담을 덜어줄 수 있다. 이 설정들은 기본적으로 서버의 웹 엔진 내부에 설정되며, 웹 엔진의

요청을 받아들이는 리스너나 커넥터별로 설정한다. 즉, HTTP 요청을 받아들이는 HTTP 리스너, WebtoB 커넥터, AJP13 커넥터별로 설정한다. 각 리스너나 커넥터별 설정은 [리스너 공통 설정](#)을 참고한다.

제한하려는 웹 공격에 대해 다음의 설정들을 적절하게 조합하여 설정하면 웹 공격에 대응할 수 있다.

domain.xml을 사용한 웹 공격 대응 설정 방법은 다음과 같다.

```
<domain xmlns="..." version="...">
  <servers>
    <server>
      <web-engine>
        <web-connections>
          <http-listener>
            <name>SERVER-HTTP</name>
            <thread-pool>
              <min>10</min>
              <max>20</max>
              <max-idle-time>300000</max-idle-time>
              <max-queue>-1</max-queue>
            </thread-pool>
            <postdata-read-timeout>600000</postdata-read-timeout>
            <max-post-size>-1</max-post-size>
            <max-parameter-count>-1</max-parameter-count>
            <max-header-count>-1</max-header-count>
            <max-header-size>8192</max-header-size>
            <max-querysting-size>8192</max-querysting-size>
            <server-access-control>false</server-access-control>
            <allowed-server>...</allowed-server>
            <server-listener-ref>http-server</server-listener-ref>
          </http-listener>
        </web-connections>
      </web-engine>
    </server>
  </servers>
</domain>
```

domain.xml에서 각 항목을 설정한다. xml 항목의 설정을 통해 각 리스너별로 웹 공격에 대하여 대응을 할 수 있다. 각 설정 항목에 대한 자세한 설명은 [리스너 공통 설정](#)을 참고한다.

1.6.14. 특정 URL 패턴의 HTTP 요청 방지

특정 URL 패턴의 HTTP 요청 방지를 위해 HTTP 클라이언트가 다음과 같은 요청을 보냈다고 가정한다.

```
GET /examples/%2e%2e%2fdb.txt HTTP/1.1
```

웹 컨테이너는 먼저 요청 URI를 URL Decode해야 한다.

```
/examples/../../db.txt
```

이렇게 되면 실제 HTTP 서비스와는 관련없는 파일들을 접근할 수 있게 된다. 주로 악의적인 목적을 가진 클라이언트가 이런 식으로 요청 URI을 보낸다.

웹 컨테이너에서 모든 악의적인 요청 패턴을 파악할 수 없으므로 사용자가 직접 악의적인 패턴을 막을 수 있도록 설정을 제공한다. HTTP 요청인 경우에만 동작하며, Query String을 제외한 URI에 대해 특정 문자열을 매칭해서 무조건 "404 Not Found"로 처리한다.



JEUS 6 및 JEUS 7 Fix#1까지는 프로퍼티 파일을 사용했지만, JEUS 7 Fix#2부터는 domain.xml에 설정한다.

domain.xml을 통한 설정 방법은 다음과 같다.

<blocked-url-patterns>의 하위 태그를 설정한다.

```
<web-engine>
...
<blocked-url-patterns>
  <encoded-pattern>%00</encoded-pattern>
  <decoded-pattern>#</decoded-pattern>
  <deny-last-space-character>true</deny-last-space-character>
  <deny-null-character>true</deny-null-character>
</blocked-url-patterns>
...
</web-engine>
```

'Encoded Pattern' 항목은 HTTP 클라이언트가 보낸 URI에 대해 설정된 문자열을 포함하는지 체크한다. 클라이언트가 보낸 URI는 웹 컨테이너가 URL Decode한 다음에 사용하게 되는데 이때 'Decoded Pattern'에 설정한 값이 포함되어 있는지 체크한다. 'Encoded Pattern' 항목을 하나라도 설정하면 앞서 언급한 기본 패턴에 대해서 처리하지 않는다('Decoded Pattern' 항목도 동일하다).



아무런 설정을 하지 않을 경우 웹 컨테이너는 Encoded Pattern의 경우 %00, %23, %2e, %2f, %5c, Decoded Pattern은 #, \에 대해서 404 응답 처리한다. 패턴은 대소문자를 구분하지 않는다. 항상 소문자로 변환해서 처리한다.

1.7. 웹 엔진 튜닝

웹 엔진의 최적화된 성능을 위해서는 다음의 몇 가지 사항을 고려해야 한다.

- <output-buffer-size>를 적절한 크기로 지정한다. WebtoB와 함께 사용할 경우에는 <webtob-connector>에 <send-buffer-size>와 <receive-buffer-size>를 적절하게 지정한다.
- <check-included-jspfile>는 include된 JSP들이 변경되지 않는다면 설정을 false로 유지한다. 이것은 include된 JSP 파일들에 대한 변경 점검을 하지 않기 때문에 성능 향상에 도움이 된다.
- JSP 파일이 변경되지 않는다면 jeus.servlet.jsp.reload 프로퍼티를 false로 설정한다. 매번 JSP 호출할 때마다 파일 시스템에 메타 데이터를 조회하지 않는다.

2. 웹 커넥션 관리

본 장에서는 웹 엔진에서 제공하는 리스너 또는 커넥터의 관리 및 설정 방법 등에 대해 설명한다.

2.1. 개요

JEUS에서는 웹 리스너와 웹 커넥터를 통칭해서 웹 커넥션이라고 한다. 웹 엔진에서는 WebtoB 및 다른 웹 서버와의 커넥션, HTTP/TCP 클라이언트와의 직접적인 커넥션, 또는 Tmax와의 커넥션을 제공한다. 웹 서버는 클라이언트의 HTTP 요청을 받아 조건에 맞는 경우 웹 엔진으로 요청을 전달한다.

대표적인 웹 서버로 WebtoB와 Apache를 사용하고 연결을 위해 다음과 같은 각각의 커넥터를 제공한다.

- **WebtoB 커넥터**

WebtoB는 연결을 생성할 때 JEUS가 클라이언트 역할을 하기 때문에 WebtoB 커넥터를 제공한다.

- **AJP 리스너**

Apache는 JEUS가 서버 역할을 하기 때문에 AJP Listener를 제공한다. AJP의 경우 IIS, SunOne(Iplanet) 웹 서버와 같은 타사 상용 웹 서버에서도 지원하므로 AJP Listener를 통해서 해당 웹 서버와 연결할 수 있다.

웹 엔진은 클라이언트와 직접 커넥션을 맺고, 관리를 위해 각각의 클라이언트별로 다음과 같은 리스너 및 커넥터를 제공한다.

- **HTTP 리스너**

HTTP 클라이언트와의 직접적인 커넥션 관리를 위해 제공한다. **Async Servlet 및 Servlet NIO, Websocket의 기능을 사용하기 위해서는 HTTP 리스너를 사용해야 한다.**

- **TCP 리스너**

TCP 클라이언트와의 커넥션 관리를 위해 제공한다.

- **Tmax 커넥터**

Tmax와의 연동을 위한 커넥터로 WebtoB 커넥터와 마찬가지로 연결을 생성할 때에는 JEUS가 클라이언트 역할을 한다.

리스너는 JEUS 서버의 설정을 기반으로 동작한다. 따라서 보안(SSL) 리스너를 사용하려면 JEUS 서버에 해당 설정을 하고 이를 각각의 웹 관련 리스너들이 사용하도록 설정한다.



JEUS 서버와 관련된 세부 설정 사항에 대한 자세한 내용은 JEUS Server 안내서의 "Listener 설정"을 참고한다.

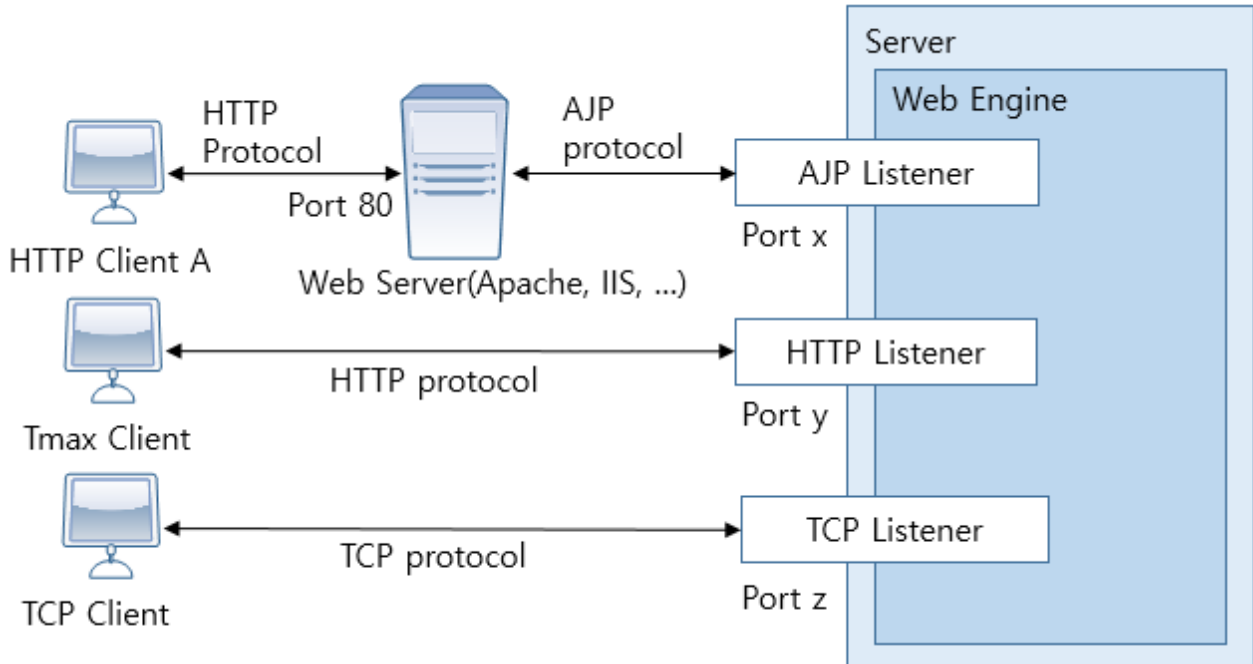
2.2. 구성 요소

본 절에서는 웹 엔진에서 제공하는 리스너 및 커넥터에 대해 설명한다.

2.2.1. 리스너

리스너는 AJP 프로토콜을 따르는 웹 서버, HTTP/TCP 클라이언트가 접근할 수 있는 웹 엔진의 채널이다.

웹 엔진의 리스너와 각 클라이언트, 프로토콜의 연결을 나타내면 다음과 같다.



웹 엔진의 리스너

다음은 각 리스너에 대한 설명이다.

- AJP 리스너

WebtoB 이외의 다른 웹 서버(Apache, IIS, SunOne(Iplanet) 등)를 사용할 경우에도 JEUS 웹 애플리케이션과의 상호적인 연동이 가능하도록 하는 프로토콜이다. mod_jk module을 통해 지원하며 AJP 1.3 프로토콜을 사용한다. AJP 리스너는 SSL을 지원한다. AJP 리스너에 대한 자세한 내용은 [AJP 리스너 설정과 부하 분산을 위한 웹 서버 설정](#)을 참고하고, AJP 리스너에서 사용할 SSL 서버 리스너의 설정은 "JEUS Server 안내서"를 참고한다.

- HTTP 리스너

HTTP 요청을 웹 엔진이 직접 받을 때 사용한다. HTTP 리스너는 SSL을 지원한다. HTTP 리스너 설정의 자세한 내용은 [HTTP 리스너 설정](#)을 참고한다.

HTTP 리스너에서 사용할 SSL 서버 리스너의 설정은 "JEUS Server 안내서"를 참고한다.

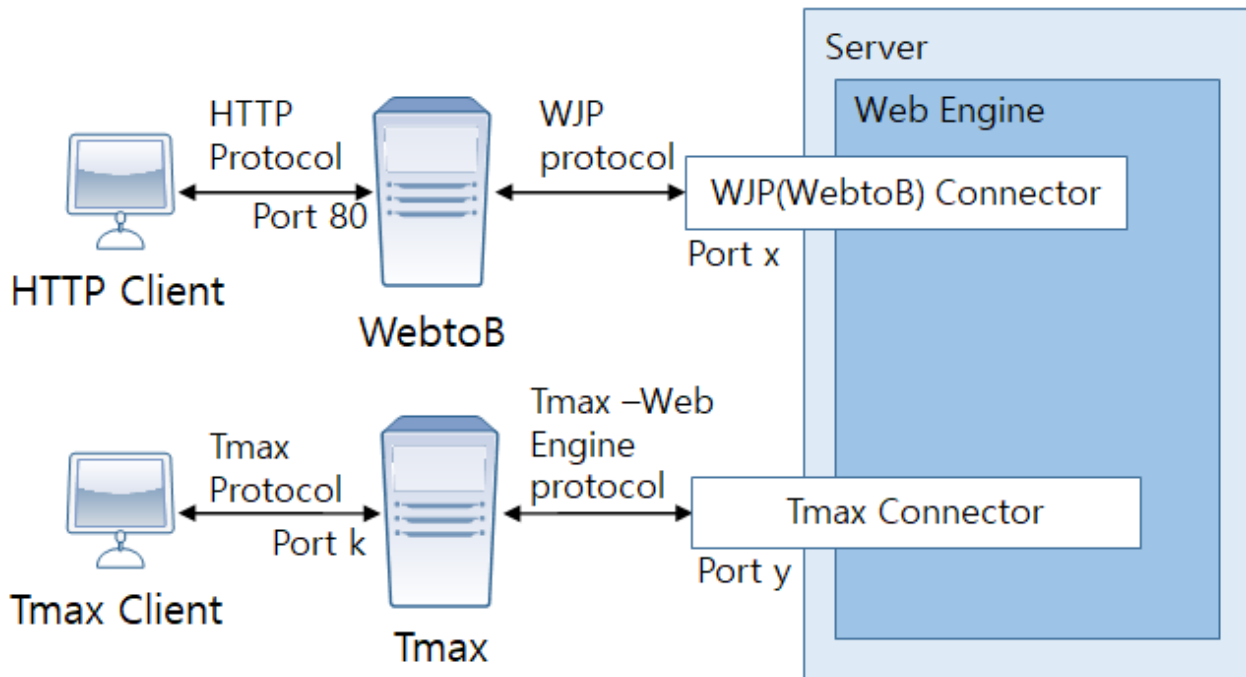
- TCP 리스너

HTTP 프로토콜이 아닌 커스텀 프로토콜로 동작하는 클라이언트에 대한 리스너이다. TCP 리스너에 대한 자세한 내용은 [TCP 리스너 설정](#)과 [TCP 리스너 사용](#)을 참고한다.

2.2.2. 커넥터

커넥터는 웹 엔진에서 WebtoB 및 Tmax에 연결하기 위한 채널이다.

웹 엔진의 커넥터와 각 클라이언트, 프로토콜의 연결을 나타내면 다음과 같다.



웹 엔진의 커넥터

다음은 각 커넥터에 대한 설명이다.

- WJP(WebtoB) 커넥터

WJP는 WebtoB-JEUS Protocol을 의미한다. WebtoB는 TmaxSoft에서 제공하는 웹 서버이다. 별도의 제품으로 구매해서 JEUS와 함께 사용할 수도 있다.

WebtoB 커넥터는 커넥션 생성 시점에 JEUS가 클라이언트 입장이기 때문에 직접 WebtoB로 접속하는 특징을 가진다. 이런 특징에 따라 방화벽 밖에 WebtoB 서버가 있을 경우에는 특별한 방화벽 설정 없이 연결을 맺을 수 있다. 이것은 방화벽이 주로 외부로부터의 연결 시도를 억제하고 내부로부터의 연결은 가능하게 하는 속성을 이용한 것이다. WebtoB 커넥터에 대한 자세한 내용은 [부하 분산을 위한 웹 서버 설정](#)을 참고한다.

- Tmax 커넥터

Tmax는 분산 환경에서 XA 트랜잭션을 관리하는 시스템 소프트웨어이다. Tmax 커넥터 역시 WebtoB 커넥터와 마찬가지로 JEUS가 클라이언트 역할을 한다. Tmax 커넥터는 JEUS와 Tmax 사이의 정보를 주고받거나, HTTP 요청을 Tmax의 게이트웨이를 통해 받음으로써 통신 채널을 일원화하는 용도로 사용할 수 있다. Tmax 커넥터에 대한 자세한 내용은 [Tmax 커넥터 설정](#)을 참고한다.



JEUS가 클라이언트 역할을 하는 것은 연결해서 커넥션을 맺을 때 뿐이다. 실제로 외부 클라이언트의 요청은 WebtoB, Tmax로부터 전달받는 것이며, JEUS 내부적으로 WebtoB나 Tmax에 요청을 보내는 경우는 없다. 즉, 실제 서비스 중에는 JEUS가 서버 역할을 한다.

2.2.3. Worker Thread Pool

리스너 또는 커넥터에는 클라이언트로의 요청을 처리하기 위한 Worker Thread Pool이 존재하는데, 이는 Worker Thread들을 관리하는 개체이다.

JEUS 21 이전에는 리스너 및 커넥터가 Worker Thread Pool을 관리하는 주체였지만, JEUS 21 FIX 1부터는 Web Connection, VirtualHost, Context가 주체가 되어 Worker Thread Pool을 관리한다. 자세한 내용은 [스레드 풀](#)을 참고한다.

리스너의 경우 요청이 오면 그 요청을 처리하기 위한 Worker Thread가 할당된다. HTTP, AJP13, TCP 리스너와 <use-nio>가 true인 WebtoB 커넥터인 경우에는 Worker Thread에 작업을 넘기기 전에 요청을 파싱하여 Context와 Host를 찾기 때문에, Context와 VirtualHost 레벨의 Worker Thread Pool에 넘길 수 있다. 커넥터의 경우 WebtoB 또는 Tmax와 JEUS가 client으로서 커넥션을 맺는다. <use-nio>가 false인 WebtoB와 Tmax 커넥터는 서버 리스너의 부재로 Worker Thread Pool에 할당하기 이전에 요청을 읽을 수 없기 때문에, 미리 커넥터에 종속되는 스레드 풀에서만 요청을 읽고 처리할 수 있다. 주의할 점은 커넥터의 Connection 개수와 스레드 개수가 동일한 구조로 되어 있어야 하기 때문에 따로 Web Connection의 스레드풀을 설정할 때 Connection, 스레드 풀 min, max가 동일한 값으로 설정되어야 한다.

Worker Thread Pool의 최솟값, 최댓값 등은 서비스 처리 성능에 큰 영향을 미치기 때문에 리스너 또는 커넥터를 설정할 때는 Worker Thread Pool을 주의해서 설정해야 한다.



JEUS는 각 Thread Pool이 자신의 상태를 직접 관리하며, 모니터링 Thread는 주기적으로 그 결과만 Logging해주는 방식을 사용한다. 따라서 로그에 남겨진 Thread 수 증감 변화와 실제 Thread Pool의 상태 변화가 서로 일치하지 않을 수 있으니 주의한다.

Active-Management와 상태 통보

Worker Thread Pool에는 Active-Management에 관한 설정이 포함되어 있다. Active-Management는 관리자가 지정한 특정 상태에 이르면 웹 엔진이 경고 메시지를 이메일(e-mail)로 통지하거나 웹 엔진이 속한 서버의 재시작을 권고하는 설정이다. 설정 가능한 값은 Worker Thread가 블록되었다고 판단하는 기준시간이다. 이에 따라 블록된 Worker Thread가 증가할 경우 몇 개 이상이면 경고 이메일 또는 엔진 재시작 권고 메시지 출력과 같은 특정 작업의 수행을 설정한다.



재시작 권고가 출력되면 서버의 요청 처리가 순조롭지 못할 가능성이 크기 때문에 관리자는 메시지를 확인하여 서버의 재시작 여부를 판단해야 한다.

2.3. 웹 커넥션 설정

AJP 리스너, WebtoB 커넥터, Tmax 커넥터를 사용하는 경우 연동 제품별로 설정이 필요하다. 모든 웹 커넥션은 웹 엔진에서 관리하므로 본 절에서는 웹 엔진의 설정에 대해서만 설명한다. WebtoB와 다른 웹 서버의 해당 설정에 대한 자세한 내용은 [부하 분산을 위한 웹 서버 설정](#)에서 설명한다.



1. JEUS 6의 컨텍스트 그룹이 없어졌다. 컨텍스트 그룹이 관리하던 대부분의 사항들은 웹 엔진이 관리한다.

2. 웹 리스너의 경우 서버에서 제공하는 통합된 서비스 리스너를 사용한다. 따라서 AJP, HTTP, TCP 리스너의 경우 서버에 리스너를 설정을 하고, 그 서버 리스너를 참조하는 방식으로 설정한다. 이에 따라 '**Server Listener Ref**'라는 설정이 추가되었다. 하지만 WebtoB, Tmax에 대한 커넥터는 JEUS가 클라이언트로서 직접 연결하므로 기존 설정 방식을 유지한다.

커넥터의 추가, 수정, 삭제는 콘솔 툴을 사용할 수 있다. 콘솔 툴을 사용하여 설정하는 방법은 JEUS Reference 안내서의 "웹 엔진 관련 명령어"를 참고한다.

2.3.1. 리스너 공통 설정

본 절에서는 각 리스너에서 공통적으로 사용하는 주요 설정이다.

- **Name**

- 웹 커넥션을 식별하는 유일한 이름이다. 웹 엔진 내에서 유일해야 하고, 반드시 설정해야 한다.

- **Server Listener Ref**

- 해당 리스너가 참조하는 서버 리스너를 나타낸다.
- HTTP, AJP 리스너의 경우 서로 같은 서버 리스너를 공유할 수 있다. 단, HTTP 리스너를 관리 목적으로 사용하겠다고 설정한 경우에는 함께 사용할 수 없다. 설정하지 않은 경우에는 서버의 기본 리스너를 사용한다.
- TCP 리스너는 함께 사용할 수 없다.
- 2개 이상의 같은 프로토콜 리스너를 함께 사용할 수 없다. 예를 들어 AJP 리스너를 2개 설정했는데 둘 다 이 설정이 없으면 서버 기본 리스너를 공유하게 되므로 설정 에러가 발생한다. 둘 중 하나는 반드시 별도의 서버 리스너를 참조해야 한다.
- 서버 리스너의 <keep-alive-timeout> 설정으로 아무런 요청이 없는 클라이언트 커넥션들을 끊어줄 수 있다. (단위: ms)

- **Connection Type**

- 각 리스너 또는 커넥터를 통해 나가는 응답의 커넥션 헤더를 강제로 설정한다.
- 다음 중 하나를 설정할 수 있다.

설정값	설명
keep-alive	응답을 보낸 후에도 연결을 계속 유지할 경우 설정한다.
close	응답을 보낸 후 연결을 끊을 경우 설정한다.
none	요청 헤더에 정의된 속성에 따라 응답의 커넥션 헤더를 따를 경우 설정한다. 웹 엔진은 아무 설정도 하지 않을 경우 'none'으로 설정된 것과 같이 동작한다.



AJP 리스너의 경우 '**Connection Type**' 항목을 설정할 수 없다. AJP 리스너는 Apache와 같은 웹 서버의 설정을 따를 것을 권장한다.

• Thread Pool

- Web Connection 레벨의 Worker Thread Pool에 대한 설정이다.
- 다음은 세부 사항을 설정하는 하위 항목에 대한 설명이다.

항목	설명
Min	Pool에서 관리할 최소 Worker Thread의 개수이다. WebtoB 커넥터의 <use-nio>가 false인 경우에는 <Connection-Count>와 동일한 값으로 설정해야 한다.
Max	Pool에서 관리할 최대 Worker Thread의 개수이다. WebtoB 커넥터의 <use-nio>가 false인 경우에는 <Connection-Count>와 동일한 값으로 설정해야 한다.
Maximum Idle Time	Pool 내에 존재하던 Thread가 제거되기 전까지의 사용되고 있지 않은 시간을 지정한다. 그 결과 시스템 리소스는 증가한다.
Max Queue	Queue에 대기할 수 있는 요청의 최대 개수를 설정한다. 값이 -1일 경우 Queue 크기에 제한을 두지 않는 것을 의미한다
Thread State Notify	블록되는 Worker Thread가 발생할 경우 이를 알려준다. 각 Thread Pool은 장애가 발생했을 때 취하는 액션을 정의하는 ' Thread State Notify ' 항목을 가지고 있다. 이것에 대해서는 자동 Thread Pool 관리 설정(Thread 상태 통보) 을 참고한다.

• Output Buffer Size

- 애플리케이션이 사용하는 응답 버퍼 크기값이다. 버퍼가 가득차면 해당 버퍼의 내용을 웹 엔진이 자동적으로 플러시한다. (단위: Byte)
- AJP 리스너의 경우 이 값을 mod_jk의 workers.properties 파일의 max_packet_size와 일치시키는 것이 바람직하다. mod_jk의 설정에 대해서는 [부하 분산을 위한 웹 서버 설정](#)을 참고한다.

• Postdata Read Timeout

- 요청 바디를 읽어 들일 때 기다릴 수 있는 최대시간을 설정한다. ServletInputStream.read() 메소드에서 적용한다. (단위: ms)
- HTTP 리스너, TCP 리스너에서는 타임아웃 적용 시점이 변경되었다.

각 리스너마다 존재하는 I/O 컨트롤 스레드에서 요청 바디를 모두 읽기 때문에 실제 서버릿에게 넘기기 전에 이미 타임아웃 여부가 결정된다. 요청 바디를 읽는 도중에 타임아웃이 발생하면 500 에러를 내보낸다. 단, HTTP 요청이 Chunked 타입인 경우에는 기존과 마찬가지로 서버릿이 ServletInputStream.read()를 호출할 때 적용된다.

• Max Post Size

- 너무 큰 크기의 POST 요청의 데이터가 들어와 이 데이터를 읽고 분석하고 처리하는 데 많은 부하가 걸려 다른 요청들의 처리에 장애가 생길 경우를 위한 설정이다. 이 설정을 통해 특정 크기 이상의 요청에 대한 처리 부담을 줄일 수 있다. (기본값: -1)
- POST 요청의 경우 요청의 Content-type에 따라 데이터의 최대 크기를 Byte 단위로 제한한다.
 - Content-Type이 x-www-form-urlencoded일 경우

요청에 따라오는 데이터의 Byte 크기가 설정된 값을 초과할 경우, chunked일 경우, 들어온 데이터의 크기의 합이 설정된 값보다 클 경우에 JEUS는 해당 요청은 처리하지 않고 "413 Request entity too large" 응답을 보내고 해당 요청 처리를 완료한다.

- Content-Type이 multipart/form-data일 경우

이 설정으로 업로드 파일의 크기와 POST 요청의 전체 크기를 제한할 수 있다. 만약 업로드 파일을 포함한 크기를 제한하려면 Servlet의 multipart 관련 설정을 이용할 것을 권장한다.

- 설정의 적용 여부는 Servlet 웹 애플리케이션 DD의 설정에 영향을 받는다. 즉, 설정에 따라 다음과 같이 동작하게 된다.

- web.xml에 <multipart-config>의 설정 여부에 따른 동작

web.xml에 <multipart-config> 설정이 존재할 경우 이 설정의 <max-file-size>와 <max-request-size>가 적용된다. <multipart-config> 설정이 없을 경우에는 이에 대한 제한을 웹 엔진에서는 제공하지 않는다. 따라서 각 애플리케이션에서 설정해서 사용하길 권장한다.

- <content-length>의 설정 여부에 따른 동작

<content-length>가 설정되어 있다면 이 값이 설정값을 초과할 경우 요청 처리가 제한이 된다. <content-length>가 설정되어 있지 않은 요청의 경우에는 이름/값 쌍의 파라미터의 Byte 합이 설정값을 초과할 때 요청 처리가 제한이 된다.

- 음수 설정한 경우에는 기본값으로 설정한 것과 같이 동작하며 데이터 크기의 제한이 없다.
- 해당 설정은 HTTP 요청을 받는 리스너 및 커넥터의 경우만 의미가 있다(TCP 리스너와 Tmax 커넥터의 경우 의미가 없다).

- **Max Parameter Count**

- 하나의 요청에 너무 많은 이름/값 쌍(파라미터)이 포함되어, 이 파라미터들을 분석 및 관리하는 부담이 증가하는 것을 방지하고자 할 경우 설정한다. (기본값: -1)
- GET과 POST 요청에 포함된 '이름/값' 쌍 즉, 파라미터의 총 개수를 설정값으로 제한한다.
- GET 요청의 Query String, POST 요청의 데이터나 multipart/form에 이름/값 쌍으로 들어온 모든 파라미터의 개수가 설정값 이상이 넘을 경우 "413 request entity too large" 응답을 보내고, 해당 요청의 처리를 중단한다.
- 음수로 설정한 경우에는 기본값으로 설정한 것과 같이 동작하며 파라미터 개수의 제한이 없다.
- 이 설정은 HTTP 요청을 받는 리스너 및 커넥터의 경우만 의미가 있다(TCP 리스너와 Tmax 커넥터의 경우 의미가 없다).

- **Max Header Count**

- 하나의 요청에 포함된 헤더가 무수히 많을 경우 이 요청을 읽는 부하가 많이 걸린다. 따라서 이럴 경우 해당 요청을 거부하여 서버의 부하를 줄일 수 있다. (기본값: -1)
- 설정된 값 이상의 헤더 개수가 포함된 요청은 처리하지 않고, "400 Bad request" 응답을 보낸 후 해당 연결을 끊는다. 즉, 헤더 이후의 데이터는 읽지 않고 버림으로써 서버의 부담을 줄이게 된다.
- 음수로 설정한 경우에는 기본값으로 설정한 것과 같이 동작하며, 헤더 개수의 제한이 없다.
- 이 설정은 HTTP 요청을 받는 리스너 및 커넥터의 경우만 의미가 있다(TCP 리스너와 Tmax 커넥터의 경우 의미가 없다).

- **Max Header Size**

- 하나의 요청에 포함된 헤더가 제한 없이 클 경우 이 헤더를 읽고 처리하는 데 많은 부하가 걸린다. 따라서 이럴 경우 해당 요청을 거부하여 서버의 부하를 줄일 수 있다. (기본값: -1)
- 설정값보다 Byte 크기(이때 헤더의 Byte는 요청의 헤더 하나를 타나내는 헤더 이름, 구분자, 헤더 값을 모두 포함한 크기)가 큰 헤더가 포함된 요청은 처리하지 않고 "400 Bad request" 응답을 보낸 후 해당 연결을 끊는다. 즉, 헤더 이후의 데이터는 읽지 않고 버림으로써 서버의 부담을 줄이게 된다.
- 음수로 설정한 경우에는 기본값으로 설정한 것과 같이 동작하며, 헤더 Byte의 제한이 없다.
- 해당 설정은 HTTP 요청을 받는 리스너 및 커넥터의 경우만 의미가 있다(TCP 리스너와 Tmax 커넥터의 경우 의미가 없다).

- **Max Query String Size**

- GET 요청의 Query String이 길 경우 이를 분석 및 관리하는 부하가 발생할 수 있다. 이런 경우 Query String의 크기를 제한하여 부하를 줄일 수 있다. (기본값: 8192, 단위: Byte)
- 설정된 Byte 이상으로 큰 Query String을 포함한 요청이 들어올 경우 해당 요청에 대해 "400 Bad Request" 응답을 보낸 후 해당 요청의 나머지 데이터들은 읽지 않고 버림으로써 서버의 부하를 줄인다.
- 음수로 설정한 경우는 Query String의 크기에 제한을 두지 않는다. 그러나 JEUS는 내부적으로 request line(요청의 첫 줄)의 크기를 64KB로 제한하므로 그 이상은 의미가 없다.
- 해당 설정은 HTTP 요청을 받는 리스너 및 커넥터의 경우만 의미가 있다(TCP 리스너와 Tmax 커넥터의 경우 의미가 없다).

2.3.2. AJP 리스너 설정

콘솔 툴을 사용하여 AJP 리스너를 추가하거나 수정 및 삭제할 수 있다. AJP 리스너는 AJP 버전 1.3을 준수한다.

콘솔 툴 사용

다음은 콘솔 툴을 사용해서 AJP 리스너를 추가, 수정, 삭제하는 방법이다.

- **추가**

콘솔 툴을 사용하여 AJP 리스너를 추가하려면 **add-ajp-listener** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "add-ajp-listener"를 참고한다.

```
add-ajp-listener [-cluster <cluster-name> | -server <server-name>]
                  [-f, --forceLock]
                  -name <web-connection-name>
                  -tmin <minimum-thread-num>
                  [-tmax <maximum-thread-num>]
                  [-tidle <max-idle-time>]
                  [-qs <max-queue-size>]
                  -slref <server-listener-ref-name>
```

- **수정**

콘솔 툴을 사용하여 AJP 리스너를 수정하려면 **modify-web-listener** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "modify-web-listener"를 참고한다.

```
modify-web-listener [-cluster <cluster-name> | -server <server-name>]
                    [-f, --forceLock]
                    -name <web-connection-name>
                    [-tmin <minimum-thread-num>]
                    [-tmax <maximum-thread-num>]
                    [-tidle <max-idle-time>]
                    [-http2 <enable-http2>]
                    [-obuf <output-buffer-size>]
```

• 삭제

콘솔 툴을 사용하여 AJP 리스너를 삭제하려면 **remove-web-listener** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "remove-web-listener"를 참고한다.

```
remove-web-listener [-cluster <cluster-name> | -server <server-name>]
                    [-f, --forceLock]
                    <web-connection-name>
```

2.3.3. HTTP 리스너 설정

콘솔 툴을 사용하여 HTTP 리스너를 추가하거나 수정 및 삭제할 수 있다. HTTP 리스너는 내부 관리 용도로만 사용할 것을 권장한다.



JEUS 9는 HTTP/2를 지원한다. HTTP/2 사용 방법에 대해서는 [HTTP/2 사용](#)을 참조한다.

콘솔 툴 사용

콘솔 툴을 사용하여 HTTP 리스너를 추가, 수정, 삭제하는 방법은 AJP 리스너의 방법과 동일하나 HTTP 리스너를 추가, 수정할 경우에 대해서만 HTTP/2를 사용할지 결정할 수 있는 옵션을 제공한다. 자세한 내용은 [AJP 리스너 설정](#)의 "콘솔 툴 사용"을 참고한다.

2.3.4. TCP 리스너 설정

TCP 리스너는 커스텀 프로토콜로 상호 간에 통신할 수 있도록 제공하는 특수한 리스너이다. TCP 리스너는 기본적으로 서버 리스너를 다른 웹 리스너들과 공유할 수 없기 때문에 사용을 위해서는 서버에 TCP 리스너를 위한 전용 리스너를 추가해야 한다. 콘솔 툴을 사용하여 TCP 리스너를 추가하거나 수정 및 삭제할 수 있다.

콘솔 툴 사용

콘솔 툴을 사용하여 TCP 리스너를 추가, 수정, 삭제하는 방법은 AJP 리스너의 방법과 동일하다. 자세한 내용은 [AJP 리스너 설정](#)의 "콘솔 툴 사용"을 참고한다.

2.3.5. WebtoB 커넥터 설정

WebtoB 커넥터는 커넥션을 생성할 때 JEUS가 클라이언트 역할을 하게 된다. 따라서 WebtoB 주소와 연결 포트가 필요하다. 콘솔 툴을 사용하여 WebtoB 커넥터를 추가하거나 수정 및 삭제할 수 있다.



WebtoB와 JEUS 사이의 통신 프로토콜인 WJP 버전이 1에서 2(이하 WJPv1 및 WJPv2)로 업그레이드되었다. WJPv2의 경우 WJPv1에 비교해서 좀더 적은 사이즈의 패킷을 사용하며, 여러 가지 부가 기능들을 제공한다. JEUS에서 연결할 WebtoB가 WJPv2를 지원하지 않는 버전일 경우에는 WJPv1을 사용하면 된다. WJP 버전 정보의 경우 WebtoB에서 자동으로 파악할 수 없기 때문에 JEUS에 <wjp-version>으로 설정해야 한다. WJP는 WebtoB-JEUS Protocol을 의미한다.

콘솔 툴 사용

다음은 콘솔 툴을 사용해서 WebtoB 커넥터를 추가, 수정, 삭제하는 방법이다.

• 추가

콘솔 툴을 사용하여 WebtoB 커넥터를 추가하려면 **add-webtob-connector** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "add-webtob-connector"를 참고한다.

```
add-webtob-connector [-cluster <cluster-name> | -server <server-name>]
                    [-f, --forceLock]
                    -name <web-connection-name>
                    -tmin <minimum-thread-num>
                    [-tmax <maximum-thread-num>]
                    [-tidle <max-idle-time>]
                    [-qs <max-queue-size>]
                    -conn <thread-number>
                    -regid <registration-id>
                    [-ver <wjp-version>]
                    [-useNio <use-nio>]
                    [-addr <WebtoB address>]
                    [-dsocket | -port <WebtoB port>]
                    [-wbhome <webtob-home>]
                    [-ipcport <ipc-base-port>]
                    [-sndbuf <send-buffer-size>]
                    [-rcvbuf <receive-buffer-size>]
                    [-hth <set-hth-count.>]
```

- 수정

콘솔 툴을 사용하여 WebtoB 커넥터를 수정하려면 **modify-webtob-connector** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "modify-webtob-connector"를 참고한다.

```
modify-webtob-connector [-cluster <cluster-name> | -server  
                        <server-name>]  
                        [-f,--forceLock]  
                        -name <web-connection-name>  
                        [-tmin <minimum-thread-num>]  
                        [-tmax <maximum-thread-num>]  
                        [-tidle <max-idle-time>]  
                        [-conn <thread-number>]  
                        [-obuf <output-buffer-size>]  
                        [-ver <wjp-version>]  
                        [-addr <WebtoB address>]  
                        [-dsocket | -port <WebtoB port>]  
                        [-wbhome <webtob-home>]  
                        [-cloud]  
                        [-ipcport <ipc-base-port>]  
                        [-regid <registration-id>]  
                        [-sndbuf <send-buffer-size>]  
                        [-rcvbuf <receive-buffer-size>]  
                        [-hth <set-hth-count.>]
```

- 삭제

콘솔 툴을 사용하여 WebtoB 커넥터를 삭제하려면 **remove-webtob-connector** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "remove-webtob-connector"를 참고한다.

```
remove-webtob-connector [-cluster <cluster-name> | -server  
                        <server-name>]  
                        [-f,--forceLock]  
                        <web-connection-name>
```

2.3.6. Tmax 커넥터 설정

Tmax 커넥터 역시 WebtoB 커넥터와 마찬가지로 커넥션을 생성할 때 JEUS가 클라이언트 역할을 하게 된다. 따라서 Tmax 주소와 연결 포트가 필요하다. 콘솔 툴을 사용하여 Tmax 커넥터를 추가하거나 설정 및 삭제할 수 있다.

콘솔 툴 사용

다음은 콘솔 툴을 사용해서 Tmax 커넥터를 추가, 수정, 삭제하는 방법이다.

- 추가

콘솔 툴을 사용하여 Tmax 커넥터를 추가하려면 **add-tmax-connector** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "add-tmax-connector"를 참고한다.

```
add-tmax-connector [-cluster <cluster-name> | -server <server-name>]
                    [-f,--forceLock]
                    -name <web-connection-name>
                    -tmin <minimum-thread-num>
                    [-tmax <maximum-thread-num>]
                    [-tidle <max-idle-time>]
                    [-qs <max-queue-size>]
                    -addr <server-address>
                    -port <server-port>
                    -svrg <server-group-name>
                    -svr <server-name>
                    -dcc <dispatcher-config-class>
```

• 수정

콘솔 툴을 사용하여 Tmax 커넥터를 수정하려면 **modify-tmax-connector** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "modify-tmax-connector"를 참고한다.

```
modify-tmax-connector [-cluster <cluster-name> | -server <server-name>]
                       [-f,--forceLock]
                       -name <web-connection-name>
                       -tmin <minimum-thread-num>
                       [-tmax <maximum-thread-num>]
                       [-tidle <max-idle-time>]
                       [-qs <max-queue-size>]
                       -addr <server-address>
                       -port <server-port>
                       -svrg <server-group-name>
                       -svr <server-name>
                       -dcc <dispatcher-config-class>
```

• 삭제

콘솔 툴을 사용하여 Tmax 커넥터를 삭제하려면 **remove-tmax-connector** 명령어를 실행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "remove-tmax-connector"를 참고한다.

```
remove-tmax-connector [-cluster <cluster-name> | -server <server-name> |
                       -f,--forceLock]
                       <web-connection-name>
```


2.4. 부하 분산을 위한 웹 서버 설정

본 절에서는 JEUS와의 연동 사례가 많은 웹 서버들의 설정 방법과 웹 엔진을 웹 커넥션과 연동할 수 있는 방법에 대해서 설명한다.

웹 엔진은 시스템의 HTTP 처리의 성능 향상을 위해 웹 서버와 웹 엔진(서블릿 엔진)으로 구성되어 서비스할 수 있다. 각 웹 엔진으로의 HTTP 요청을 웹 엔진 앞의 웹 서버에서 1차로 부하를 분산시킨다. 이후 같은 연결이나 세션의 요청에 대해 HTTP 세션 클러스터링 서비스를 이용하여 처음 요청을 처리한 웹 엔진으로 할당하여 서비스 처리 효율을 높인다.

처음 요청을 처리한 웹 엔진에 장애가 발생한 경우, 장애 발생 이후에 들어온 요청은 웹 엔진 앞의 웹 서버에서 장애가 발생하지 않은 웹 엔진으로 전달하여 끊김 없는 서비스를 가능하게 한다. 이 서비스는 HTTP 세션 클러스터링을 사용한다는 전제 조건이 필요하다. HTTP 세션 클러스터링에 대한 자세한 내용은 JEUS 세션 관리 안내서의 "세션 서버"를 참고한다.

만약 웹 서버를 사용하지 않고 웹 엔진만을 사용할 경우에 요청을 처리한 웹 엔진으로 이후의 요청을 전달할 수 있는 장비나 소프트웨어를 사용하면 웹 서버를 사용할 때와 같이 효율적인 서비스가 가능할 것이다. 그러나 JEUS에서는 이에 필요한 소프트웨어를 제공하지 않고, 이런 사용을 권장하지 않는다.

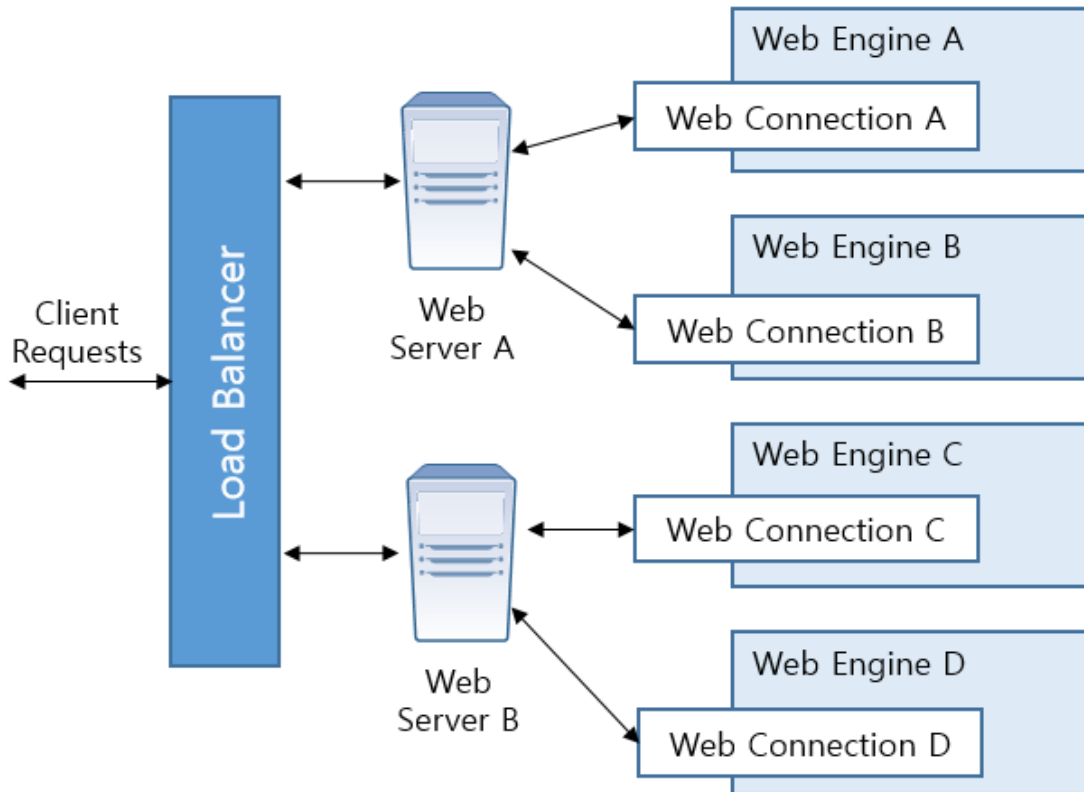


각 웹 서버들의 설정과 mod_jk 설정은 웹 서버 버전에 따라서 차이가 있을 수 있다. 본 절의 설정 방법은 JEUS 사용자들의 이해를 돕기 위해서 제공하는 것이므로 **실제로 설정할 때에는 반드시 각 웹 서버들의 문서, 국내외 커뮤니티에서 제공하는 설정 사례들을 참고해야 한다.**

2.4.1. 부하 분산 구조

서비스 요청이 많은 웹 사이트는 한 대의 웹 서버와 웹 엔진으로는 서비스를 제공하기 어렵기 때문에 부하 분산을 위해서 여러 개의 웹 서버와 웹 엔진이 필요하다.

다음은 2대의 웹 서버와 4대의 웹 엔진이 연결되어 있는 부하 분산 구조를 나타낸다.



소규모 웹 사이트에서의 부하 분산 구조

각 웹 엔진에는 동일한 웹 컨텍스트들이 deploy되어 있어야 한다. 이러한 환경에서 중요한 것은 세션의 공유 여부이다. 애플리케이션을 좀 더 편리하게 관리하고 싶거나 분산 세션 서비스가 필요하다면 웹 엔진들을 하나의 클러스터로 묶어야 한다. 세션 클러스터에 대한 자세한 내용은 JEUS 세션 관리 안내서의 "세션 트래킹"을 참고한다.

2.4.2. Apache 웹 서버

Apache를 웹 엔진과 연동하려면 다음과 같은 과정을 수행해야 한다.

1. Apache에 mod_jk 라이브러리를 설치한다.
2. JEUS 웹 엔진에 AJP13 리스너를 설정한다.
3. workers.properties 파일을 생성하고 편집한다.
4. Apache 웹 서버의 httpd.conf 파일에 연동 구절을 삽입한다.
5. Apache 웹 서버를 재시작한다.



Apache의 경우 2.2.4, mod_jk의 경우 1.2.20을 기준으로 설명한다.

mod_jk 라이브러리 설치

Apache 웹 서버를 웹 엔진의 앞 단에서 사용하려면 "mod_jk"라는 모듈을 Apache 설치 모듈에 추가해야 한다. "mod_jk" 모듈은 서버와 엔진 간의 통신 프로토콜을 구현한 것으로, 이 프로토콜은 Apache JServ Protocol 1.3(AJP 1.3)이다.



현재 Windows 바이너리를 제공하고 있는 [Apache Tomcat 커넥터 다운로드 페이지](#)에서 소스를 다운로드받아서 해당 머신에서 컴파일한다.

AJP13 리스너 설정

mod_jk 라이브러리 설치가 완료되면 JEUS 웹 엔진에 AJP13 리스너를 설정한다. AJP13 리스너 설정 방법의 자세한 내용은 [AJP 리스너 설정](#)을 참고한다.

workers.properties 설정

workers.properties 파일은 mod_jk를 위한 설정 파일이다. 예를 들어 JEUS는 server1, server2라는 2대의 서버가 있다. 호스트 이름이 각각 server1, server2라고 가정한다. 각 웹 엔진에 설정된 AJP 리스너들은 9901, 9902로 설정했다고 가정하자.

다음은 가정한 JEUS 환경에서의 workers.properties 예제이다.

mod_jk 설정 파일 예제 : <workers.properties>

```
worker.list=jeus_load_balancer_workers
worker.jeus_load_balancer_workers.type=lb
worker.jeus_load_balancer_workers.sticky_session=true

#####
# listener specific configuration
#####
worker.jeus_load_balancer_workers.balance_workers=server1
worker.server1.reference=worker.template
worker.server1.host=192.168.0.101
worker.server1.port=9901
worker.server1.route=ZG9tYWluMS9zZXJ2MQ==

worker.jeus_load_balancer_workers.balance_workers=server2
worker.server1.reference=worker.template
worker.server1.host=192.168.0.102
worker.server1.port=9902
worker.server1.route=ZG9tYWluMS9zZXJ2Mg==

#####
# common config
#####
worker.template.type=ajp13
worker.template.socket_connect_timeout=5000
worker.template.socket_keepalive=true
worker.template.ping_mode=A
worker.template.ping_timeout=10000
worker.template.connection_pool_minsize=0
worker.template.connection_pool_timeout=600
worker.template.reply_timeout=300000
worker.template.recovery_options=3
```

workers.properties 파일은 몇몇 정의를 제외하고 일반적으로 "worker.<worker_name>.<directive>=<value>

>"의 형식으로 정의한다.

다음은 중요한 설정에 대한 설명이다.

- **worker.list**

해당 worker를 정의하는 항목이다. worker의 이름들은 콤마(,)로 구분되며 여러 worker들을 나열할 수 있다.

다음은 "jeus_load_balancer_workers"라는 이름의 worker를 하나 정의한 예제이다.

```
worker.list=jeus_load_balancer_workers
```

- **worker.<worker_name>.type**

worker의 타입을 정의한다. 'ajp13', 'ajp14', 'jni', 'lb', 'status' 등을 선택할 수 있다. AJP13을 지원하기 위해 JEUS에서는 'ajp13', 'lb'만의 설정으로도 충분하다.

다음은 "jeus_load_balancer_workers"를 Load Balancer type으로 정의한 예제이다.

```
worker.jeus_load_balancer_workers.type=lb
```

- **worker.<worker_name>.balance_workers**

콤마(,)로 구분하여 Load Balance를 원하는 worker들을 나열할 수 있다. 위의 woker.list에 적은 worker들이 나타나서는 안 된다.

JEUS의 AJP13 리스너와 연동하는 경우 올바른 Load Balancer 및 sticky session 역할을 하려면 worker의 이름을 JEUS의 서블릿 엔진 이름으로 해야 한다. JEUS의 경우 Session ID의 라우팅 정보로 서블릿 엔진 이름을 이용하며 mod_jk에서는 worker의 이름을 이용하기 때문이다.

- **worker.<worker_name>.sticky_session**

Session ID를 기반으로 라우팅을 지원할지 여부를 설정한다. true(or 1) 또는 false(or 0)로 설정할 수 있다. JEUS에서는 Sticky Session을 기본으로 지원하므로 항상 true로 설정한다. (기본값: true)

다음은 "jeus_load_balancer_workers"는 Sticky Session을 지원한다는 예제이다.

```
worker.jeus_load_balancer_workers.sticky_session=true
```

- **worker.<worker_name>.host**

JEUS의 AJP13 리스너가 존재하는 호스트 이름 또는 IP 주소를 입력한다.

- **worker.<worker_name>.port**

JEUS의 AJP13 리스너의 포트 번호를 설정한다.

- **worker.<worker_name>.reference**

많은 Load Balancer worker들을 설정할 때 유용하며 지정된 값에 해당하는 worker의 설정값을 reference할 수 있는 설정이다. 예를 들어 "worker.castor.reference=worker.template"라고 설정했다면 명시적으로 castor worker에서 설정한 값을 제외하고 모든 worker.template의 설정들을 상속받는다.

- **worker.<worker_name>.route**

설정값에 해당하는 값이 sticky로 요청이 왔을 경우에 해당 worker가 처리하는 설정이다. JEUS의 세션 매니저에서 생성하는 세션에 붙이는 세션 라우팅 기술로 해당하는 worker를 매치시킴으로서 로컬 세션의 활용도를 높일 수 있다. 세션 라우팅에 대한 자세한 내용은 JEUS 세션 관리 안내서의 "세션 트래킹 구조"를 참고한다.

해당 값은 domain_name/server_name의 Base64 알고리즘으로 인코딩한 값이다. 해당 인코딩 값은 JEUS_HOME/bin의 encryption 명령을 통해 얻을 수 있다.

다음은 사용 예제이다.

```
${JEUS_HOME}/bin/encryption -algorithm base64 -text domain1/server1
```

설정할 때 "domain1/server1", "domain1/server2"와 같이 "도메인 이름/서버 이름"을 사용하는 것에 유의한다.



이 설정들은 mod_jk 버전에 따라 deprecated될 수도 있으므로 본 절에서 제시한 내용은 참고 용도로만 사용한다. 반드시 Tomcat 홈 페이지의 mod_jk 관련 설명에 따라 설정한다.

httpd.conf 설정

httpd.conf 파일에는 mod_jk 모듈을 사용하기 위해 다음 사항을 추가해야 한다.

Apache에 mod_jk 설정 예제 : <httpd.conf>

```
. . .
LoadModule      jk_module  "/usr/local/apache/modules/mod_jk.so"
JkWorkersFile   "/usr/local/apache/conf/workers.properties"
JkLogFile       "/usr/local/apache/logs/mod_jk.log"
JkLogLevel      info
JkMount  /examples/*  jeus_load_balancer_workers
. . .
```



Apache 버전에 따라 deprecated될 수도 있으므로 여기에서 제시한 것은 참고 용도로만 사용한다. 반드시 Apache 매뉴얼에 따라 설정한다.

2.4.3. IIS 웹 서버 및 Iplanet 웹 서버 설정

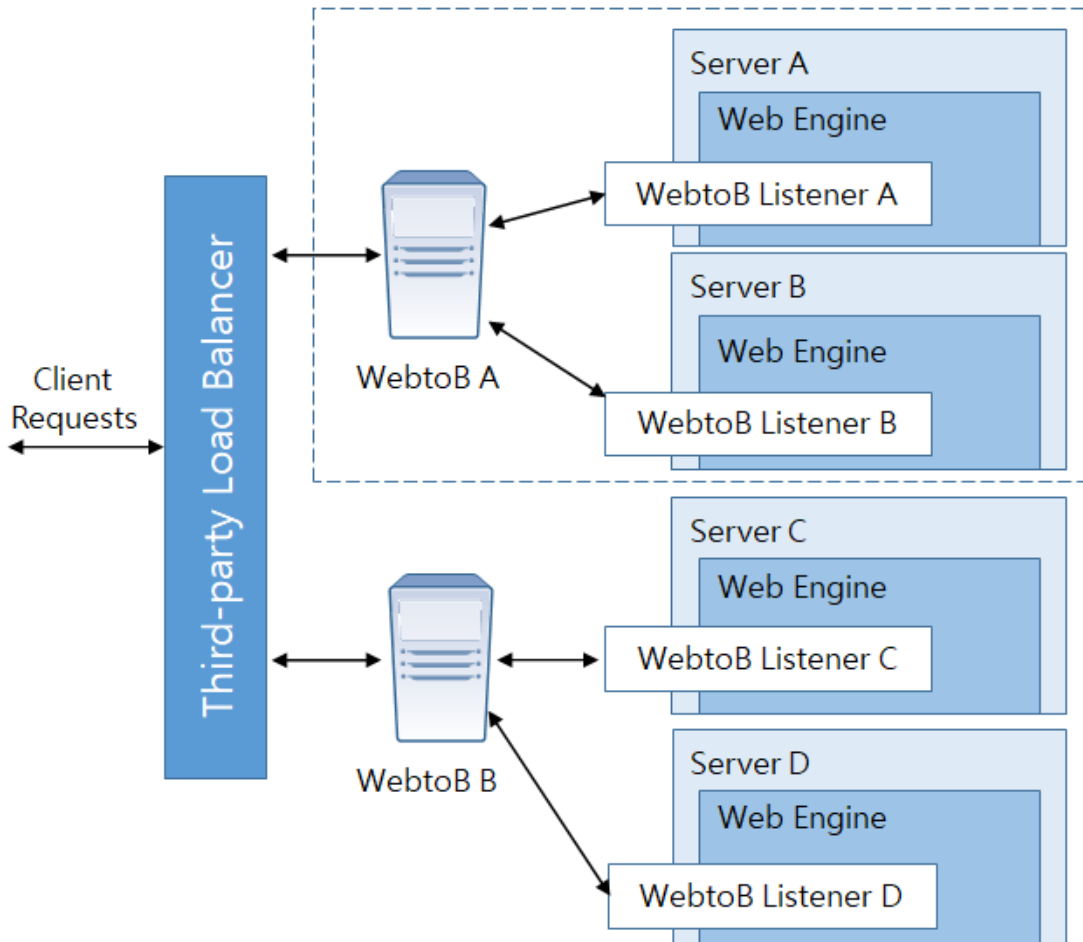
IIS 웹 서버와 Iplanet 웹 서버도 AJP13 프로토콜을 지원한다. 따라서 workers.properties와 JEUS 설정 방식은 모두 동일하다. 단, 각 제품에 따라 mod_jk 모듈을 설치하는 방법과 mod_jk 사용 설정을 하는 방법이 다르다.

이러한 설정 방법은 각 웹 서버에서 제공하는 매뉴얼을 참고한다.

2.4.4. WebtoB 부하 분산 설정

본 절에서는 예제를 통하여 WebtoB와 JEUS의 부하 분산 처리 환경을 구성한다.

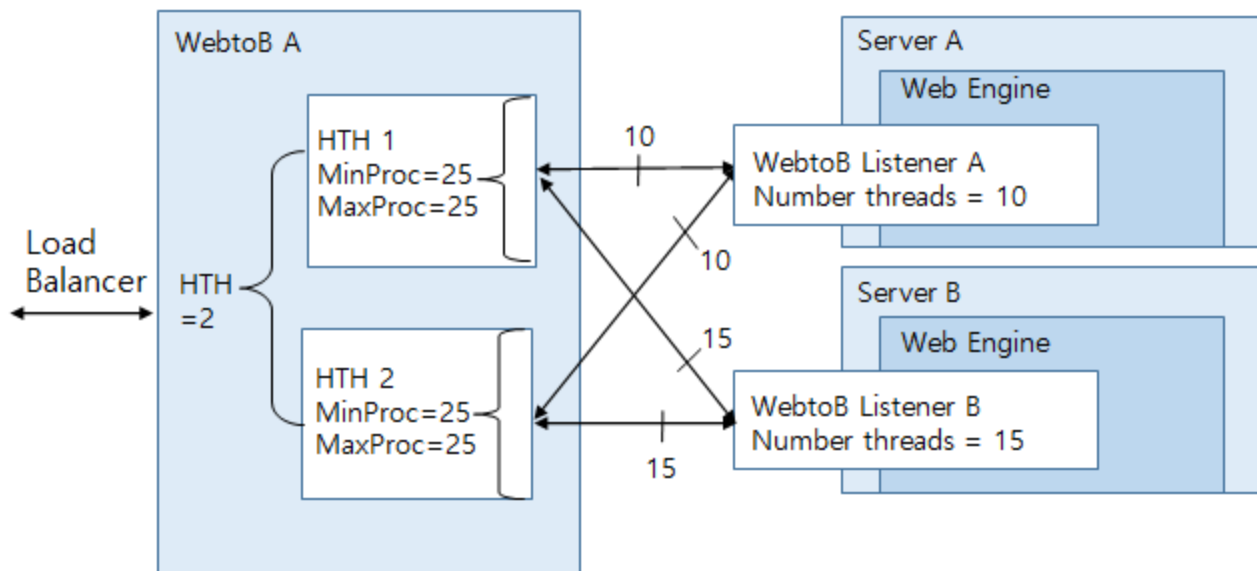
다음은 예제의 서버 구조를 나타낸다. 각 엔진이 각각의 WebtoB 서버를 갖는 구조에 대한 설정으로 예제는 2대의 WebtoB가 각각 2대의 웹 엔진에 연결된다.



부하 분산 서버 구조 - 2대의 WebtoB가 각각 2대의 웹 엔진에 연결된 경우

Server A부터 D에는 동일한 웹 컨텍스트를 deploy해야 한다. 위와 같은 구성에서는 일반적으로 Server A부터 D가 하나의 클러스터를 이루게 되며, 그에 따라 분산식 세션 서버를 사용하게 된다. 만약 사용자 세션이 필요 없는 경우에는 클러스터를 구성하지 않아도 무방하다.

다음은 WebtoB A와 Server A, Server B가 연결된 모습을 좀 더 자세히 표현한 것으로 각각의 WebtoB 커넥터 설정 상태를 보여준다.



WebtoB 커넥터 설정 상태

설정할 때는 WebtoB의 HTH 프로세스 수를 고려해야 한다.

WebtoB HTH 프로세스는 몇 개의 하위 프로세스를 가지고 있는 엔진처럼 행동한다. 이는 WebtoB 커넥터의 Worker Thread와 1대 1로 연결된다. 따라서 WebtoB 커넥터의 '**connection-count**' 설정값과 WebtoB 설정의 '**MaxProc**' 값을 주의해서 설정해야 한다.

HTH 프로세스의 개수는 WebtoB 설정에 존재하는 HTH 프로세스 개수를 그대로 WebtoB 커넥터의 '**Hth Count**' 설정에 반영한다. WebtoB의 '**MaxProc**' 값과 WebtoB 커넥터의 '**connection-count**' 값은 다음과 같은 공식으로 표현될 수 있다.

MinProc = Listener A connection-counts + Listener B connection-counts + ... + Listener X connection-counts setting.
MaxProc = Listener A connection-counts + Listener B connection-counts + ... + Listener X connection-counts setting.

WebtoB 커넥터의 '**Hth Count**' 값은 WebtoB 설정 파일의 NODE 절의 요소 중 HTH 프로세스 개수와 반드시 동일해야 한다. WebtoB 커넥터 설정은 [WebtoB 커넥터 설정](#)을 참고한다.

다음은 WebtoB A가 위의 예제와 같이 설정되기 위한 http.m 파일의 설정 예이다.

WebtoB 설정 예 : <http.m>

```

*NODE
foo    ...
    HTH = 2,
    JSVPORT = 9900,
    ...
*SVRGROUP
jsvg    NODENAME = foo, SVRTYPE = JSV

*SERVER
default    SVGNAME = jsvgl, MinProc = 6, MaxProc = 25

*URI
uri1    Uri = "/examples/", Svrtype = JSV

```

```
uri2 Uri = "/test/", Svrtype = JSV

*EXT
jsp MimeType = "application/jsp", SvrType = JSV
```

connection count는 6과 25 사이여야 한다.

JEUS는 WebtoB와의 Connection이 끊기면 재연결을 시도하게 된다. Application Layer에 있는 JEUS는 TCP Close HandShake가 언제 완료되는지 알 수 없기 때문에, JEUS는 일정 시간을 기다린 후에 재연결을 해야 한다. 만약, http.m의 MinProc, MaxProc과 webtob-connector의 connection-count와 같을 경우, JEUS가 TCP Close HandShake가 끝나기 전에 재연결을 요청하면 새로운 연결이 MaxProc보다 많아지기 때문에 WebtoB는 재연결을 받아주지 않는다. 이럴 경우, TCP Close HandShake가 끝나서 WebtoB가 재연결을 받아줄 때까지 JEUS는 reconnect-interval 주기마다 재연결을 시도한다. JEUS는 default로 150ms를 대기한 후에 재연결을 시도하고, jeus.servlet.wjpReconnectTime 시스템 프로퍼티를 jvm option으로 설정할 수 있다.

2.5. TCP 리스너 사용

TCP 리스너는 통신 프로토콜을 직접 정의해서 TCP 클라이언트와 TCPServlet 간에 통신할 수 있다. 단, TCP 리스너는 HTTP 또는 HTTPS 프로토콜로 만족할 수 없는 사항이 있을 경우에만 사용하기를 권장한다.

TCP 리스너를 사용하기 위해서는 다음과 같은 과정을 수행해야 한다.

1. 맞춤 통신 프로토콜을 정의한다.
2. Dispatcher Config Class를 구현한다(프로토콜 설정 정보).
3. TCP 핸들러 서블릿을 구현한다(프로토콜 구현).
4. 맞춤 프로토콜 구현을 위한 TCP 리스너를 설정한다.
5. TCP 클라이언트를 구현한다.
6. TCP 클라이언트를 컴파일하고 구동한다.

본 절에서 사용할 용어는 다음과 같다.

용어	설명
프로토콜(통신 프로토콜)	TCP 클라이언트와 TCP 핸들러 사이에서 (TCP 리스너가 중간역할) 교환되는 메시지의 구조와 내용을 정의한다.
메시지	TCP 클라이언트와 TCP 핸들러 사이에서 교환되는 데이터이다. 프로토콜에서 정의한 구조에 맞아야 한다.
TCP 클라이언트	TCP 리스너와 교신하며 메시지를 주고받는 외부의 애플리케이션(TCP 핸들러에 의해 처리된다)이다. 메시지를 주고받는 것은 표준 소켓을 사용한다.
TCP 핸들러	TCP 리스너를 통해 메시지를 받고 구현된 방법대로 메시지를 처리한다. TCP 핸들러는 jeus.servlet.tcp.TCPServlet의 하위 클래스의 형태로 구현되고 웹 엔진에 보통 서블릿처럼 등록된다. TCP 핸들러는 TCP 프로토콜 위에 존재하는 맞춤 프로토콜의 "제공자" 또는 "구현체"처럼 동작한다고 할 수 있다.

용어	설명
TCP Dispatcher Configuration Class	<p>TCP Dispatcher Configuration Class는 jeus.servlet.tcp.TCPDispatcherConfig를 확장한 클래스이다. 이 클래스는 맞춤 프로토콜에 대한 정보를 TCP 리스너로 전달하여 이 non-HTTP 메시지를 해석하여 처리한다.</p> <p>이 클래스는 DOMAIN_HOME/lib/application 디렉터리 아래에 위치시키고, 웹 엔진 설정 하위의 TCP 리스너 설정 항목 중 'Dispatcher Config Class' 항목에 설정한다(TCP 리스너 설정 참조).</p>
TCP 리스너	<p>맞춤 메시지에 대한 해석과 라우팅 인프라를 제공한다. TCP 클라이언트와 TCP 핸들러 사이에서 non-HTTP 중간 매개체로 역할을 한다.</p> <p>이것은 다른 HTTP 리스너와 마찬가지로 웹 엔진 내에 존재한다.</p>

2.5.1. 맞춤 통신 프로토콜 정의

TCP 리스너는 TCP 클라이언트와 TCP 핸들러 간에 통신되는 모든 메시지들을 두 부분으로 나누는데, 헤더와 바디가 그것이다. 일반적으로 헤더는 기본적인면서도 표준 정보를 전달하고 크기도 정해져 있다. 바디는 전송될 임의의 사용자 데이터가 포함된다. (예: HTTP 응답의 HTML 코드)

TCP 리스너의 사용을 설명하기 위해 간단한 프로토콜(메시지 구조)을 정의한다(이것은 나중에 사용할 것이다).

맞춤 통신 프로토콜(맞춤 메시지 구조)은 다음과 같은 내용을 가진다.

- 헤더
 - 헤더는 4Bytes의 **magic** 수로 시작한다. 이는 사용되는 프로토콜을 식별한다. '7777'로 설정한다.
 - 1Byte의 **type** 필드가 magic 수 다음에 쓰인다. '0'은 요청을 '1'은 응답을 의미한다.
 - 세 번째 항목은 4Bytes의 **body length**이다. 이 항목은 메시지의 바디 부분의 Byte 수를 기록한다. 예제에서는 크기가 128Bytes로 고정되어 있다.
 - 마지막 항목은 32Bytes의 string으로 **service name**을 기록한다. 예제에서는 그 값으로 요청을 처리하는 TCPServlet의 이름을 입력한다.
- 바디
 - 메시지의 바디 부분은 128Bytes 길이를 가진 문자 데이터를 넣는다. 예제에서는 이 128Byte 블록을 2개의 임의의 64Bytes 텍스트 string으로 넣는다.

2.5.2. Dispatcher Config Class 구현

Dispatcher Config Class는 jeus.servlet.tcp.TCPDispatcherConfig Class의 하위 클래스이다. 이 클래스의 abstract 메소드는 TCP 리스너가 알맞은 TCP 핸들러에게 메시지를 전달하기 위해 필요한 여러 가지 정보들이 구현되어야 한다. 이 메소드들은 JEUS_HOME/docs/api/jeus-servlet/index.html에 설명되어 있다.

JEUS 7 Fix#1부터 getBodyLengthLong 메소드가 추가되었다. TCP 바디의 크기가 2GB 이상이라면 해당 메소드를 사용해서 바디 길이를 8Byte로 나타내면 된다.

다음은 정의된 프로토콜에 기반하여 Dispatcher Config Class를 어떻게 구현했는지를 보여준다. 특정 메소드들이

어떻게 사용되었는지는 주석을 참고한다.

TCPDispatcherConfig 구현 예 : <SampleConfig.java>

```
package sample.config;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jeus.servlet.tcp.*;

/*
 * This class extends the abstract class jeus.servlet.tcp.
 * TCPDispatcherConfig class. This implementation provides
 * routing and handling information to the TCP listener
 * according to our defined communications protocol.
 */
public class SampleConfig extends TCPDispatcherConfig {
    /*
     * Any init code goes into this method. This method will be
     * Called once after starting the TCP listener.
     * We leave this method empty for our simple example.
     */
    public void init() {}

    /*
     * This method returns the fixed-length of the header so
     * that the TCP listener knows when to stop
     * reading the header. The header length for
     * our example is 41 (bytes): 4 (magic) + 1
     * (type) + 4 (body length) + 32 (service name).
     */
    public int getHeaderLength() {
        return 41;
    }

    /*
     * This method must return the length of the body.
     * For our example, this length is represented as an
     * "int" in the header, starting at position "5"
     * (see the protocol definition above).
     */
    public int getBodyLength(byte[] header) {
        return getInt(header, 5);
    }

    /**
     * Returns the long-size length of request body of
     * incoming request packet.
     * If you don't need to support long, you may map to
     * {@link #getBodyLength(byte[])}.
     */
    public long getBodyLengthLong(byte[] header) {
        return getBodyLength(header);
    }

    /*
     * This method must return the context path so that the
     * request can be routed by the TCP listener to the context
     * that contains the TCP handler (TCPServlet implementation).
     */
}
```

For our example, we always use the context path “/tcptest”.

```
*/
public String getContextPath(byte[] header) {
    return "/tcptest";
}

/*
    This method must return the name (path) of the TCP
    handler(TCPServlet) relative to the context path.
    For our example, we fetch this name from the 9th position
    in the header.
*/
public String getServletPath(byte[] header) {
    return "/" + getString(header, 9, 32);
}

/*
    This method returns some path info from the header.
    It is not used in our example and thus returns “null”.
*/
public String getPathInfo(byte[] header) {
    return null;
}

/*
    This method returns any session ID embedded in the header.
    It is not used in our example and thus returns “null”.
*/
public String getSessionId(byte[] header) {
    return null;
}

/*
    This method determines whether the TCP listener
    should keep the socket connection open after the TCP handler
    has delivered its response. If it returns “false”, the
    connection will be dropped like in HTTP communications.
    If it returns “true” the connection will be kept open
    like in the Telnet or FTP protocols. For our example,
    we choose to make it persistent (connection not closed
    by the TCP listener).
*/
public boolean isPersistentConnection() {
    return true;
}
}
```

2.5.3. TCP 서블릿 구현

TCP 서블릿은 항상 `jeus.servlet.tcp.TCPServlet`의 하위 클래스이다. 여기에는 항상 overridden되는 `abstract void service(TCPServletRequest req, TCPServletResponse res)` 메소드가 존재한다.

`service` 메소드는 맞춤 프로토콜에 준하는 메시지를 처리하도록 구현해야 한다. 웹 컨테이너는 헤더를 읽어서 `TCPServletRequest` 객체에 전달하며, TCP 서블릿에서는 `TCPServletResponse` 객체의 `output stream`으로 응답을 쓴다.

다음 예제는 맞춤 프로토콜에 준하는 메시지를 처리하는 TCP 서블릿의 구현 코드이다.

TCP 서블릿 구현 예 : <SampleTCPServlet.java>

```
package sample.servlet;

import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jeus.servlet.tcp.*;

/**
 * Sample TCPServlet implementation
 *
 * Protocol scheme:
 *
 * common header (for request and response) : total 41 byte
 *
 * magic field: length = 4 byte, value = 7777
 * type field : length = 1 byte, 0 : request, 1:response
 * body length field : length = 4, value = 128
 * service name field : length = 32
 *
 * request and response body
 *
 * message1 field : length = 64
 * message2 field : length = 64
 */

public class SampleTCPServlet extends TCPServlet {

    public void init(ServletConfig config) throws ServletException {
    }

    public void service(TCPServletRequest req, TCPServletResponse res)
        throws ServletException, IOException {
        ServletContext context = req.getServletContext();
        byte[] header = req.getHeader();
        byte[] body = new byte[req.getContentLength()];
        int read = req.getInputStream().read(body);
        if (read < body.length) {
            throw new IOException("The client sent the wrong content.");
        }

        String encoding = res.getCharacterEncoding();
        if (encoding == null)
            encoding = "euc-kr";

        DataInputStream in = new DataInputStream(new ByteArrayInputStream(header));
        int magic = in.readInt();
        context.log("[SampleTCPServlet] received magic = " + magic);

        byte type = (byte)in.read();
        context.log("[SampleTCPServlet] received type = " + type);

        int len = in.readInt();
        context.log("[SampleTCPServlet] received body length = " + len);
    }
}
```

```

byte[] svcname = new byte[32];
in.readFully(svcname);
context.log("[SampleTCPServlet] received service name = "
            + (new String(svcname)).trim());

String rcvmsg = null;
rcvmsg = (new String(body, 0, 64)).trim();
context.log("[SampleTCPServlet] received msg1 = " + rcvmsg);

try {
    rcvmsg = (new String(body, 64, 64, encoding)).trim();
} catch (Exception e) {}
context.log("[SampleTCPServlet] received msg2 = " + rcvmsg);

String msg1 = "test response";
String msg2 = "test response2";

byte[] result1 = null;
byte[] result2 = null;
if (encoding != null) {
    try {
        result1 = msg1.getBytes(encoding);
        result2 = msg2.getBytes(encoding);
    } catch (UnsupportedEncodingException uee) {
        result1 = msg1.getBytes();
        result2 = msg2.getBytes();
    }
} else {
    result1 = msg1.getBytes();
    result2 = msg2.getBytes();
}

header[4] = (byte)1; // mark as response
ServletOutputStream out = res.getOutputStream();
out.write(header);

byte[] buf1 = new byte[64];
System.arraycopy(result1, 0, buf1, 0, result1.length);
out.write(buf1);

byte[] buf2 = new byte[64];
System.arraycopy(result2, 0, buf2, 0, result2.length);
out.write(buf2);

out.flush();
}

public void destroy() {
}
}

```



JEUS 7 Fix#2부터는 `TCPServletRequest.getBody()` 메소드 사용을 권장하지 않는다. 이 메소드는 요청 바디가 매우 큰 경우 메모리 문제를 일으킬 수 있다. 따라서 서블릿 표준에 정의된 `ServletInputStream`을 사용하는 것을 권장한다. 이는 `TCPServletRequest.getInputStream()`으로 얻을 수 있다.

2.5.4. 맞춤 프로토콜 코드를 위한 TCP 리스너 설정

구현한 코드(SampleConfig.java와 SampleTCPServlet.java)를 기반으로 TCP 리스너를 설정한다. 설정 과정은 다음과 같다.

1. 웹 엔진에 TCP 리스너 정보를 설정한다. TCP 리스너 정보 설정에 대한 자세한 내용은 [TCP 리스너 설정](#)을 참고한다. 단, TCP 리스너에서 참조하는 서버 리스너의 포트는 '5555', '**Dispatcher Config Class**'는 'sample.config.SampleConfig'를 입력해야 한다.
2. JEUS 서버를 시작하고 위에서 작성한 TCP 서블릿을 포함시킨 웹 애플리케이션을 deploy한다. 이때 웹 애플리케이션의 DD의 예는 다음과 같다.

TCP 핸들러 DD : <web.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
  metadata-complete="false"
  version="5.0">
  <display-name>test</display-name>
  <distributable/>
  <servlet>
    <servlet-name>SampleServlet</servlet-name>
    <servlet-class>sample.servlet.SampleTCPServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
    <async-supported>true</async-supported>
  </servlet>
  <servlet-mapping>
    <servlet-name>SampleServlet</servlet-name>
    <url-pattern>/sample</url-pattern>
  </servlet-mapping>
  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>
</web-app>
```

2.5.5. TCP 클라이언트 구현

TCP 클라이언트는 TCP 리스너와 소켓 연결을 맺고 이 연결을 통하여 메시지를 전송한다. 이 메시지는 [맞춤 통신 프로토콜 정의](#)에서 정의한 맞춤 통신 프로토콜에 준하는 Byte 스트림이다.

설정된 TCP 리스너는 메시지를 받아서 SampleConfig 클래스에 정의된 dispatch 정보를 바탕으로 SampleTCPServlet의 service() 메소드를 호출한다. SampleTCPServlet은 클라이언트로부터 전달된 데이터를 바탕으로 응답을 생성하여 전송한다. 이 응답은 클라이언트가 받아 System.out으로 출력한다.

다음은 그에 대한 예이다.

TCP 클라이언트 구현 예 : <Client.java>

```
package sample.client;

import java.io.*;
```

```

import java.net.*;

public class Client {
    private String address;
    private int port;

    private int magic = 7777;
    private byte type = 0;
    private int bodyLength = 128;
    private byte[] serviceName="sample".getBytes();

    public Client(String host, int port) {
        this.address = host;
        this.port = port;
    }

    public void test()
        throws IOException, UnsupportedEncodingException {
        Socket socket = new Socket(address, port);
        DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(socket.getOutputStream()));
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(socket.getInputStream()));

        out.writeInt(7777);
        out.write(type);
        out.writeInt(bodyLength);
        byte[] buf = new byte[32];
        System.arraycopy(serviceName, 0, buf, 0, serviceName.length);
        out.write(buf);
        byte[] msg1 = "test request".getBytes();
        byte[] msg2 = "test request2".getBytes();
        buf = new byte[64];
        System.arraycopy(msg1, 0, buf, 0, msg1.length);
        out.write(buf);
        buf = new byte[64];
        System.arraycopy(msg2, 0, buf, 0, msg2.length);
        out.write(buf);

        out.flush();

        // rx msg
        int magic = in.readInt();
        System.out.println("[Client] received magic = " + magic);

        byte type = (byte)in.read();
        System.out.println("[Client] received type = " + type);

        int len = in.readInt();
        System.out.println("[Client] received body length = " + len);

        byte[] svcname = new byte[32];
        in.readFully(svcname);
        System.out.println("[Client] received service name = " +
            (new String(svcname)).trim());

        byte[] body = new byte[128];
        in.readFully(body);
        String rcvmsg = null;
    }
}

```

```

        rcvmsg = (new String(body, 0, 64)).trim();
        System.out.println("[Client] received msg1 = " + rcvmsg);
        rcvmsg = (new String(body, 64, 64, "euc-kr")).trim();
        System.out.println("[Client] received msg2 = " + rcvmsg);

        out.close();
        in.close();
        socket.close();
    }

    public static void main(String[] argv) throws Exception {
        Client client = new Client("localhost", 5555);
        client.test();
    }
}

```

위의 클라이언트 코드에서 프로토콜에 필요한 다양한 헤더의 필드들의 설정을 주의 깊게 확인한다.

'magic' 수는 '7777'로, 'type'은 '0'(요청), 'body length'는 '128Bytes'(고정 길이), 'service name'은 'sample'로 설정되어 있다(**SampleServlet의 이름은 web.xml에 설정되어 있다**). 그리고 2개의 메시지들을 생성하여 헤더 정보를 전송한 후에 TCP 리스너에게 그들을 전송한다. 마지막으로 'hostname'을 'localhost'로 포트 번호는 '5555'로 설정하였다.

2.5.6. TCP 클라이언트 컴파일과 실행

TCP 리스너가 설정되어 'localhost'에 '5555' 포트를 이용하여 운영되고 있다고 가정한다. 이런 환경에서 TCP 클라이언트를 컴파일하고 실행하는 과정은 다음과 같다.

1. Client.java를 컴파일한다.

```
javac -classpath ${JEUS_HOME}/lib/system/jeus.jar -d . Client.java
```

2. Client.class를 다음과 같이 실행한다.

```
java -classpath ${JEUS_HOME}/lib/system/jeus.jar:. sample.client.Client
```

3. JEUS 관리자의 콘솔 화면은 다음과 같이 TCP 핸들러의 결과값을 보여줘야 한다(SampleServlet 클래스).

```

[SampleServlet] received magic = 7777
[SampleServlet] received type = 0
[SampleServlet] received body length = 128
[SampleServlet] received service name = sample
[SampleServlet] received msg1 = test request
[SampleServlet] received msg2 = test request2

```

4. 다음의 내용이 클라이언트의 실행 화면에 표시된다.


```
[Client] received magic = 7777
[Client] received type = 1
[Client] received body length = 128
[Client] received service name = sample
[Client] received msg1 = test response
[Client] received msg2 = test response2
```

2.6. HTTP/2 사용

HTTP/2는 HTTP/1.1에서 성능 향상을 위하여 새로 나온 HTTP 표준의 차기 버전이다.

다음은 HTTP/2의 특징에 대한 설명이다.

- **Header Compression**

HTTP/2에서는 기본적으로 헤더를 허프만 코딩으로 압축을 하여 보낸다. 또한, 반복되는 헤더가 여러 번 전송되지 않도록 서버, 클라이언트가 각자 헤더 테이블에 인덱싱을 해둔다. 얼마나 많은 양의 헤더를 인덱싱을 해둘지는 Settings Header Table Size로 설정할 수 있다.

- **Request/Response Multiplexing**

HTTP/2에서는 일련의 요청/응답 쌍을 하나의 스트림이라 부르며 하나의 스트림은 여러 개의 프레임(헤더 프레임, 데이터 프레임, ...)으로 이루어져 있다. 실제로 연결을 통해 전송되는 단위는 프레임이다.

HTTP/1.1까지는 하나의 요청/응답이 완료된 후에 다시 요청을 보낼 수 있었지만 HTTP/2에서는 여러 개의 스트림을 생성하여 동시에 여러 요청을 보낼 수 있다.

몇 개의 스트림을 동시에 사용할지는 Settings Max Concurrent Streams 설정을 통해 할 수 있다. 실제 데이터(body)가 전송되는 프레임인 데이터 프레임의 크기는 Settings Max Frame Size로 설정할 수 있다.

- **Server Push**

Server Push에 관한 내용은 [Server Push](#)를 참고한다.



본 안내서는 JEUS에서 HTTP/2 사용에 관한 내용만을 주로 다루고 있으며 HTTP/2에 관한 더 자세한 내용은 "[RFC 문서](#)"를 참고한다.

2.6.1. HTTP/2 사용 설정

HTTP/2는 h2c와 h2 두 가지 식별자를 정의하고 있다. 각각의 의미는 아래와 같다.

구분	설명
h2c	HTTP/2를 가리킨다. 'http'로 서비스된다.
h2	TLS(Transport Layer Security)를 사용해 동작하는 HTTP/2를 가리킨다. 'https'로 서비스된다.



대부분의 브라우저는 **h2c**를 지원하지 않고 **h2**만 지원한다.

HTTP/2를 **h2c**로 사용하기 위해서는 [HTTP 리스너 설정](#)을 참고하여 HTTP/2 사용 항목을 체크한다. HTTP/2를 h2로 사용하기 위해서 TLS(Transport Layer Security)를 사용해 동작하는데 TLS handshake 과정에서 ALPN(Application Layer Protocol Negotiation) 기능을 이용해야 한다. h2는 TLS를 사용해 동작하므로 당연히 보안 리스너로 설정한 포트를 사용해야 한다.

다음은 ALPN을 사용하기 위해 필요한 작업이다.



OpenJDK 8u252 버전 이후부터는 JDK 자체에서 ALPN을 지원하므로, JVM 옵션으로 Jetty ALPN 설정이 필요하지 않다. 각 JDK의 지원 기준이 다를 수 있으므로 이를 참고하여 설정한다.

1. 다음 주소에서 사용할 JDK 버전에 맞는 alpn-boot 라이브러리를 확인한 후 "[MVN Repository](#)"에서 다운받는다.

```
http://www.eclipse.org/jetty/documentation/current/alpn-chapter.html#alpn-versions
```



alpn-boot 라이브러리는 OpenJDK, OracleJDK만 지원한다. 따라서 현재 IBM JDK에서는 HTTP/2를 사용할 수 없다.

2. 다운받은 alpn-boot 라이브러리를 bootclasspath에 추가하는 <jvm-option>을 domain.xml에 추가한다.

```
<jvm-config>
  <jvm-option>-Xbootclasspath/p:<path_to_alpn_boot_jar> ...</jvm-option>
</jvm-config>
```



1. HTTP/2 스펙에서는 TLS 통신에 필요한 Cipher Suites 중 권장하지 않는 Cipher Suites를 명시하고 있다([TLS 1.2 Cipher Suite Black List](#)). 스펙에 따르면 권장하지 않는 Cipher Suites를 사용할 경우 연결이 끊어질 수 있으며 대부분의 브라우저는 이에 따라 연결을 끊고 있다. JDK 7에서 제공하는 Cipher Suites는 모두 권장하지 않고 있기 때문에 더 강력한 암호화 Suites를 사용하는 JDK 8 버전 사용이 필수이다.

HTTP/2 스펙에서 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256는 필수로 지원하도록 명시하고 있기 때문에 이 암호화 Suite를 Listener SSL 설정에서 설정하길 권장한다.

2. JEUS 서버와 관련된 세부 설정 사항에 대한 자세한 내용은 JEUS Server 안내서의 "Listener 설정"을 참고한다.

2.6.2. Server Push

Server Push는 지정된 리소스에 대하여 클라이언트가 요청을 하지 않아도 서버에서 응답을 주는 기능으로 요청 전송에 필요한 시간을 줄여 더 빠른 응답을 받을 수 있다. Server Push 기능을 사용하기 위해서는 웹 애플리케이션을 개발할 때에 Server Push할 리소스를 지정해야 한다. 본 절에서는 해당 기능의 사용법에 대해서 설명한다.



Servlet 4.0 API를 기반으로 Server Push를 구현하였다.

Server Push 사용의 기본적인 흐름은 PushBuilder 객체를 얻은 후 리소스의 경로를 지정하고 push() 메소드를 호출한다.

다음은 Server Push의 기본 흐름에 대한 예제 소스이다. 이외에도 Push될 리소스에 전달된 Query String, Cookie, Header 등을 바꿀 수 있다.

Server Push 사용 예 : <ServerPushServlet.java>

```
package sample.serverpush

...

@WebServlet("/ServerPush")
public class ServerPushServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
        IOException {

        PushBuilder builder = req.newPushBuilder();
        String pushResourcePath = "resources/a.txt";

        // Push할 리소스의 패스를 넣어준 후 push()함수를 호출한다.
        builder.path(pushResourcePath);
        builder.push();

        resp.getWriter().write("main page\n");
    }
}
```



Server Push를 사용하지 않겠다고 설정한 경우 위 예제는 아무 영향을 미치지 않는다.

2.7. 리스너 튜닝

최적의 성능을 위하여 리스너를 설정할 때 다음의 몇 가지 사항을 고려해야 한다.

- 시스템 리소스를 많이 사용하거나 대기시간을 길게 하여 출력 버퍼의 크기를 증가시킨다.
- 일반적으로 Worker Thread Pool의 min, max 값을 크게 부여하면 웹 엔진에 많은 클라이언트가 접근할 때 Pool이 좋은 성능을 가지게 된다. 시스템 메모리를 적게 사용하기 위해서는 이들의 값을 낮게 설정한다.
- '**Server Access Control**' 옵션을 비활성화하면 성능 개선을 기대할 수 있다.

- WebtoB 커넥터에서는 앞에서 언급했던 공식에 따라 웹 엔진의 WebtoB 커넥터의 Worker Thread 개수와 http.m의 값들을 동일하게 설정한다. 이렇게 해야 가장 좋은 성능을 기대할 수 있다.

3. 웹 컨텍스트

본 장에서는 JEUS 웹 애플리케이션을 패키징하는 방법과 이를 JEUS 웹 엔진에 deploy하고 모니터링하는 방법에 대해 설명한다.

3.1. 개요

웹 애플리케이션을 웹 엔진에 deploy하면 웹 컨텍스트가 생성된다. 즉, 웹 애플리케이션과 웹 컨텍스트는 동일한 의미로 봐도 무방하므로 두 용어를 혼용하지 않고 웹 컨텍스트로 통일해서 사용한다.

3.2. 기본 구조

본 절에서는 웹 컨텍스트가 포함하는 내용과 내부 구조에 대해서 설명한다.

3.2.1. 웹 컨텍스트 콘텐츠

웹 애플리케이션은 클라이언트의 요청에 의한 웹 기반의 서비스를 수행하기 위한 정적 콘텐츠와 동적 콘텐츠로 구성되어 있다.

- 정적 콘텐츠

사전에 이미 완성되어 있는 형태로 웹 엔진에서 어떠한 처리도 하지 않고 반환하는 모든 종류의 데이터들을 의미한다. 정적 콘텐츠의 종류는 다음과 같다.

- HTML 페이지
- 텍스트 파일
- 이미지 파일
- 비디오
- 기타

- 동적 콘텐츠

서블릿, JSP와 같이 사용자의 요청에 따라 응답을 생성하는 콘텐츠들을 의미한다.

3.2.2. 웹 컨텍스트 내부 구조(WAR 파일 구조)

웹 컨텍스트는 웹 엔진에 deploy하기 전에 .war 확장자를 가진 JAR 파일 형식의 압축 파일로 패키징해야 한다. 이를 WAR(Web ARchive) 파일이라고 한다.

WAR 파일의 구조는 다음과 같다.

```
WEB WAR
|--[T]index.html
```

```

|--static
|   |--[T].html
|--jsp
|   |--.jsp
|--images
|   |--[T].jpg, .gif
|--META-INF
|   |--[T]MANIFEST.MF
|--WEB-INF
|   |--[X]web.xml
|   |--[X]jeus-web-dd.xml
|   |--[X]ejb-jar.xml
|   |--[X]jeus-ejb-dd.xml
|--classes
|   |--[C]Servlet.class
|   |--[C]EJB.class
|--lib
|   |--[J]Library JAR
|--tlds
|   |--[T].tld

```

* Legend

- [01]: binary or executable file
- [X] : XML document
- [J] : JAR file
- [T] : Text file
- [C] : Class file
- [V] : java source file
- [DD] : deployment descriptor

META-INF/

이 디렉터리는 선택 사항이다. 이 디렉터리를 사용하는 경우에는 JAR 포맷에 정의된 Descriptor 파일인 MANIFEST.MF 파일을 포함한다.

WEB-INF/

이 디렉터리는 필수 사항이고 서블릿, 필터, 리스너 클래스들, 라이브러리들이 포함되어 있다. 이 디렉터리 하위에는 다음과 같은 컴포넌트들이 존재한다.

컴포넌트	설명
web.xml	Jakarta EE 표준 웹 애플리케이션 DD 파일로 웹 애플리케이션의 메타 정보를 가지고 있다. Servlet 3.0부터는 web.xml이 반드시 필요하지 않다. 대신 Annotation, Registration API를 통해서 서블릿, 필터, 리스너들을 추가할 수 있다. 자세한 사항은 서블릿 표준을 참고한다.
jeus-web-dd.xml	JEUS의 웹 애플리케이션 DD로 자세한 설명은 jeus-web-dd.xml 설정 을 참고한다.
ejb-jar.xml	Java EE 6부터 WAR 파일에 EJB를 포함할 수 있다. 서블릿과 마찬가지로 Annotation으로 정의할 수 있다. 이에 대한 자세한 설명은 EJB 표준 또는 "JEUS EJB 안내서"를 참고한다.
jeus-ejb-dd.xml	
classes/	서블릿 클래스들과 유틸리티 클래스들이 하위 디렉터리에 포함되어 있다. 표준 Java 패키지 구조로 정리되어 있다.

컴포넌트	설명
lib/	웹 애플리케이션에 필요한 Java 라이브러리들을 포함한다. 라이브러리들은 .jar 파일들로 패키징되어 이 경로에 저장된다. 이 파일들의 내용들은 자동적으로 모든 서블릿의 클래스 경로에 추가된다.
tlds/	JSP 페이지들을 위한 Custom Tag Library Descriptor들을 포함한다.

기타

JSP, HTML, 이미지 파일들과 같은 콘텐츠를 포함하고 있는 임의의 이름의 디렉터리이다.

3.3. 웹 컨텍스트 Deploy

본 절에서는 웹 컨텍스트의 Deploy에 대해 설명한다. Deploy는 웹 애플리케이션을 JEUS 웹 엔진에서 서비스할 수 있도록 준비하여 웹 컨텍스트를 생성하는 과정이다.

웹 컨텍스트는 논리적으로 가상 호스트 하위에 존재한다. 가상 호스트에 대한 더 자세한 사항은 [가상 호스트](#)를 참고한다.

3.3.1. jeus-web-dd.xml 설정

반드시 필요한 경우에 한해 WEB-INF/ 디렉터리 아래에 web.xml과 jeus-web-dd.xml을 생성한다.

다음은 jeus-web-dd.xml 파일의 예이다. 몇몇 항목들은 생략되어 있다. 생략된 항목들은 "JEUS XML Reference"의 "13. jeus-web-dd.xml 설정"을 참고한다.

웹 컨텍스트 설정 파일 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
  <context-path>/examples</context-path>
  <enable-jsp>true</enable-jsp>
  <auto-reload>
    <enable-reload>true</enable-reload>
    <check-on-demand>true</check-on-demand>
  </auto-reload>
  <added-classpath>
    <class-path>/home/user1/libs/lib.jar</class-path>
  </added-classpath>
  <session-config>
    . . .
  </session-config>
  <thread-pool>
    <base>
      <name>base</name>
      <min>1</min>
      <max>5</max>
    </base>
    <service-group>
      <name>svcg1</name>
      <uri>/service1,/service2</uri>
      <min>10</min>
      <max>20</max>
    </service-group>
  </thread-pool>
</jeus-web-dd>
```

```

    </service-group>
    <service-group>
      <name>longJsp</name>
      <uri>/jsp/servlet/asyncContext.jsp</uri>
      <min>5</min>
      <max>10</max>
    </service-group>
    <service-group>
      <name>Jsp</name>
      <uri>/jsp/servlet</uri>
      <min>1</min>
      <max>5</max>
    </service-group>
  </thread-pool>
  <upgrade>
    <upgrade-executor>
      <min>0</min>
      <max>30</max>
      <keep-alive-time>60000</keep-alive-time>
      <queue-size>4096</queue-size>
    </upgrade-executor>
  </upgrade>
  <async-config>
    <async-timeout-millis>60000</async-timeout-millis>
    <background-thread-pool>
      <min>0</min>
      <max>10</max>
    </background-thread-pool>
    <dispatch-thread-pool>
      <min>0</min>
      <max>10</max>
    </dispatch-thread-pool>
  </async-config>
  <properties>
    <property>
      <key>jeus.servlet.jsp.modern</key>
      <value>true</value>
    </property>
  </properties>
</jeus-web-dd>

```

다음은 설정 태그에 대한 설명이다.

태그	설명
<context-path>	<p>웹 컨텍스트에 접근하기 위한 Root Path이다. 이것은 반드시 '/'로 시작해야 하며 가상 호스트 내에서 유일해야 한다.</p> <p>"http://www.foo.com/examples/index.html"과 같은 URL에서 <context path>는 "/examples"가 된다.</p>
<enable-jsp>	false로 설정했을 때 JSP 기능을 사용하지 않는다. 기본적으로 JSP를 지원한다.
<user-log>	웹 컨텍스트가 남기는 로그이다. 유저 로그에 대한 자세한 내용은 "유저 로그 설정" 을 참고한다.
<auto-reload>	디스크의 서블릿 클래스가 변경되었을 때 자동적으로 로딩할지 여부를 결정하는 옵션이다. 자세한 내용은 클래스 동적 반영 을 참고한다.

태그	설명
<added-classpath>	서블릿을 실행할 때 참조할 라이브러리들의 리스트이다. 디렉터리도 설정 가능하다.
<allow-indexing>	클라이언트가 요청했을 때 실제 디렉터리 목록이 출력되는 URL 경로의 목록이다. 클라이언트에서 요청한 파일이 디렉터리에 없을 경우에 디렉터리의 목록이 출력된다. 대신에 디렉터리 목록의 파일을 하나 선택하면 해당 파일의 내용이 출력된다.
<deny-download>	어떤 경우에도 직접적으로 다운로드받거나 접근할 수 없는 자원들을 규정하기 위한 필터이다. 이 필터들은 파일 이름의 확장자로 파일 이름과 함께 경로로 지정할 수 있다. 필터의 파일 이름과 디렉터리들은 context document base 디렉터리의 상대적인 경로로 설정된다.
<aliasing>	Custom URL 경로에 대한 디렉터리 매핑을 의미한다. 다른 위치의 디렉터리를 URL 요청 경로에 매핑시킬 필요가 있는 경우에 aliasing 기능을 사용한다.
<file-caching>	응답 속도 향상을 위해 런타임 메모리에 어떤 정적 콘텐츠를 Cache해야 할지 결정한다. 이 항목의 하위 설정들은 Cache 메모리의 최대 크기(단위: MB), 요청되지 않을 때 Cache 내에 머무를 수 있는 정적 콘텐츠의 최대 크기, caching되어야 할 콘텐츠의 경로가 있다.
<jsp-engine>	웹 컨텍스트에 포함된 JSP 페이지에 대한 설정이다. 자세한 내용은 jeus-web-dd.xml 설정 을 참고한다.
<session-config>	웹 컨텍스트 레벨의 세션을 설정한다. 이 설정은 웹 엔진에 설정한 것보다 우선순위가 높다. 세션 설정에 대한 자세한 내용은 세션 설정 을 참고한다.
<webinf-first>	클래스 로딩 우선순위에 관한 옵션으로 true로 설정하면 해당 애플리케이션에 한해서 WEB-INF 하위의 클래스를 우선적으로 로딩한다. true로 설정하였을 때 클래스 로딩 관련 충돌이 발생할 경우 아래와 같이 특정 패키지 또는 클래스만 제외할 수 있는 기능이 JEUS 7 Fix#4부터 추가되었다. <pre> <webinf-first> <enabled>true</enabled> <excluded-package>package.name.</excluded-package> ... </webinf-first> </pre>
<attach-stacktrace-on-error>	서버에 문제가 발생했을 때 오류의 상세 내역을 브라우저로 보여줄 것인지에 대한 설정이다. 이 메시지는 개발하는 경우에는 유용하지만 운영하는 경우에는 제거하는 것이 좋다.

태그	설명
<keep-alive-error-response-codes>	<p>에러 응답 코드 중에서 Connection: Close 대신 Keep-Alive 응답을 주길 원하는 값을 설정한다. 웹 애플리케이션이 직접 에러 응답을 내보낼 때 적용되며, 엔진 내부적으로 커넥션을 끊어야 할 필요가 있다고 판단한 경우에는 적용되지 않는다.</p> <p>예를 들어 서블릿이 response.sendError(500)을 수행한 경우에는 적용되지만 서블릿에서 exception이 발생해서 엔진이 500 에러를 보내는 경우에는 적용되지 않는다. 여러 개의 응답 코드를 설정하려면 "404,503"과 같이 콤마(,)로 구분한다.</p>
<encoding>	<p>Request, Response에 대한 인코딩 설정이다. 웹 엔진 설정(domain.xml)과 동일하지만 동시에 설정될 경우 jeus-web-dd.xml의 <request-encoding> 설정과 <response-encoding> 설정이 각각 우선적으로 적용된다. 자세한 내용은 인코딩 설정을 참고한다.</p> <ul style="list-style-type: none"> ◦ Request Encoding : URL Query String으로 application/x-www-form-urlencoded 형식의 바디이다. getReader()로 호출된다. JEUS 7 Fix#4부터 <request-encoding><url-mapping> 설정이 추가되었다. 이는 특정 Servlet Path에 매핑된 인코딩을 적용하는 설정이다. 단, <forced> 또는 <client-override> 설정이 있는 경우에는 무시된다. ◦ Response Encoding : 응답 HTTP 바디에 적용되는 인코딩이다.
<cookie-policy>	<p>HTTP Request Header에 있는 쿠키를 읽거나 애플리케이션 요청에 의해 새로운 쿠키를 HTTP Response Header를 사용할 때 적용하는 정책을 설정할 수 있다. 자세한 내용은 쿠키 정책 설정을 참고한다.</p>
<async-config>	<p>Servlet 3.0부터 추가된 Async Processing에 관한 설정이다. 하위에는 Async Timeout, Background Thread Pool, Dispatch Thread Pool에 대한 설정을 할 수 있다. Async Processing에 관한 자세한 내용은 서블릿 표준을 참고한다.</p>
<upgrade>	<p>Upgrade Inbound Message를 처리하기 위해서 Container 내부적으로 사용하는 쓰레드 풀 관련 설정이다.</p>
<web-security>	<p>sendRedirect할 경우에 주소값 검증에 대한 정책 설정 등 웹 애플리케이션에 국한된 보안 정책들의 설정 그룹이다.</p>
<websocket>	<p>웹 애플리케이션의 WebSocket Container에 대한 설정이다. 자세한 내용은 WebSocket 컨테이너 설정을 참고한다.</p>
<properties>	<p>웹 애플리케이션에 적용되는 프로퍼티를 설정한다. 여기에 설정된 프로퍼티는 다른 웹 애플리케이션에는 적용되지 않고, 설정된 웹 애플리케이션에만 적용된다.</p>



<async-config><async-timeout-millis>의 기본값은 30초이다. 만약 Background Thread에서 수행하는 작업이 너무 오래 걸리는 경우가 있어서 타임아웃 에러가 발생한다면 jeus-web-dd.xml에 이 값을 설정하거나 jakarta.servlet.AsyncContext#setTimeout()으로 프로그램에서 조절한다.

다음은 Thread Pool 태그에 대한 설명이다.

태그	설명
<base>	Service Group의 uri에 매칭되지 않는 요청들을 처리할 default Thread Pool이다. 상위의 Thread Pool인 Virtual Host 또는 Web-Connection Thread Pool에서 처리하게 하려면 설정하면 안된다.
<service-group>	uri에 매칭되는 service들을 처리할 Thread Pool이다.
<name>	base와 service-group들을 구분하기 위한 이름이다.
<uri>	service-group Thread Pool에서 처리할 요청들의 uri이다. "," 로 구분해서 하나 이상의 resource를 나타낼 수 있으며 각 uri는 '/' 로 시작해야 한다.

3.3.2. 웹 컨텍스트 Redeploy(Graceful Redeploy)

웹 컨텍스트의 Redeploy에 대한 내용은 본 안내서에서는 생략한다. 자세한 내용은 JEUS Applications & Deployment 안내서의 "Graceful Redeployment"를 참고한다.

3.3.3. Shared Library Reference 추가

공유 라이브러리(Shared Library)는 애플리케이션별로 필요한 라이브러리를 WEB-INF/lib가 아닌 JEUS_HOME/lib/shared 아래에 추가한 후에 여러 애플리케이션에서 공유하여 사용할 수 있다. JEUS의 재기동 없이도 libraries.xml 파일 수정 후 모듈을 다시 deploy하면 변경된 라이브러리가 반영된다.

공유 라이브러리를 추가하려면 다음과 같이 JEUS_HOME/lib/shared/libraries.xml 파일에서 <library> 태그를 추가하고 공유할 JAR 파일들을 지정한다. 버전은 스펙 버전과 구현체 버전을 각각 지정할 수 있다.

Shared Library Reference 추가 : <libraries.xml>

```
<libraries xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
  <!-- DO NOT MODIFY OUR SYSTEM libraries!!! -->
  <library>
    <library-name>jsf</library-name>
    <specification-version>1.2</specification-version>
    <implementation-version>1.2.15</implementation-version>
    <files dir="jsf_r1_1.2"/>
  </library>
  <library>
    <library-name>jsf</library-name>
    <specification-version>2.2</specification-version>
    <implementation-version>2.2.13</implementation-version>
    <files dir="jsf_r1_2.2"/>
  </library>
  <library>
    <library-name>jsf-jeusport</library-name>
    <files dir=".">
      <include name="jsf-injection-provider.jar"/>
    </files>
  </library>
  <library>
    <library-name>jsf-weld-integration</library-name>
    <files dir=".">
      <include name="jsf-weld-integration.jar"/>
    </files>
  </library>
  <library>
```

```

    <library-name>jstl</library-name>
    <specification-version>1.2</specification-version>
    <implementation-version>1.2</implementation-version>
    <files dir="jstl_1.2"/>
</library>
<library>
    <library-name>jstl</library-name>
    <specification-version>1.2</specification-version>
    <implementation-version>1.2</implementation-version>
    <files dir="jstl_1.2"/>
</library>
<library>
    <library-name>spring-support</library-name>
    <specification-version>4.2.0.RELEASE</specification-version>
    <implementation-version>4.2.0.RELEASE</implementation-version>
    <files dir="spring-support/4.2.0.RELEASE"/>
</library>
<library>
    <library-name>spring-support</library-name>
    <specification-version>4.0.0.RELEASE</specification-version>
    <implementation-version>4.0.0.RELEASE</implementation-version>
    <files dir="spring-support/4.0.0.RELEASE"/>
</library>

<!-- Add user libraries from here -->
<library>
    <library-name>myLibrary</library-name>
    <specification-version>2.0</specification-version>
    <implementation-version>2.1</implementation-version>
    <files dir=".">
        <include name="commons-logging.jar"/>
        <include name="commons-util.jar"/>
    </files>
    <files dir="myLib-2.1"/>
</library>
</libraries>

```

설정한 공유 라이브러리를 참조하려면 jeus-web-dd.xml에 다음과 같이 <library-ref>를 설정한다. 버전이 명시되지 않은 경우는 libraries.xml에 등록된 동일한 이름의 라이브러리 중에서 최상위 버전을 참조한다.

```

<library-ref>
    <library-name>myLibrary</library-name>
    <specification-version>
        <value>2.0</value>
    </specification-version>
</library-ref>

```

JSF, JSTL 라이브러리 설정

Shared Library Reference 추가 : <libraries.xml>에 설명한 libraries.xml 파일에는 JSF와 JSTL이 공유 라이브러리로 추가되어 있다. Jakarta EE 표준은 JSF와 JSTL을 포함하고 있기 때문에 JEUS에서 기본적으로 제공한다.

JEUS 시스템 라이브러리로 제공할 수도 있지만 하나의 JSF, JSTL 구현체만 사용할 수 있는 제약 사항이 발생한다. JSF

구현체는 2.0뿐만 아니라 1.2 버전도 있고 Sun RI가 아닌 Apache의 MyFaces도 있기 때문에 JEUS에서는 여러 가지 버전의 구현체를 사용할 수 있도록 공유 라이브러리 형태로 JSF, JSTL을 제공한다.

구현체의 위치와 종류에 따라 사용되는 구현체는 다음과 같이 구분된다.

- WEB-INF/lib 아래에 포함한 JSF나 JSTL 구현체
- Shared Library로 포함시킨 구현체(위의 구현체보다 우선 순위가 낮다.)

[참고]

다음은 JEUS 7 Fix#2 버전부터 더이상 JEUS_HOME/lib/shared 아래의 Oracle JSF RI를 자동으로 추가해주지 않는다. 이를 사용하려면 명시적으로 jeus-web-dd.xml에서 <library-ref>로 참조하도록 설정해야 한다.

다음과 같이 선언하면 JEUS_HOME/lib/shared에 기본으로 포함된 라이브러리를 사용할 수 있다.

```
<library-ref>
  <library-name>jsf</library-name>
</library-ref>
```

또한 JEUS 내부적으로 판단해서 Oracle JSF RI 2.x에 포함된 웹 리스너를 자동으로 추가하지 않는다. 명시적으로 web.xml에서 <listener>로 설정해야 한다.

```
<listener>
  <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
```

동일한 JSF나 JSTL 라이브러리를 여러 애플리케이션에서 참조해서 사용하려면 직접 JEUS_HOME/lib/shared/libraries.xml에 <library>를 등록하고 jeus-web-dd.xml에서 <library-ref>로 참조하도록 설정한다.

JSF의 경우 Managed Bean에서 @EJB나 @Resource와 같은 Annotation으로 WAS가 제공하는 리소스에 접근할 수 있고 WAS는 Annotation으로 명시된 리소스들을 injection할 수 있어야 한다.

JEUS에서는 JSF RI에 JEUS Injection을 지원하기 위해서 공유 라이브러리로 jsf-injection-provider.jar를 제공하고 있다. 공유 라이브러리는 다음의 디렉터리에 존재한다.

```
JEUS_HOME/lib/shared
```

이는 web.xml에 jakarta.faces.webapp.FacesServlet이 포함된 경우 자동으로 포함된다.

Spring Support 라이브러리 설정

[Shared Library Reference 추가](#) : <libraries.xml>에 설명한 libraries.xml 파일에는 Spring Support 라이브러리가 공유 라이브러리로 추가되어 있다.

Spring Support 라이브러리는 Spring Framework을 이용해 만든 애플리케이션이 JEUS의 기능을 사용하기 위해 필요하다. 이 절에서는 WebSocket 관련 기능만 설명한다.

Spring Framework을 사용하여 WebSocket 기능을 구현 할 경우 spring-websocket 모듈을 사용하게 된다. 하지만, 이 모듈에서 JEUS의 WebSocket Handshake를 지원하고 있지 않기 때문에 JEUS에서 정상동작 되지 않는 문제가 있다. Spring Support 라이브러리를 사용하면 JEUS의 WebSocket 기능을 정상적으로 사용할 수 있다.

Spring Support 라이브러리의 WebSocket 기능을 사용하기 위해서는 jeus-web-dd.xml에서 <library-ref>로 참조 설정을 한 후 다음과 같이 Spring 설정에 관련 bean을 추가해 주어야 한다.

```
<beans ...>
  <websocket:handlers>
    <websocket:mapping .../>
    <websocket:handshake-handler ref="handshakeHandler"/>
  </websocket:handlers>

  <bean id="handshakeHandler"
class="org.springframework.web.socket.server.support.DefaultHandshakeHandler">
    <constructor-arg ref="upgradeStrategy"/>
  </bean>
  <bean id="upgradeStrategy" class="jeus.spring.websocket.JeusRequestUpgradeStrategy"/>
</beans>
```

4.2.0.RELEASE 버전 라이브러리를 사용할 경우 아래와 같이 설정할 수 있다.

```
<beans ...>
  <websocket:handlers>
    <websocket:mapping .../>
    <websocket:handshake-handler ref="handshakeHandler"/>
  </websocket:handlers>

  <bean id="handshakeHandler" class="jeus.spring.websocket.JeusHandshakeHandler"/>
</beans>
```

3.3.4. 호환성을 위한 웹 컨텍스트 레벨의 옵션 설정

이전 JEUS 버전이나 타 WAS와 호환성에 문제가 발생한 경우에 웹 컨텍스트 레벨에서 옵션을 설정하여 해결할 수 있다.

이전 JEUS와의 호환성

Servlet 2.4 표준 이전에는 표준 자체에 제약 사항이 제대로 기술되지 않은 경우가 많았다. 이는 Servlet 2.4, 2.5를 거치면서 보완되었고 JEUS에서도 자연스럽게 제약 사항이 생기게 되었다. 또한 JEUS도 여러 가지 문제점을 경험하고 이를 수정하면서 표준을 따르지 않는 동작들을 지원하지 않게 되었다. 이에 따라 JEUS 4에서 작성한 웹 애플리케이션이 JEUS 6에서 제대로 동작하지 않는 문제들이 발생할 수 있다.

JEUS는 Servlet 및 JSP 표준을 준수해야 할 책임이 있지만 기존 애플리케이션의 동작 호환성을 보장해야 할 책임도 있다. 하지만 이 두 가지 책임은 서로 상충하는 면이 있다. 주로 표준 준수를 위해서 jakarta.servlet 표준 API, 내부 동작을 수정했을 때, 기존의 잘못된 동작에 기반하여 작성한 애플리케이션이 있을 때 문제가 발생한다. 이런 상황에서는 기본적으로 표준 준수 요구 사항이 우선순위가 더 높기 때문에 해당 애플리케이션을 수정해야 할 필요가 있다. 왜냐하면 그 애플리케이션을 다른 웹 컨테이너에서는 실행할 수 없기 때문이다. 또한 시간이 지남에 따라

애플리케이션을 업그레이드하는 과정에서 서블릿 표준을 준수하는 프레임워크들과도 충돌하는 문제가 발생할 수 있다.

하지만 이러한 애플리케이션의 문제점을 초기에 파악하지 못하고 서비스 개시일을 앞두고 발견한 경우 애플리케이션의 수정 대신 JEUS의 동작을 수정하는 요구 사항이 발생한다. 이를 지원하기 위해서 JEUS에서는 하위 호환성을 위한 프로퍼티를 제공한다.

JSP 엔진에 관한 호환성 문제는 [JSP 하위 호환성을 위한 웹 컨텍스트 레벨의 옵션 설정](#)을 참고하고, 하위 호환성을 위해 JEUS에서 제공하는 프로퍼티에 대한 자세한 내용은 JEUS Reference 안내서의 "호환성 제공 프로퍼티"를 참고한다.

다른 WAS와의 호환성

Servlet 2.4 표준 이전에는 Servlet과 JSP를 직접 사용해서 웹 애플리케이션을 개발했지만 현재는 웹 프레임워크를 사용하는 추세이고, Java EE 5부터는 표준 웹 프레임워크로 JSF와 JSTL이 추가되었다. 이러한 웹 프레임워크는 주로 Servlet, JSP 스펙에 대한 참조 구현체인 Tomcat을 사용해서 개발하고 테스트하기 때문에 웹 프레임워크들이 비표준적인 기능을 지원할 경우에 다양한 호환성 관련 문제가 발생할 수 있다.

호환성에 문제가 발생했을 때 필요한 경우에 한해서 jeus-web-dd.xml에 설정할 수 있다. jeus-web-dd.xml의 설정에 대한 자세한 내용은 JEUS Reference 안내서의 "웹 엔진 프로퍼티"를 참고한다.

3.3.5. 부가 기능 사용을 위한 설정

JEUS에서 제공하는 다음의 부가 기능을 사용하려면 별도의 설정이 필요하다.

- 정적 리소스를 처리하는 서블릿 매핑

서블릿 표준에서는 Servlet 또는 JSP가 아닌 경우 WAS에서 제공하는 Default Servlet으로 처리하도록 되어 있다. 이를 ResourceServlet이라고 한다.

다음과 같이 web.xml에서 <servlet-mapping> 설정에 jeus.servlet.servlets.ResourceServlet으로 매핑한다.

정적 리소스를 처리하는 서블릿 매핑 : <web.xml>

```
<servlet-mapping>
  <servlet-name>jeus.servlet.servlets.ResourceServlet</servlet-name>
  <url-pattern>/static/*</url-pattern>
</servlet-mapping>
```

- Spring 3.0 이상에서 mvc:default-servlet-handler 등록 방법

JEUS 9에서는 Spring 3.0 이상에서 mvc:default-servlet-handler을 더이상 등록할 필요가 없다. Tomcat과 마찬가지로 "default"라는 이름으로 JEUS의 default servlet을 제공한다.

- Servlet Multipart 기능 사용하는 경우 ProgressListener 등록 방법

Servlet 3.0부터는 서블릿별로 multipart-config 설정을 등록해서 POST 요청의 multipart/form-data 처리를 웹 컨테이너에 맡길 수 있다. 이때 처리 진행 상황에 대한 이벤트를 받고 싶은 경우 ProgressListener를

등록한다.

1. jeus-servlet.jar에 포함된 **jeus.servlet.engine.multipart.ProgressListener** 인터페이스를 구현해야 한다. 인터페이스에 관한 내용은 Servlet API 문서를 참고한다.
2. 위 인터페이스 구현 클래스의 인스턴스를 `HttpServletRequest` attribute에 추가한다. 이름은 인터페이스 이름과 마찬가지로 **jeus.servlet.engine.multipart.ProgressListener** 이다. 이때 주의할 사항은 반드시 `getPart()`, `getParts()` 또는 `getParameter()`를 호출하기 이전이어야 한다.

3.3.6. 웹 보안 설정

jeus-web-dd.xml을 통해 애플리케이션별로 적용할 웹 보안 설정에 대해 설명한다.

Redirect Location 보안 설정

애플리케이션은 `jakarta.servlet.http.HttpServletResponse.sendRedirect(String location)` 표준 API를 통해서 "302 Found" 응답을 보낼 수 있다. 이때 기본적으로 Location으로 넘겨주는 문자열에 대해서 아무런 확인을 하지 않고 그대로 URL로 전환해서 Location 헤더로 설정한다. 그렇기 때문에 악의적인 사용자가 CRLF injection과 같은 공격을 할 수 있다. 이를 방지하기 위해 애플리케이션은 `jeus.servlet.security.RedirectStrategy` 인터페이스를 구현해서 jeus-web-dd.xml에 설정할 수 있다.

다음은 `jeus.servlet.security.RedirectStrategy` 인터페이스를 설정한 예이다.

Redirect Location 보안 설정 인터페이스

```
package jeus.servlet.security;

import jakarta.servlet.http.HttpServletRequest;

public interface RedirectStrategy {
    /**
     * Makes the redirect URL.
     *
     * @param location the target URL to redirect to, for example "/login"
     */
    String makeRedirectURL(HttpServletRequest request, String location)
        throws IllegalArgumentException;
}
```

설정된 인터페이스를 다음과 같이 구현할 수 있다.

Redirect Location 보안 설정 구현 예

```
package jeus.servlet.security;
import jakarta.servlet.http.HttpServletRequest;

public class RejectCrLfRedirectStrategy implements RedirectStrategy {
    private Pattern CR_OR_LF = Pattern.compile("\\r|\\n");

    @Override
    String String makeRedirectURL(HttpServletRequest request, String location)
        throws IllegalArgumentException {
        if (CR_OR_LF.matcher(location).find()) {
```



```

        throw new IllegalArgumentException("invalid characters (CR/LF) in redirect location");
    }
    return makeAbsolute(location);
}

private String makeAbsolute(String location) {
    // make code for make absolute path
}
}

```

jeus-web-dd.xml의 설정 방법은 다음과 같다.

Redirect Location 보안 설정 : <jeus-web-dd.xml>

```

...
<web-security>
    <redirect-strategy-ref>
        jeus.servlet.security.RejectCrLfRedirectStrategy
    </redirect-strategy-ref>
</web-security>
..

```

<redirect-strategy-ref>는 jeus.servlet.security.RedirectStrategy 인터페이스를 구현한 클래스 이름이다. 이 클래스는 웹 애플리케이션의 클래스 패스에 포함시키면 된다. 위에서 제시한 jeus.servlet.security.RejectCrLfRedirectStrategy의 경우 기본적으로 제공하는 RedirectStrategy이며 CR, LF 또는 CRLF가 있는 Location이 sendRedirect API로 넘어올 경우 500 에러가 발생한다.

그 외에도 CR, LF 또는 CRLF 문자열을 빈 문자열로 치환해주는 jeus.servlet.security.RemoveCrLfRedirectStrategy를 제공한다.

3.3.7. 보안 Role 매핑

web.xml에 정의된 보안 Role을 실제 시스템 사용자와 사용자 그룹에 설정해야 한다. 이 매핑은 jeus-web-dd.xml에 기술된다.

다음의 내용을 web.xml에 정의했다고 가정한다.

보안 Role 매핑 : <web.xml>

```

<web-app>
    <security-role>
        <role-name>manager</role-name>
    </security-role>
    <security-role>
        <role-name>developer</role-name>
    </security-role>
    <servlet>
        . . .
    </servlet>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>
                MyResource
            </web-resource-name>
        </web-resource-collection>
    </security-constraint>

```

```

        </web-resource-name>
        <url-pattern>/jsp/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>
</web-app>

```

위의 예제에서는 2개의 보안 Role이 선언되어 있다(굵은 글씨로 표현된 부분). 세 번째 굵은 글씨로 표현된 부분은 manager role이 <security-constraint> 태그에 어떻게 사용되는지 나타낸다.

이 매핑을 jeus-web-dd.xml에 기술하면 다음과 같다.

보안 Role 매핑 : <jeus-web-dd.xml>

```

<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
    . . .
    <role-mapping>
        <role-permission>
            <principal>Peter</principal>
            <role>manager</role>
        </role-permission>
        <role-permission>
            <principal>Linda</principal>
            <role>developer</role>
        </role-permission>
    </role-mapping>
    . . .
</jeus-web-dd>

```

웹 애플리케이션이 실제의 시스템에 deploy될 때 "manager"와 "developer" Role을 시스템의 특정한 사용자로 바인딩해야 한다. 이 매핑은 <context><role-mapping>에 정의되어 있다. <role-mapping> 태그는 <role-permission> 태그를 포함한다. 이 태그를 이용하여 <principal-to-role> 매핑을 정의한다.

다음의 하위 태그들과 함께 바인딩된다. 예제에서 2개의 Role "manager"와 "developer"가 "Peter"와 "Linda"로 각각 매핑되어 있다.

태그	설명
<role>(1개, 필수)	web.xml에 정의된 <role-name> 값이며, 위의 예에서는 <role-name>이 "manager"이다.
<principal>(0개 이상)	<role-name>과 연계되어야 할 JEUS가 관리하는 사용자의 이름이다. 자세한 내용은 JEUS Security 안내서의 "웹 모듈 보안 설정"을 참고한다.

3.3.8. Symbolic Reference 매핑

Role이 실제 사용자에게 매핑되듯이 EJB Reference, Resource Reference, 관리되는 객체의 Reference를 실제 시스템 자원에 매핑할 필요가 있다.

다음은 Symbolic Reference 매핑의 예이다.

Symbolic Reference 매핑 : <web.xml>

```
<web-app>
. . .
<ejb-ref>
  <ejb-ref-name>ejb/account</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.mycompany.AccountHome</home>
  <remote>com.mycompany.Account</remote>
</ejb-ref>
<resource-ref>
  <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>
    jms/StockQueue
  </resource-env-ref-name>
  <resource-env-ref-type>
    jakarta.jms.Queue
  </resource-env-ref-type>
</resource-env-ref>
. . .
</web-app>
```

이 웹 애플리케이션을 등록하기 위해 web.xml의 <ejb-ref>, <resource-ref>, <resource-env-ref> 아래의 모든 Symbolic Reference 이름들이 실제 의도하는 자원의 JNDI 이름과 매핑되어야 한다. 이 매핑은 jeus-web-dd.xml에서 해당 <ejb-ref>, <res-ref>, <res-env-ref>를 추가하면 완성된다.

이 태그들은 다음의 하위 태그들을 포함한다.

- <jndi-info>

태그	설명
<ref-name>	Reference의 이름으로 web.xml과 서블릿 코드에 정의된 것과 같은 것이다.
<export-name>	JNDI Export Name은 <ref-name>이 가리키는 실제 리소스의 Export Name이다.

다음은 JNDI 이름이 위의 3개의 Reference와 매핑된 jeus-web-dd.xml의 예이다.

Symbolic Reference 매핑 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
. . .
<ejb-ref>
  <jndi-info>
    <ref-name>ejb/account</ref-name>
    <export-name>AccountEJB</export-name>
  </jndi-info>
</ejb-ref>
<res-ref>
  <jndi-info>
```

```

        <ref-name>jdbc/EmployeeAppDB</ref-name>
        <export-name>EmployeeDB</export-name>
    </jndi-info>
</res-ref>
<res-env-ref>
    <jndi-info>
        <ref-name>jms/StockQueue</ref-name>
        <export-name>StockQueue</export-name>
    </jndi-info>
</res-env-ref>
    . . .
</jeus-web-dd>

```

3.4. 웹 컨텍스트 모니터링

모니터링은 웹 컨텍스트의 정보를 확인하는 것이다. 웹 엔진에 배치되어 있는 웹 컨텍스트의 목록 및 현재 상태 등을 콘솔 툴을 사용하여 조회할 수 있다.

콘솔 툴 사용

다음과 같이 **application-info** 명령어를 수행하면 deploy된 웹 컨텍스트의 모든 정보가 조회된다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "application-info"를 참고한다.

```
application-info -type WAR
```

특정 웹 컨텍스트의 정보를 조회하려면 다음과 같이 **-id <application-id>** 옵션을 설정하여 명령어를 수행한다. 해당 ID의 웹 컨텍스트 정보가 조회된다.

```
application-info -id <application-id>
```

3.5. 웹 컨텍스트 제어

웹 컨텍스트를 제어한다는 것은 웹 컨텍스트의 상태를 변경하여 서비스를 변경하는 것을 의미한다. 콘솔 툴을 사용하여 웹 컨텍스트를 리로드하거나 중지 및 재개하는 등의 제어가 가능하다.

3.5.1. 웹 컨텍스트 리로드

배치되어 있는 웹 컨텍스트가 변경이 있거나 특별한 이유로 다시 초기화해야 할 필요가 있을 경우에 웹 컨텍스트를 제거한 후 다시 배치하지 않고, 변경사항을 반영하여 웹 컨텍스트를 리로드한다. 콘솔 툴을 사용하여 웹 컨텍스트를 리로드할 수 있다.

콘솔 톨 사용

다음과 같이 **reload-web-context** 명령어를 수행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "reload-web-context"를 참고한다.

```
reload-web-context -ctx <context-name>
seccessfully reloaded
```

3.5.2. 웹 컨텍스트의 비동기 요청에 대한 Thread Pool 모니터링

웹 컨텍스트가 비동기 요청(Asynchronous Request)을 처리할 때 JEUS가 제공하는 Thread Pool을 사용하려면 jeus-web-dd.xml의 <async-config>에 관련 정보를 설정해야 한다. <async-config>를 설정한 후에 JEUS에서 제공하는 Async Thread Pool 정보를 모니터링할 수 있다.

다음은 jeus-web-dd.xml의 설정 예이다. 설정된 정보에 따른 Async Thread Pool 정보는 콘솔 명령을 통해 확인할 수 있다.

<async-config> 설정 : <jeus-web-dd.xml>

```
...
<async-config>
  <dispatch-thread-pool>
    <min>5</min>
    <max>20</max>
  </dispatch-thread-pool>
  <background-thread-pool>
    <min>5</min>
    <max>20</max>
  </background-thread-pool>
  <async-timeout-millis>120000</async-timeout-millis>
</async-config>
...
```

다음은 설정 태그에 대한 설명이다.

태그	설명
<dispatch-thread-pool>	AsyncContext#dispatch를 호출할 때 사용할 Thread Pool의 최소 Thread 값과 최대 Thread 값을 설정한다.
<background-thread-pool>	AsyncContext#start를 호출할 때 사용할 Thread Pool의 최소 Thread 값과 최대 Thread 값을 설정한다.
<async-timeout-millis>	Asynchronous 작업을 수행할 때 웹 컨테이너가 타임아웃을 처리하는 데 기준이 되는 시간을 설정한다. 애플리케이션이 AsyncContext#setTimeout을 호출하지 않은 경우 AsyncContext#getTimeout일 때 설정된 값을 리턴한다. (기본값: 30000, 단위: ms)

콘솔 툴 사용

콘솔을 사용한 웹 컨텍스트의 Thread 정보 조회 방법은 다음과 같이 `thread-info` 명령어를 수행한다. `thread-info` 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "thread-info"를 참고한다.

```
thread-info -server <server-name>
```

3.5.3. 웹 컨텍스트 중지

배치되어 서비스되고 있는 웹 컨텍스트를 관리자가 특정 시간이나 특별한 사유로 웹 엔진 내의 특정 웹 컨텍스트의 서비스를 잠깐 중지할 수 있다. 중지된 웹 컨텍스트를 다시 리로드할 수도 있다. 웹 컨텍스트 리로드 방법에 대한 자세한 내용은 [웹 컨텍스트 재개](#)를 참고한다.

콘솔 툴 사용

다음과 같이 **suspend-web-component** 명령어를 수행한다. `suspend-web-component` 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "suspend-web-component"를 참고한다.

```
suspend-web-component -server <server-name> -ctx <context> -svl <servlet>
```

또는

```
suspend-web-component -cluster <cluster-name> -ctx <context> -svl <servlet>
```

3.5.4. 웹 컨텍스트 재개

중지된 웹 컨텍스트를 콘솔 툴을 사용하여 다시 재개할 수 있다.

콘솔 툴 사용

콘솔 툴을 사용하여 중지된 웹 컨텍스트를 재개하려면 다음과 같이 **resume-web-component** 명령어를 수행한다. `resume-web-component` 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "resume-web-component"를 참고한다.

```
resume-web-component -server <server-name> -ctx <context> -svl <servlet>
```

또는

```
resume-web-component -cluster <cluster-name> -ctx <context> -svl <servlet>
```

3.6. 웹 컨텍스트 튜닝

최고의 성능을 위해 웹 컨텍스트의 설정(jeus-web-dd.xml)을 튜닝할 때 다음과 같은 사항들을 고려해야 한다.

- 유저 로그는 버퍼의 크기를 증가시키고 "stdout"의 사용을 자제한다.
- JSP 엔진은 사용하지 않으면 OFF시킨다.
- <enable-reload>와 <check-on-demand> 옵션은 항상 OFF시킨다. 이 옵션은 클래스 변경이 잦은 개발 환경에서만 사용한다.
- 가능하면 파일 Caching 기능을 사용한다. 최대한 많은 양의 정적 콘텐츠 디렉토리를 파일 Caching 태그에 설정한다. Cache의 <max-idle-time>과 <max-cache-memory>의 값을 크게 설정한다(무한대로 설정해도 무방하다).

3.7. 비동기식 처리 프로그래밍 가이드

본 절에서는 Servlet 3.0부터 추가된 비동기식 처리(Asynchronous Processing)에 관해서 애플리케이션 개발자가 반드시 숙지해야 할 내용에 대해 설명한다.

3.7.1. Servlet 표준에 따른 주의사항

애플리케이션 개발자가 직접 Thread를 다뤄야 하는 비동기식 처리 프로그래밍의 특성으로 인해 다음과 같은 실수가 발생할 수 있다.

- Thread의 과다 생성
- 서로 다른 Thread 간에 공유하면 안 되는 객체의 공유
- 비동기식으로 처리하는 Thread에서 jakarta.servlet.RequestDispatcher 사용

이 중에는 테스트할 때는 발견하지 못하다가 실제 서비스 중에 부하가 많이 발생했을 때 나타나는 문제들이 있다. 이는 JEUS에서 보장할 수 없는 문제들로 이에 대한 책임은 애플리케이션 개발자에게 있다.



본 절에서 설명하는 내용은 서블릿 표준을 기반으로 작성한 것이다(JEUS에만 해당하는 것이 아니다).

Thread의 과다 생성

Thread를 과도하게 많이 생성하면 JVM의 성능이 떨어지거나 OutOfMemoryError 상태에 빠진다. 다음은 Thread를 과도하게 많이 생성한 잘못된 비동기식 처리 프로그래밍의 작성 예이다.

잘못된 비동기식 처리 프로그래밍 작성 예 (1) : <WrongAsyncServlet1.java>

```
import jakarta.servlet.AsyncContext;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
```

```
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(urlPatterns = "/WrongAsyncServlet1", asyncSupported = true)
public class WrongAsyncServlet1 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        AsyncContext asyncContext = req.startAsync();
        Thread thread = new Thread(new TestRunnable(asyncContext));
        thread.start();
    }

    private class TestRunnable implements Runnable {
        private final AsyncContext asyncContext;

        public TestRunnable(AsyncContext asyncContext) {
            this.asyncContext = asyncContext;
        }

        @Override
        public void run() {
        }
    }
}
```

비동기식 처리가 필요한 요청은 서비스 상황에 따라서 동시에 수천 개가 될 수도 있다. 그때마다 새로운 Thread를 생성하면 동시에 수천 개의 Thread가 존재하게 된다. 이로 인해서 JVM의 성능이 심각하게 저하되거나 "OutOfMemoryError: unable to create new native thread"와 같은 에러가 발생할 수 있다. 따라서 가능하면 **jakarta.servlet.AsyncContext#start(Runnable)** 메소드를 사용하거나 직접 관리하는 Thread Pool을 사용할 것을 권장한다.

```
AsyncContext asyncContext = req.startAsync();
asyncContext.start(new TestRunnable(asyncContext));
```

서로 다른 Thread 간에 공유하면 안 되는 객체의 공유

서로 다른 Thread 간에 공유하면 안 되는 멀티 스레드에 안전하지 않은 객체를 함께 사용하면 성능 저하, 교착 상태 또는 데이터의 무결성을 침해할 수 있다.

다음은 서로 다른 Thread 간에 공유하면 안 되는 객체를 공유한 잘못된 비동기식 처리 프로그래밍의 작성 예이다.

잘못된 비동기식 처리 프로그래밍 작성 예 (2) : <WrongAsyncServlet2.java>

```
import jakarta.servlet.AsyncContext;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletInputStream;
import jakarta.servlet.ServletOutputStream;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
```



```

@WebServlet(urlPatterns = "/WrongAsyncServlet2", asyncSupported = true)
public class WrongAsyncServlet2 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        AsyncContext asyncContext = req.startAsync();
        asyncContext.start(new TestRunnable(asyncContext));

        byte[] buffer = new byte[1024];
        ServletInputStream inputStream = req.getInputStream();
        inputStream.read(buffer);

        ServletOutputStream outputStream = resp.getOutputStream();
        outputStream.write(buffer);
    }

    private class TestRunnable implements Runnable {
        private final AsyncContext asyncContext;

        public TestRunnable(AsyncContext asyncContext) {
            this.asyncContext = asyncContext;
        }

        @Override
        public void run() {
            byte[] buffer = new byte[1024];
            try {
                ServletInputStream inputStream = asyncContext.getRequest().getInputStream();
                inputStream.read(buffer);
            } catch (IOException e) {
                //log error
                return;
            }

            ServletOutputStream outputStream;
            try {
                outputStream = asyncContext.getResponse().getOutputStream();
                outputStream.write(buffer);
            } catch (IOException e) {
                //log error
                return;
            }
        }
    }
}

```

위의 예제는 Request 객체로부터 얻을 수 있는 ServletInputStream이나 Reader 객체 또는 Response 객체로부터 얻을 수 있는 ServletOutputStream이나 Writer 객체를 서로 다른 Thread에서 함께 사용하는 경우이다. Servlet 표준에 의거하여 JEUS는 이러한 동작에 대해 안정성을 보장하지 않는다. **필터 또는 서블릿에서 비동기식 처리를 시작해서 다른 Thread로 동작을 넘겼다면 그 이후 필터 및 서블릿 코드에서는 가능하면 모든 동작을 비동기식 처리를 수행하는 Thread 측에 맡겨야 한다.** 이는 멀티 스레드 프로그래밍에 대한 잘못된 이해로 인한 성능 저하, 교착 상태(Dead Lock)나 데이터 무결성 침해를 피하는 방법이기도 하다.

startAsync()로 직접 호출하지 않은 필터 또는 서블릿에서 비동기식 처리가 시작됐는지 확인하려면 **jakarta.servlet.ServletRequest#isAsyncStarted()**를 체크한다.

비동기식으로 처리하는 Thread에서 jakarta.servlet.RequestDispatcher 사용

비동기식으로 처리하는 Thread에서 jakarta.servlet.RequestDispatcher를 사용하면 원하는 대로 프로그램이 동작하지 않거나 Exception이 발생할 수 있다.

다음은 Thread에서 jakarta.servlet.RequestDispatcher를 사용한 잘못된 비동기식 처리 프로그래밍의 작성 예이다.

잘못된 비동기식 처리 프로그래밍 작성 예 (3) : <WrongAsyncServlet3.java>

```
import jakarta.servlet.AsyncContext;
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(urlPatterns = "/WrongAsyncServlet3", asyncSupported = true)
public class WrongAsyncServlet3 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        AsyncContext asyncContext = req.startAsync();
        asyncContext.start(new TestRunnable(asyncContext));
    }

    private class TestRunnable implements Runnable {
        private final AsyncContext asyncContext;

        public TestRunnable(AsyncContext asyncContext) {
            this.asyncContext = asyncContext;
        }

        @Override
        public void run() {
            // ..... do something
            RequestDispatcher requestDispatcher =
                asyncContext.getRequest().getRequestDispatcher("/views/report.jsp");
            try {
                requestDispatcher.forward(asyncContext.getRequest(),
                    asyncContext.getResponse());
            } catch (ServletException e) {
                // log error
            } catch (IOException e) {
                // log error
            }
        }
    }
}
```

비동기식 처리를 진행하는 Thread에서 다른 서블릿이나 JSP로 dispatch하기 위한 방법은 jakarta.servlet.AsyncContext에서 제공하고 있다.

```
asyncContext.dispatch("/views/report.jsp");
```

3.7.2. 기타 주의사항

비동기식 처리 프로그래밍에 대해 표준에서 설명하고 있는 내용 외에도 주의해야 할 사항이 존재한다.

- **AsyncContext.start(Runnable)를 호출했을 때 Thread에서 java.util.concurrent.RejectedExecutionException이 발생하는 경우**

AsyncContext.start는 새로운 태스크(Task)를 나타낸다. 이 태스크를 수행하기 위해서는 Thread가 필요한데, 일반적으로 Thread는 제한된 리소스이기 때문에 Thread Pool을 사용한다. 새로운 태스크를 Thread Pool에 할당했을 때 해당 태스크를 수행할 수 있는 Thread가 하나도 없다면 큐에 쌓이게 된다.

일반적으로 오랜 수행시간을 필요로 하는 태스크의 경우에 Thread의 처리가 지연되면 태스크들은 큐에 적체된다. 큐 역시 사이즈(기본값(최대값): 4096)가 제한되어 있으므로 결국에는 새로운 태스크를 할당할 수 없는 상태가 된다. 이러한 상황에 도달하면 **java.util.concurrent.RejectedExecutionException**이 발생한다. 그러므로 애플리케이션 프로그래머는 이러한 상태를 고려해서 비동기식 처리를 프로그래밍해야 한다.

- **WebtoB, Apache 등 웹 서버와 통신하는 방식의 리스너에서 비동기식 처리를 하는 경우**

WebtoB, AJP13 리스너는 제한된 수의 TCP 커넥션들을 사용한다. 웹 서버에서 WAS로 전달해야 하는 HTTP 요청이 한꺼번에 몰릴 경우 WAS의 처리 용량을 넘어서 결국 서비스 불능 상태가 될 수 있기 때문이다. 이러한 방식은 본래의 동기식 처리를 타깃으로 디자인된 것이다. 웹 서버로부터 HTTP 요청이 전달되면 WAS는 그 요청이 전달된 TCP 커넥션에 대한 소유권을 Servlet에게 넘기는데 해당 Servlet이 최대한 빠르게 HTTP 응답을 보내고 TCP 커넥션을 리턴할 거라고 가정하고 있다.

하지만 비동기식 처리는 Servlet이 빠르게 HTTP 응답을 보낸다고 가정할 수 없다. 만약 웹 서버와의 TCP 커넥션들을 모두 비동기식 처리에 사용하게 되는 경우 일반적인 HTTP 요청 처리를 일시적으로 하지 못하게 된다.

이러한 문제를 해결하기 위해서는 다음과 같은 방법들을 고려해볼 수 있다.

- 웹 서버에서 비동기식 요청에 대한 URL 매핑 등을 통해 Reverse Proxy로 처리한다. Reverse Proxy의 타깃은 HTTP 리스너이다.
- 웹 서버와의 TCP 커넥션 수를 충분히 지정해서 모든 커넥션이 비동기식 처리를 하는 상황이 없도록 만든다.



WebtoB는 4.1.6 이상을 사용하길 권장한다.

3.8. 변경된 예외 처리 정책 안내

본 절에서는 이전 버전과 비교하여 변경된 예외 처리 정책을 설명한다.

Servlet API 명세를 준수하기 위한 조치로, 기존에는 무시되던 일부 예외 상황에서 명시적으로 예외를 발생시키도록 변경되었다.

- 문자 인코딩 시 예외 발생

Servlet API 명세에 따라, `HttpServletRequest.getReader()` 및 `HttpServletResponse.getWriter()` 메서드 호출 시, 사용되는 문자 인코딩이 유효하지 않을 경우 `UnsupportedEncodingException`을 발생시킨다.

- **`ServletContext.getResource(String path)` 호출 시 예외 발생**

Servlet API 명세에 따라, 웹 애플리케이션 컨텍스트 내의 리소스 조회 시 유효하지 않은 인자가 전달되면 `MalformedURLException`을 발생시킨다.

4. 스레드 풀

본 장에서는 JEUS 웹 엔진에서 사용하는 스레드 풀의 구성 요소와 기본적인 설정 방법에 대해 설명한다.

4.1. 개요

웹 엔진에서 요청을 처리하기 위해서는 스레드 풀이 필수적이다.

스레드 풀은 Web-Connection, VirtualHost, Context 단에서 설정할 수 있으며, 우선순위가 존재하여 요청에 따라 각각 다른 스레드 풀에서 처리될 수 있다.

우선 3개의 스레드 풀을 다 활용하기 위해서는 Selector 기반으로 돌아가야 하며, 서버 리스너 설정이 있는 Ajp, Http, Tcp 리스너와 use-nio가 true인 WebtoB 커넥터가 이에 해당한다. use-nio가 false인 WebtoB 커넥터와 Tmax 커넥터는 오직 웹 커넥션 레벨의 스레드 풀에서만 처리된다.

Context와 VirtualHost 레벨에서의 스레드 풀은 선택사항이고, Web-Connection 레벨의 스레드 풀은 필수적으로 설정해야 한다. Context 스레드 풀은 jeus-web-dd.xml에서 설정이 가능하고, VirtualHost와 Web-Connection 스레드 풀은 domain.xml에서 각각 <virtual-host>와 <web-connections> 하위에서 설정이 가능하다.

[참고] JEUS 21 이전에는 Web Connection 별로 스레드 풀이 존재했었고, JEUS 21 FIX 0에서는 서버 리스너가 있는 Ajp, Http, Tcp 웹 커넥션은 WebContainer, VirtualHost, Context 레벨의 스레드 풀에서 처리될 수 있었고, WebtoB와 Tmax 커넥터는 자신에게 종속하는 스레드 풀에서 처리되었다.

4.2. 기본 구조

본 절에서는 WebContainer, VirtualHost, Context의 레벨의 스레드 풀 생성, 소멸 시점과 역할을 설명한다.

Web-Connection 레벨의 스레드 풀은 웹 엔진이 기동될 때 설정된 Web-Connection을 생성하거나 동적으로 Web-Connection을 추가할 때 생성되며, 조건에 맞는 Context와 VirtualHost의 스레드 풀이 없을 때 디폴트로 요청을 처리하는 스레드 풀이다. Web-Connection이 제거될 때 같이 스레드 풀도 소멸된다.

VirtualHost 레벨의 스레드 풀은 웹 엔진이 기동될 때 설정된 VirtualHost를 생성하거나 동적으로 VirtualHost를 추가할 때, 스레드 풀에 대한 설정이 존재하면 각 VirtualHost마다 스레드 풀을 생성한다. 참고로, System application을 위한 System Host와, 디폴트로 설정되는 Default Host는 스레드 풀을 생성하지 않는다. VirtualHost를 제거할 때 해당하는 스레드 풀이 소멸된다.

Context 레벨의 스레드 풀은 application이 deploy될 때 스레드 풀에 대한 설정이 존재하면 스레드 풀을 생성한다. Context 레벨의 스레드 풀은 다른 레벨의 스레드 풀과 달리 하나 이상의 스레드 풀을 설정할 수 있다. Context 레벨의 스레드 풀은 하나의 base 스레드 풀과 하나 이상의 Service Group 스레드 풀을 설정할 수 있다. Service Group 스레드 풀은 특정 uri로 온 요청들을 처리하기 위해 하나 이상의 uri를 ','로 구분하여 uri들을 설정할 수 있으며, 요청이 많이 들어올 리소스를 효율적으로 처리할 수 있도록 도와준다. uri에 매칭되는 Service Group 스레드 풀이 없을 경우에 base 스레드 풀이 설정되어 있다면, Service Group에 설정된 uri를 제외한 모든 요청들은 base 스레드 풀로 처리된다. 만약 base 스레드 풀도 설정되어 있지 않다면, VirtualHost 또는 디폴트인 Web-Connection 레벨의 스레드 풀로 요청을 처리한다. application이 undeploy될 때 생성된 모든 스레드 풀이 소멸된다.

4.3. 설정

본 절에서는 각 레벨의 스레드 풀 설정 방법을 설명한다.

Web-Connection 레벨의 스레드 풀은 domain.xml에 정의되어야 한다. 만약 설정하지 않는다면, JEUS 기동에 실패하게 된다.

Web-Connection 레벨의 스레드 풀 : <domain.xml>

```
<web-connections>
  <http-listener>
    <thread-pool>
      <min>10</min>
      <max>40</max>
      <max-idle-time>300000</max-idle-time>
      <max-queue>30</max-queue>
    </thread-pool>
  </http-listener>
</web-connections>
```

위의 예제에 나온 것 같이 Web-Connection 레벨의 스레드 풀은 <Web-Connections> 하위에 설정된 각 리스너 혹은 커넥터마다 하나의 스레드 풀을 설정할 수 있다. 선택된 Context와 VirtualHost 레벨의 스레드 풀이 없을 경우에 요청을 처리한다.

또한 jeusadmin에서 동적으로 추가할 때도 스레드 풀을 필수적으로 설정해야 하며, 자세한 것은 [콘솔 툴을 사용한 ajp-listener 추가](#)를 참고한다.

VirtualHost 레벨의 스레드 풀은 domain.xml에 정의되어야 한다. 선택적으로 설정할 수 있으며, 설정하지 않을 경우에는 Web-Connection 레벨의 스레드 풀에서 처리된다.

VirtualHost 레벨의 스레드 풀 : <domain.xml>

```
<web-engine>
  <virtual-host>
    <virtual-host-name>testHost</virtual-host-name>
    <host-name>192.1.1.1</host-name>
    <thread-pool>
      <min>10</min>
      <max>20</max>
      <max-idle-time>300000</max-idle-time>
      <max-queue>30</max-queue>
    </thread-pool>
  </virtual-host>
</web-engine>
```

위의 예제에 나온 것 같이 VirtualHost 레벨의 스레드 풀은 <virtual-host> 하위에 하나의 스레드 풀을 설정할 수 있다. 선택된 Context 레벨의 스레드 풀이 없을 경우에 요청을 처리하고, VirtualHost 레벨의 스레드 풀을 설정하지 않으면 Web-Connection 레벨의 스레드 풀로 넘긴다.

또한 jeusadmin에서 동적으로 추가할 때도 스레드 풀을 설정할 수 있으며, 자세한 것은 [가상 호스트 설정](#)을 참고한다.

Context 레벨의 스레드 풀은 jeus-web-dd.xml에 정의되어야 한다. 선택적으로 설정할 수 있으며, 설정하지 않을 경우에는 VirtualHost 레벨의 스레드 풀에서 처리된다.

Context 레벨의 스레드 풀 : <jeus-web-dd.xml>

```
<jeus-web-dd>
  <thread-pool>
    <base>
      <name>baseThreadPool</name>
      <min>30</min>
      <max>50</max>
      <max-idle-time>300000</max-idle-time>
      <max-queue>30</max-queue>
    </base>
    <service-group>
      <name>service1</name>
      <uri>/index.html,/index.jsp</uri>
      <min>10</min>
      <max>20</max>
      <max-idle-time>300000</max-idle-time>
      <max-queue>30</max-queue>
    </service-group>
    <service-group>
      <name>service2</name>
      <uri>/mostRequestedResource</uri>
      <min>10</min>
      <max>20</max>
      <max-idle-time>300000</max-idle-time>
      <max-queue>30</max-queue>
    </service-group>
  </thread-pool>
</jeus-web-dd>
```

위의 예제에 나온 것 같이 Context 레벨의 스레드 풀은 <jeus-web-dd> 하위에 하나의 base 스레드 풀과 하나 이상의 service-group 스레드 풀을 설정할 수 있다. 다수의 Context 스레드 풀을 구분하기 위해서 unique한 name을 지정해주어야 한다. 요청된 uri가 service-group에 설정된 uri와 매칭된다면 해당 service-group의 스레드 풀로 요청을 처리하고, 없을 경우에는 base 스레드 풀로 처리한다. base 스레드 풀이 설정되지 않았다면, VirtualHost 레벨의 스레드 풀로 넘긴다.

또한 jeusadmin에서 deploy할 때에도 단 하나의 Context 스레드 풀을 추가적으로 넣어줄 수가 있다. 자세한 것은 JEUS Reference 안내서의 "deploy-application"을 참고한다.

• Thread Pool 공통 설정

- Worker Thread Pool에 대한 설정이다.
- 다음은 세부 사항을 설정하는 하위 항목에 대한 설명이다.

항목	설명
Min	Pool에서 관리할 최소 Worker Thread의 개수이다.
Max	Pool에서 관리할 최대 Worker Thread의 개수이다.
Maximum Idle Time	Pool 내에 존재하던 Thread가 제거되기 전까지의 사용되고 있지 않은 시간을 지정한다.

항목	설명
Max Queue	Queue에 대기할 수 있는 요청의 최대 개수를 설정한다.
Thread State Notify	<p>블록되는 Worker Thread가 발생할 경우 이를 알려준다.</p> <p>각 Thread Pool은 장애가 발생했을 때 취하는 액션을 정의하는 'Thread State Notify' 항목을 가지고 있다. 이것에 대해서는 자동 Thread Pool 관리 설정(Thread 상태 통보)을 참고한다.</p>

4.4. 자동 Thread Pool 관리 설정(Thread 상태 통보)

Thread 상태 통보는 이메일 통지나 서버 재시작 권고를 위해 필요한 최소의 Worker Thread 수와 관련된 오류 조건을 정의한다.

다음은 콘솔에서 Thread 상태 통보를 설정하는 과정에 대한 설명이다. 설정이 완료되면 Thread Pool의 Thread 상태 변경에 대한 통보 설정을 통해 원하는 통보를 받을 수 있다.

자동 Thread Pool 관리 설정이 되면 '**Max Thread Active Time**' 설정값을 기준으로 Worker Thread가 관리된다. 요청을 받아 처리를 시작한 시점부터 Max Thread Active 설정 시간을 초과한 Thread들은 Blocked Thread로 관리가 된다. 그리고 Blocked Thread가 아닌 일반 Thread들의 숫자가 Thread Pool에서 설정한 최솟값보다 작을 경우는 새로운 Worker Thread를 생성하여 최솟값을 유지할 수 있게 한다.

이메일 알림자를 사용할 경우 이메일 알림자를 통해 받을 이메일의 설정은 JEUS Server 안내서의 "로깅 설정"에서 "SMTP 핸들러" 항목을 참고한다.

```
<web-engine>
  <web-connections>
    <http-listener>
      <thread-pool>
        ...
        <thread-state-notify>
          <max-thread-active-time>10000</max-thread-active-time>
          <interrupt-thread>false</interrupt-thread>
          <active-timeout-notification>false</active-timeout-notification>
          <notify-threshold-ratio>0.1</notify-threshold-ratio>
          <notify-subject>Notify-Subject</notify-subject>
          <restart-threshold-ratio>0.1</restart-threshold-ratio>
          <restart-subject>Restart-Subject</restart-subject>
        </thread-state-notify>
      </thread-pool>
    </http-listener>
  </web-connections>
</web-engine>
```

다음은 각 항목에 대한 설명이다.

항목	설명
Max Thread Active Time	<p>블록되기 전까지 Worker Thread가 사용될 수 있는 최대시간을 설정한다.</p> <p>이 시간은 Worker Thread가 클라이언트 요청을 서비스할 때부터 측정된다(서블릿이 수행될 때). 설정한 시간이 초과되었거나 또는 Thread가 자유롭게 되어 Thread Pool로 되돌아 갈 때 만료된다(Thread가 block되지 않는 상황). 설정값이 0이거나 음수이면 무시된다.</p>
Notify Threshold Ratio	<p>존재할 수 있는 Blocked Thread의 최대 비율을 설정한다. 전체 Thread 개수 대비 Blocked Thread의 비율이 초과되면, 오류 조건으로 결정되어 이메일 알림자를 통하여 이메일 통보가 전송된다.</p> <p>1보다 작은 소수점으로 설정하며, 0이거나 음수이면 무시된다.</p>
Restart Threshold Ratio	<p>존재할 수 있는 Blocked Thread의 최대 비율을 설정한다. 전체 Thread 개수 대비 Blocked Thread의 비율이 초과되면, 오류 조건으로 결정되어 이메일 알림자를 통하여 이메일 통보가 전송된다. 그 후에 서버 로그에 서버 재시작 권고 메시지가 기록된다.</p> <p>1보다 작은 소수점으로 설정하며, 0이거나 음수이면 이 설정이 무시된다.</p>
Notify Subject	경고 이메일을 전송할 때 사용된다.
Restart Subject	엔진 로그 및 통보할 때 사용하는 이메일에 사용된다.
Interrupt Thread	<p>Active Timeout이 발생할 때, 즉 Worker Thread가 'Max Thread Active Time' 설정값 이상으로 요청을 처리 중일 때 해당 Thread의 interrupt 발생 유무를 결정한다.</p> <ul style="list-style-type: none"> ◦ true : interrupt를 발생시킨다. ◦ false : interrupt를 발생시키지 않는다. (기본값)
Active Timeout Notification	<p>Active Timeout 발생이 이메일 전송 여부를 결정한다.</p> <ul style="list-style-type: none"> ◦ true : 이메일을 전송한다. ◦ false : 이메일을 전송하지 않는다. (기본값)



이메일을 받기 위해 SMTP 핸들러를 설정할 로거는 이름을 "jeus.servlet.threadpool.notify"로 하여 따로 추가하길 권장한다.

4.5. 규칙

스레드 풀을 구성할 때 적용되는 규칙은 다음과 같다.

- Context, VirtualHost 레벨의 스레드 풀을 사용하기 위해서는 스레드 풀로 작업을 넘기기 전에 요청을 읽고 Context와 Host를 판단할 수 있어야 한다.

이에 해당하는 Web-Connections는 서버 리스너가 설정 가능한 Http, Ajp13, Tcp 리스너와 use-nio가 true인 WebtoB 커넥터이다. use-nio가 false인 WebtoB 커넥터와 Tmax 커넥터는 Web-Connection 스레드풀만 사용한다.

- WebtoB 설정할 때 주의점

WebtoB 커넥터는 use-nio에 설정에 따라 스레드 풀 설정에 주의해야 한다. use-nio가 true인 경우에는 connection 개수와 스레드 사이에는 더 이상 아무 관련이 없기 때문에, 튜닝에 맞게 적절한 값을 넣으면 되지만, use-nio가 false인 경우에는 WebtoB 커넥터의 Connection-Count와 thread-pool의 min, max 값이 동일해야 한다. 동일하지 않을 경우에는 서버 기동에 실패하게 된다.

- Service Group 스레드 풀을 설정할 때 똑같은 uri를 서로 다른 Service Group에 정의하지 않도록 주의해야 한다. uri는 '/'로 시작해야 하며, 서블릿 표준에서 Context path 다음으로 오는 Servlet path와 path Info에 해당한다.
- uri를 정의할 때는 하위의 경로도 포함할 수 있다는 것을 고려해야 한다. 예를 들어 Service Group의 uri에 "/myServlet/"으로 지정하였다면, /testContext/myServlet/servlet1과 /testContext/myServlet/servlet2 요청 모두 "/myServlet/"으로 설정한 Service Group으로 처리한다.
- 다수의 Service Group이 정의되어 있을 때는 가장 많이 매칭되는 uri를 선택하도록 한다.

예를 들어, Service Group1의 uri에 "/myServlet/"으로 지정하였고, Service Group2의 uri에 "/myServlet/servlet2"으로 지정하였다면, /testContext/myServlet/servlet1 요청은 Service Group1로 /testContext/myServlet/servlet2 요청은 Service Group2로 처리한다.

- 요청을 어떤 스레드 풀로 처리될지 선택하는 과정은 간단하다.
 1. 요청이 들어온 Web-Connection이 Context와 VirtualHost 스레드 풀을 사용할 수 있는지 판단한다. 해당 스레드풀을 사용할 수 없는 Web-Connection이라면 Web-Connection 스레드 풀로 처리한다.
 2. 요청에 해당하는 Context가 스레드 풀들을 가지고 있는지 확인한다. Service Group 스레드 풀이 설정되어 있다면, uri를 비교해서 맞으면 해당 스레드 풀로 처리하고, 아니면 다음 스레드 풀을 찾는다.
 3. Context 스레드 풀에 base 스레드 풀이 설정되어 있다면 base 스레드 풀로 처리한다.
 4. 요청에 해당하는 Virtual Host에 스레드 풀이 설정되어 있다면 해당 스레드풀로 처리한다.
 5. 4번까지 해당하는 스레드 풀이 없다면 Web-Connection 스레드 풀로 요청을 처리한다.

5. JSP 엔진

본 장에서는 JSP 엔진의 개념과 기능, 설정 방법에 대해 설명한다.

5.1. 개요

JSP 엔진은 웹 엔진에 내부적으로 포함되어 있는 형태이며 웹 컨텍스트마다 하나씩 존재한다. 본 장에서는 JSP 표준에 대한 설명은 포함하지 않는다. JSP 작성 등에 대한 자세한 정보는 JSP 표준을 참고한다.

JSP 엔진은 웹 클라이언트가 JSP 페이지를 요청했을 때 해당 페이지를 찾아서 서블릿으로 전환하는 역할을 한다. 서블릿으로 전환하는 과정에서 Java 파일과 SMAP 파일을 생성하고, Java 파일을 컴파일해서 서비스에 이용할 클래스 파일을 생성한다.

JSP 프리컴파일 기능을 사용하지 않았다면 JSP 컴파일은 최초 요청 시점에 수행한다. 따라서 최초 요청의 경우에는 OS 파일 시스템에 접근하는 일이 빈번하게 발생하기 때문에 응답시간이 늦을 수 있다.

NAS(Network Attached Storage)를 사용하는 환경에서 태그와 JSP include 관계때문에 컴파일해야 할 파일이 많은 경우에는 NAS로 많은 요청이 집중되서 응답시간이 지체되는 현상이 발생할 수 있다. 또한 NAS 드라이버 동작에 따라 java.io.IOException가 발생하지 않고 사이즈가 0인 JSP 파일을 읽는 경우도 발생한다. Jasper에서는 JVM File I/O API를 통해서 파일을 읽기 때문에 이런 경우 JSP 파일에 대한 파싱이 실패할 수 밖에 없다.

웹 컨텍스트 내에서의 JSP

JSP 파일은 웹 컨텍스트의 루트 아래에 존재한다. 별도로 디렉터리를 생성해서 패키징할 수도 있고 Servlet 3.0부터 정의한 META-INF/resources/ 디렉터리 아래에 패키징할 수도 있다.

WEB-INF/lib/ 아래의 *.jar 라이브러리 파일들 안에 META-INF/resources/ 디렉터리가 있는 경우에도 JSP 파일을 찾을 수 있다. 단, WEB-INF/ 디렉터리 아래에 있는 JSP 파일은 서비스되지 않는다. 웹 클라이언트가 WEB-INF/ 아래의 파일들에 보안상 접근할 수 없다. 웹 컨텍스트 내부 구조에 대한 자세한 내용은 [웹 컨텍스트 내부 구조\(WAR 파일 구조\)](#)를 참고한다.



META-INF/resources/에 대한 자세한 내용은 [Jakarta ServletContext API 문서](#)의 리소스 관련 API 설명을 참고한다.

[주의]

내부 테스트 결과 Oracle JDK 6에서 제공하는 javac 라이브러리가 thread-safe하지 않다. 이에 따라 jeus.servlet.jsp.compile-java-source-concurrently 프로퍼티를 제공한다(기본값: false). 요청 스레드가 .java 파일을 컴파일할 때는 JVM Scope의 Lock을 잡고 수행한다. 만약 JDK에서 제공하는 javac 라이브러리가 thread-safe한 경우에는 다음과 같이 jeus-web-dd.xml에 true로 설정해서 사용해도 된다. 단, 반드시 단일 스레드가 아닌 멀티 스레드 상황에서 JSP 컴파일이 정상적으로 되는지 테스트해야 한다.

```
<properties>
  <property>
    <key>jeus.servlet.jsp.compile-java-source-concurrently</key>
    <value>true</value>
```

```
</property>
</properties>
```

5.2. Apache Tomcat Jasper

JEUS는 기존부터 Tomcat의 JSP Parser인 Japser를 도입해서 사용했으나 패키지 이름을 변경하고 이를 jeus.jar에 포함하여 제공하였다. JEUS 9은 Tomcat 8 기반의 Jasper를 사용하며 org.apache.jasper 패키지의 이름을 그대로 유지하고 별도의 jasper.jar 라이브러리로 패키징하여 제공한다. 이 라이브러리는 다음 경로에 위치한다.

```
$JEUS_HOME/lib/system/jasper.jar
```

Jasper를 사용할 때는 다음 사항에 주의한다.

- jasper.jar는 일부 JEUS에 맞춰서 수정한 것이므로 Tomcat의 것으로 덮어쓰면 안 된다. 이 경우 서버 기동이 실패한다.
- 웹 애플리케이션에서 WEB-INF/lib 내에 Tomcat의 jasper.jar를 사용하는 경우 jeus-web-dd.xml의 <webinf-first> 옵션을 true로 설정해야 한다. 그렇지 않으면 JEUS에서 사용하는 jasper.jar의 클래스를 사용하게 되어 기대하는 동작과 다를 수 있다.

Tomcat Jasper와 JEUS Jasper의 차이점

위에서 언급한 바와 같이 JEUS는 Tomcat에서 제공하는 Japser를 일부 수정해서 제공한다. 그러므로 사용할 때 다음의 사항을 고려해야 한다.

- JEUS는 In-memory JSP Compilation 기능과 같이 Tomcat Jasper에서 제공하지 않는 기능을 제공한다.
- JEUS에서는 Tag Handler Pool, PageContext Pool을 사용하지 않는다.

Java 소스 컴파일할 때 64KB 메소드 크기 제한 문제

JSP를 작성할 때 하나의 .jsp 파일 내에 내용이 너무 많은 경우 이를 .java 파일로 생성하면 내부 메소드 크기가 64KB를 초과할 수 있다. 이 경우 해당 .java 파일은 Java Language 표준 규약을 어긴 것이므로 Java 컴파일러로 컴파일되지 않는다. 그러나 .jsp의 내용 중 SQL 문이나 HTML 태그와 같은 문자열이 대부분을 차지하는 경우에는 64KB 제한을 피해갈 수 있다.

애플리케이션이 가지고 있는 web.xml에 다음과 같이 설정한다.

```
<servlet>
  <servlet-name>jeus.servlet.servlets.JspServlet</servlet-name>
  <servlet-class>jeus.servlet.servlets.JspServlet</servlet-class>
  <init-param>
    <param-name>genStringAsCharArray</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>jeus.servlet.servlets.JspServlet</servlet-name>
```

```
<url-pattern>*.jsp</url-pattern>
</servlet-mapping>
```

문자열이 아니라 실제로 Java 코드량이 많은 경우에는 컴파일이 실패할 수 밖에 없다. 이때는 `<jsp:include>`를 사용하여 응답 내용을 분리해서 구현한다. 다음은 `<jsp:include>`를 사용한 예이다.

```
<body>
  Template Page<br/>
  <jsp:include page="module_one.jsp" />
  <jsp:include page="module_two.jsp" />
</body>
```

5.3. JSP 엔진 기능

JSP 엔진은 Graceful Reloading, 프리컴파일, 메모리에서의 JSP 컴파일 및 실행 기능 등을 제공한다.

5.3.1. JSP Graceful Reloading

JSP 파일은 사용 목적이 비즈니스 로직보다는 사용자 뷰에 가깝기 때문에 실제 서비스 운영 중에도 수정할 수 있기를 원하는 경우가 대부분이다. 이런 이유로 JSP를 많이 사용하는 웹 애플리케이션의 경우 WAR 형태보다는 디렉터리 형태로 deploy한다.

- 디렉터리 형태로 deploy한 경우

디렉터리로 deploy한 상태에서 해당 디렉터리 아래의 JSP 파일을 수정하면 JSP 엔진에서 기존에 컴파일된 클래스 파일의 마지막 수정 시각과 원본 JSP 파일의 수정 시각을 비교해서 컴파일을 수행한다.

이때 `<check-included-jspfile>` 설정이 true인 경우에는 요청한 JSP 파일이 변경되지 않았더라도, 해당 JSP 파일이 include한 JSP 파일들이나 TAG 파일들(.tld)을 체크해서 JSP 파일을 재컴파일한다.

JSP 엔진은 Graceful Reloading을 지원하기 때문에 위와 같이 JSP 파일 수정으로 인해 재컴파일이 발생하더라도 기존의 컴파일된 것으로 지속적으로 서비스가 가능하다.

- WAR 형태로 deploy한 경우

WAR 형태로 deploy할 경우 WAR 파일 전체를 리패키징해서 redeploy를 해야 하기 때문에 서비스 운영 측면에서 리스크가 큰 편이다. 하지만 변경하는 JSP 파일이 많은 경우에는 JEUS에서 제공하는 Graceful Redeploy를 고려할 수 있다. Graceful Redeploy에 관한 자세한 사항은 [웹 컨텍스트 Redeploy\(Graceful Redeploy\)](#)를 참고한다.



JSP 파일 내에서 `System.LoadLibrary()`를 사용해서 Native Library를 사용할 경우 JSP 리로딩이 실패할 수 있다. JSP 리로딩을 하기 위해서는 JVM 구조상 각 JSP별로 클래스 로더를 생성해야 하며, 리로딩할 때 클래스 로더를 교체해야 한다. Native Library에 관한 레퍼런스는 JVM이 클래스 로더에서 관리하며 클래스 로더 인스턴스가 GC(Garbage Collection)될 때 해당 레퍼런스를 정리한다. 그렇기 때문에 JSP의 클래스 로더를 교체하는 시점에 맞춰서 JVM GC가 발생하지 않으면 JSP 리로딩이 실패하게 된다.

JEUS에서는 JVM GC를 컨트롤할 수 없고, JVM 내부적인 구조에 따라 발생하는 문제이므로, 이를 해결할 방법이 없다. 따라서 Native Library 로딩 작업은 부팅할 때에 한 번만 하도록 구현하는 것이 바람직하다. 그리고 Native Library를 변경하는 경우에는 JVM 내부적으로 라이브러리 리로딩이 된다는 보장을 JEUS에서 할 수 없다.

5.3.2. JSP 프리컴파일

JSP 프리컴파일은 웹 컨텍스트의 JSP 페이지들을 미리 컴파일할 수 있는 기능이다. 이는 **appcompiler** 스크립트나 콘솔 툴의 **precompile-jsp**(약어 jspc) 명령어를 통해서 가능하다. appcompiler는 오프라인 상태에서도 JSP 프리컴파일이 가능하고, precompile-jsp는 JEUS에 deploy된 모듈에 대해서 프리컴파일을 수행할 수 있다. 이 기능을 사용하면 JSP가 처음 요청되었을 때의 성능을 향상시킬 수 있다. appcompiler나 콘솔 툴의 precompile-jsp 명령어에 대한 자세한 내용은 각각 JEUS Reference 안내서의 "appcompiler"와 "precompile-jsp"를 참고한다.

5.3.3. 메모리에서의 JSP 컴파일 및 실행 기능

NAS를 사용해서 여러 개의 웹 엔진이 이를 공유해서 하나의 소스로 동일한 서비스를 수행하는 경우 JSP 컴파일 시점에 필요한 OS 파일 시스템 접근 작업으로 인해 서비스 지연 현상 또는 에러가 발생할 수 있다. 이러한 문제점을 해결하기 위해서 JSP 프리컴파일, JSP Graceful Reloading 기능을 제공하지만 이를 좀 더 근본적으로 해결하기 위해서 메모리에서의 JSP 컴파일 및 실행 기능을 제공한다.

JSP 엔진은 요청 처리 스레드에서 JSP 컴파일의 결과물인 .java 파일 및 .class 파일을 저장하지 않고 모두 메모리에 두고 사용한다. 따라서 .jsp 파일의 메타 데이터와 .jsp 파일의 콘텐츠를 읽는 작업 이외에는 파일 시스템에 접근하지 않는다. 이렇게 파일 시스템 접근을 최소화하여 JSP 컴파일 타임에 발생할 수 있는 서비스 지연을 최소화하였다.

그러나 .java 파일 및 .class 파일은 추후 다른 용도를 위해서 필요하다. 이 파일들은 요청 처리 스레드가 아닌 JSP 엔진마다 하나씩 가지고 있는 Background Thread를 이용해서 파일 시스템에 사용한다. 순차적으로 처리하기 때문에 파일 I/O가 파일 시스템에 한 번에 몰릴 확률이 낮다. 하지만 .smap 파일은 사용하지 않는다. 만약 .smap 파일이 필요한 경우에는 기존의 JSP 컴파일 방식을 사용해야 한다.

JSP 엔진은 이 기능을 기본적으로 사용한다. 만약 기존과 같은 방식을 원하는 경우에는 jeus-web-dd.xml에 설정할 수 있다. 자세한 설정 사항은 [jeus-web-dd.xml 설정](#)을 참고한다.

5.4. JSP 엔진 설정

JSP 엔진은 domain.xml을 사용하거나 각 웹 애플리케이션의 jeus-web-dd.xml에 설정할 수 있다.

5.4.1. 웹 엔진 레벨에서 설정

다음은 domain.xml에서 웹 엔진의 JSP 엔진 설정하는 과정에 대한 설명이다.

```
<web-engine>
  <jsp-engine>
    <check-included-jspfile>true</check-included-jspfile>
    <keep-generated>true</keep-generated>
```

```

<use-in-memory-compilation>true</use-in-memory-compilation>
<graceful-jsp-reloading>false</graceful-jsp-reloading>
<graceful-jsp-reloading-period>30000</graceful-jsp-reloading-period>
<jsp-work-dir>...</jsp-work-dir>
<java-compiler>java6</java-compiler>
<compile-output-dir>...</compile-output-dir>
<compile-option>...</compile-option>
</jsp-engine>
</web-engine>

```

다음은 설정 항목에 대한 설명이다.

항목	설명
Jsp Work Dir	<p>JSP에서 생성된 Java 소스 파일들이 저장되는 루트 디렉터리를 지정한다. 설정했다고 루트 디렉터리에 바로 파일을 생성하는 것은 아니다. JSP 엔진이 속한 도메인, 서버 이름, 그리고 웹 애플리케이션 이름으로 디렉터리를 생성한 후 그 아래에 파일을 생성한다. 즉, 서로 다른 웹 엔진 간에 클래스 파일들을 공유하지 않는다.</p> <p>'Compile Output Dir' 항목을 설정하지 않은 경우 클래스 파일도 동일한 위치에 생성된다. 기본적으로 다음과 같은 위치에 생성된다.</p> <ul style="list-style-type: none"> ◦ EAR 애플리케이션 <div>INTERNALGENERATED_HOME/ear1/web1/__jsp_work</div> ◦ Standalone 웹 모듈 <div>INTERNALGENERATED_HOME/web1/__jsp_work</div>
Java Compiler	<p>Java Compiler 실행 명령어이다. JSP의 생성된 Java 소스를 서블릿 클래스로 컴파일할 컴파일러를 지정한다.</p> <p>기본 설정이 가장 효율적이기 때문에 사용하지 않는 것을 권장한다.</p>
Compile Output Dir	<p>JSP Parser가 생성한 Java 파일을 컴파일한 클래스 파일의 위치이다. 이 클래스 파일을 실제로 서비스에 사용한다.</p> <p>설정하지 않는 경우 Java 파일과 동일한 위치에 생성된다.</p>
Compile Option	<p>Java 컴파일러 실행 옵션이다.</p>
Check Included Jspfile	<p>JSP 엔진은 기본적으로 요청한 JSP 페이지의 변경 여부뿐만 아니라 <%@ include file="xxx.jsp" %> directive로 include된 모든 JSP 파일들 및 태그 파일들에 대해 변경되었는지 확인해서 컴파일한다.</p> <p>만약 false로 설정하면 요청한 JSP 페이지의 변경 여부만 확인하여 컴파일한다.</p>
Keep Generated	<p>JSP Parser가 생성한 Java 파일 및 SMAP 파일을 유지하는 옵션이다.</p> <p>디버깅할 때 유용하고, false로 설정하면 파일을 생성한 후 삭제하는 것이기 때문에 특별한 이유가 없다면 성능을 위해 별도의 설정을 하지 않는 것을 권장한다.</p>

항목	설명
Graceful Jsp Reloading	JSP 파일이 변경된 경우 지정된 주기마다 이를 감지하여 JSP 페이지 인스턴스를 새로 생성한다.
Graceful Jsp Reloading Period	Graceful Jsp Reloading이 동작되는 주기를 설정한다. (단위: ms)
Use In Memory Compilation	서비스 중인 JSP 파일을 새로 컴파일해야 할 때 .java 및 .class 파일을 메모리에 생성해서 컴파일하고 이를 실행하는 기능이다. 자세한 사항은 메모리에서의 JSP 컴파일 및 실행 기능 을 참고한다.

5.4.2. jeus-web-dd.xml 설정

JSP 엔진 설정은 jeus-web-dd.xml에서도 가능하다.

웹 컨텍스트 설정 파일 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
  <enable-jsp>true</enable-jsp>
  <jsp-engine>
    <jsp-work-dir>/home/user1/myjspwork/</jsp-work-dir>
    <java-compiler>java6</java-compiler>
    <compile-option>-g:none -verbose</compile-option>
    <compile-encoding>UTF-8</compile-encoding>
    <check-included-jspfile>true</check-included-jspfile>
    <keep-generated>true</keep-generated>
    <use-in-memory-compilation>true</use-in-memory-compilation>
  </jsp-engine>
</jeus-web-dd>
```

다음은 설정 태그에 대한 설명이다.

태그	설명
<enable-jsp>	false로 설정하면 JSP 기능을 사용하지 않으며, JEUS default servlet에 의해 jsp 파일 내용으로 서비스한다.
<jsp-engine>	웹 컨텍스트에 포함된 JSP 페이지에 대한 설정이다. 만약 jeus-web-dd.xml 파일에 JSP 엔진이 설정되어 있다면, 여기에 설정된 내용이 domain.xml에 설정된 것보다 우선한다. domain.xml에서 JSP 엔진 설정에 대한 자세한 내용은 웹 엔진 레벨에서 설정 을 참고한다.

5.4.3. JSP 하위 호환성을 위한 웹 컨텍스트 레벨의 옵션 설정

JEUS 4 및 5에서는 사용자의 편의성과 Servlet 2.3 이전에 개발된 애플리케이션을 위해서 표준이 아닌 기능도 제공하고, JSP 문법 체크도 최신 스펙에 비해서 엄격하지 않게 적용하였다. 하지만 JEUS 6부터는 좀 더 엄격한 문법 체크와 JSP 2.1 기능을 제대로 지원하기 위해 Jasper 기반의 JSP Parser로 교체하였다. 하지만 기존 JSP에 대한 하위 호환성을 위해서 기존의 JSP Parser도 지원한다.

따라서 JEUS 4 및 5에서는 문제가 없던 웹 모듈을 deploy하면 여러 가지 에러가 발생할 수 있다. JSP 2.1 등의 최신

스펙을 사용하려면 JSP 컴파일할 때 발생하는 에러 메시지를 확인 후 수정해서 업그레이드해야 한다.

만약 최신 스펙이 필요하지 않고 기존에 개발된 모듈을 그대로 사용하려면 다음과 같이 jeus-web-dd.xml에 JEUS 4 및 5 호환의 JSP Parser가 해당 웹 컨텍스트에만 적용되도록 설정할 수 있다.

JSP 하위 호환성을 위한 웹 컨텍스트 설정 : <jeus-web-dd.xml>

```
<properties>
  <property>
    <key>jeus.servlet.jsp.modern</key>
    <value>false</value>
  </property>
</properties>
```

jeus-web-dd.xml에 설정한 옵션은 해당 웹 컨텍스트에만 적용되고, 웹 엔진이나 가상 호스트 단위로도 옵션을 설정할 수 있다. 웹 엔진 레벨에서 옵션을 적용하려면 다음과 같은 VM 옵션을 설정한다. VM 옵션의 설정은 JEUS Server 안내서의 "서버 추가"를 참고한다.

```
-Djeus.servlet.jsp.modern=false
```

VM 옵션은 가상 호스트, 웹 컨텍스트 레벨에서 치환이 가능하며 jeus-web-dd.xml에 설정한 옵션이 최종적으로 적용된다. jeus-web-dd.xml에 옵션 설정이 없다면 상위의 기본 설정이 적용된다.



이 옵션의 사용을 절대로 권장하지 않으며 부득이하게 하위 호환성이 필요한 경우만 적용하고, 신규 개발은 새로운 애플리케이션으로 표준을 준수하여 개발할 것을 권장한다. **이 호환성 옵션은 차기 버전이나 다음 Fix에서 삭제될 수 있다.** 나머지 옵션에 대한 자세한 내용은 JEUS Reference 안내서의 "웹 엔진 프로퍼티"를 참고한다.

5.4.4. jarscan.properties 파일 설정

애플리케이션이 deploy될 때 classpath의 모든 .jar에서 .tld 파일을 스캔한다. 애플리케이션이 사용하는 .jar 형태의 라이브러리 중에서 .tld 파일이 포함되어 있지 않는 것을 알고 있을 경우 JEUS_HOME/domains/<domain-name>/config에 있는 jarscan.properties 파일에 해당 .jar를 추가하여 스캔하지 않도록 할 수 있다. 이를 통해서 deploy 시간을 단축시킬 수 있다.

애플리케이션이 deploy될 때 JEUS의 시스템 라이브러리가 classpath에 포함되므로 jarscan.properties 파일에는 .tld 파일을 포함하지 않는 시스템 라이브러리가 기본적으로 설정되어 있다.

불필요한 tld 스캔을 막기 위한 jarscan.properties 파일 설정 : <jarscan.properties>

```
jeus.servlet.jsp.JarScanFilterImpl.skipSet= \
  lib/system/EJML-core-0.29.jar, lib/system/EJML-dense64-0.29.jar, \
  ...
  lib/shared/wsit-2.3/webservices-rt-2.3.1.jar, lib/shared/wsit-2.3/webservices-tools-2.3.1.jar, \
  WEB-INF/lib/noTldLib.jar
```

6. 가상 호스트

본 장에서는 가상 호스트의 사용 목적, 규칙 및 설정 방법 등에 대해 설명한다.

6.1. 개요

가상 호스트는 인터넷 도메인 이름을 기준으로 같은 URL로 서로 다른 웹 애플리케이션에 매핑할 수 있도록 한다. 즉, 2개 이상의 도메인 이름(예: “www1.foo.com” and “www2.foo.com”)을 하나의 웹 엔진에 설정하여 서로 다른 웹 컨텍스트를 서비스할 수 있다.



웹 엔진 관점에서 웹 컨텍스트는 웹 애플리케이션과 동일한 의미이다.

6.2. 웹 엔진과 가상 호스트

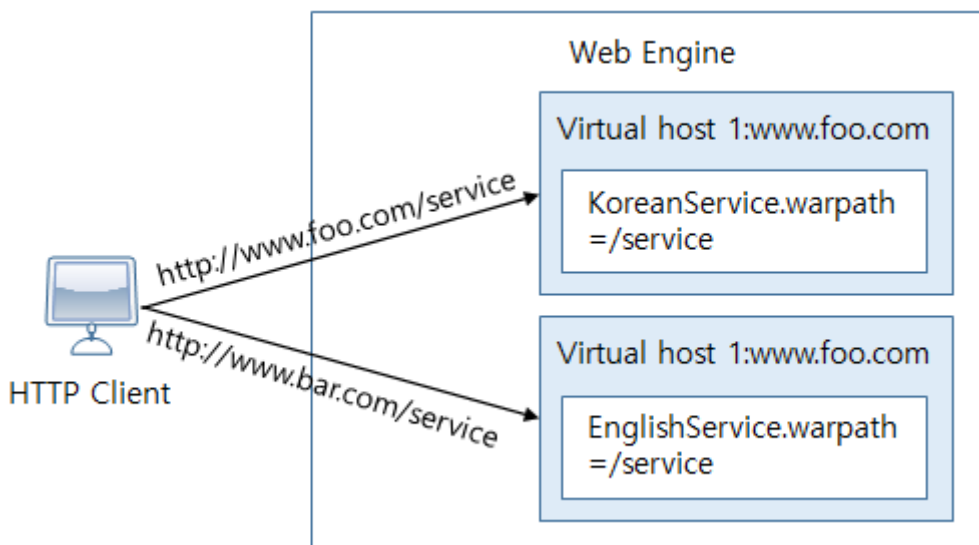
본 절에서는 가상 호스트의 사용 목적, 규칙, ServletContext 객체와 가상 호스트의 관계에 대해 설명한다.

사용 목적

가상 호스트에 매핑된 도메인 이름을 기준으로, 같은 URL로 서로 다른 웹 애플리케이션에 매핑할 수 있다. 따라서 서비스 제공자는 하나의 웹 엔진으로 2개 이상의 웹 사이트를 서비스 이용자에게 제공할 수 있다. 이는 HTTP 1.1의 호스트 헤더를 이용해서 가상 호스트를 제공하는 기능과 동일하다.

가상 호스트는 웹 엔진에 설정할 수 있는 일종의 웹 컨텍스트 그룹이다. 가상 호스트가 웹 엔진의 구성 요소로서 어떻게 위치하는지는 [웹 엔진의 구성 요소](#)를 참고한다.

다음은 가상 호스트의 사용 목적에 따른 이용 패턴을 보여준다.



가상 호스트의 이용 패턴

위의 예제를 보면, 서로 다른 2개의 주소로 서로 같은 컨텍스트 패스(/service)에 접근할 수 있다. 실제로는 하나의

서버뿐이지만 HTTP 클라이언트 입장에서는 "www.foo.com"과 "www.bar.com"이라는 2대의 서버가 존재하는 것처럼 인식된다.

서비스 제공자 입장에서는 "/service"라는 동일한 주소 패턴으로 서로 다른 서비스를 제공할 수 있다. 위의 예제에서는 "www.foo.com"은 한국어 서비스를, "www.bar.com"은 영어 서비스를 제공하고 있다.

규칙

가상 호스트를 구성할 때 적용되는 규칙은 다음과 같다.

- 가상 호스트에는 가상 호스트 이름을 부여한다.

가상 호스트의 이름은 설정 파일 내에서 가상 호스트를 참조하기 위해서 내부적으로 사용되는 이름으로 웹 엔진 내에서 유일해야 한다.

- 하나의 가상 호스트는 1개 이상의 도메인 이름이나 IP 주소를 매핑할 수 있다.

JEUS는 이를 호스트 이름이라고 한다. 서로 다른 가상 호스트에 같은 호스트 이름을 매핑할 수 없다는 점에 유의한다.

- 동일한 이름의 웹 컨텍스트는 서로 다른 가상 호스트에 deploy할 수 없다.



서블릿 표준에서 서로 다른 가상 호스트에서 동일한 웹 컨텍스트를 공유할 수 없다고 정의되어 있다.

- 동일한 패스를 가진 서로 다른 웹 컨텍스트를 각각 서로 다른 가상 호스트에 deploy할 수 있다. 단, 하나의 가상 호스트 내에서는 동일한 패스를 가진 2개 이상의 웹 컨텍스트는 존재할 수 없다.

웹 컨텍스트 이름은 Jakarta EE 표준에서 정의한 애플리케이션 또는 모듈 이름을 의미한다. 패스는 웹 애플리케이션 내에서 정의하는 Context Root 또는 Context Path를 의미한다. 웹 컨텍스트 이름은 JEUS Deploy 차원에서 관리하는 것이며, 웹 엔진은 Context Path를 관리한다.

JEUS 웹 엔진에는 기본 가상 호스트(Default Virtual Host)라는 묵시적인 가상 호스트가 존재한다. 웹 컨텍스트를 deploy할 때 명시적으로 가상 호스트에 지정하지 않으면, 기본 가상 호스트로 deploy한다. 이 가상 호스트의 이름은 "DEFAULT_HOME"이다. 예약어이므로 다른 가상 호스트 이름으로 지정할 수 없다.

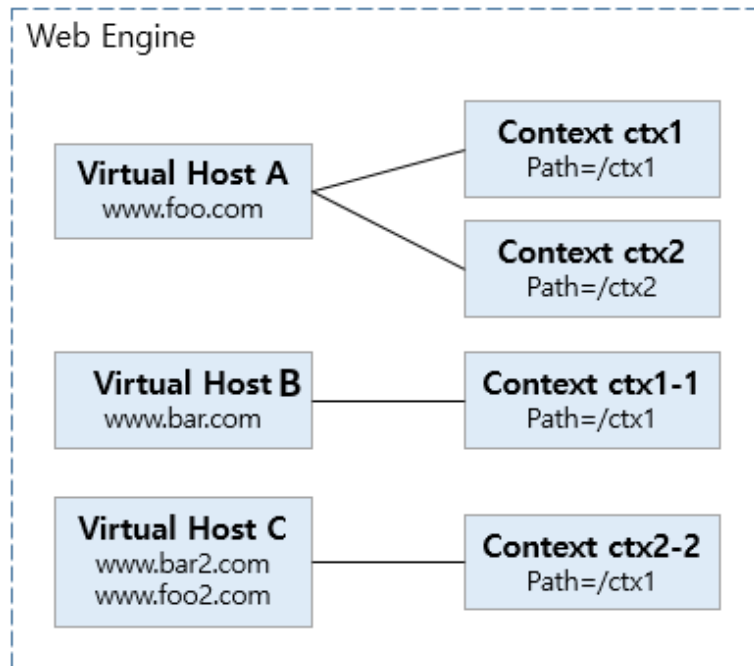
6.2.1. ServletContext 객체와 가상 호스트

Servlet API에는 `jakarta.servlet.ServletContext.getContext(String contextPath)`라는 메소드가 있다. "contextPath"에 의해 주어진 ServletContext 객체를 리턴한다. 이 메소드는 ServletContext가 속하는 가상 호스트에 존재하는 ServletContext를 리턴한다. 만약 해당 가상 호스트 내에 없으면 기본 가상 호스트에서 찾는다.

6.3. 가상 호스트를 통한 웹 컨텍스트 요청

본 절에서는 URL과 가상 호스트 내에 존재하는 웹 컨텍스트를 매칭하는 방법에 대해 설명한다.

다음은 웹 엔진과 가상 호스트, 웹 컨텍스트 간의 유효한 관계의 예시를 나타낸다.



웹 엔진과 가상 호스트, 웹 컨텍스트 간의 유효한 관계의 예

6.3.1. URL 매칭 예제

웹 엔진과 가상 호스트, 웹 컨텍스트 간의 유효한 관계의 예를 기반으로 각각의 URL이 매칭되는 가상 호스트와 웹 컨텍스트는 다음과 같다.

- `http://www.foo.com/ctx1/test.jsp`

매칭되는 가상 호스트	A
매칭되는 웹 컨텍스트 이름	ctx1

- `http://www.foo.com/ctx2/test.jsp`

매칭되는 가상 호스트	A
매칭되는 웹 컨텍스트 이름	ctx2

- `http://www.bar.com/ctx1/`

매칭되는 가상 호스트	B
-------------	---

매칭되는 웹 컨텍스트 이름	ctx1-1
----------------	--------

• `http://www.bar2.com/ctx1/test.jsp`

매칭되는 가상 호스트	C
매칭되는 웹 컨텍스트 이름	없음 (404 Not Found)

• `http://www.foo2.com/ctx2/`

매칭되는 가상 호스트	C
매칭되는 웹 컨텍스트 이름	ctx2-2



웹 컨텍스트 이름과 컨텍스트 패스는 서로 다른 개념이다. 일반적으로 같은 값을 사용하지만 지금처럼 가상 호스트를 이용해서 서비스를 구분하는 경우에는 서로 달라진다.

6.3.2. URL 매칭 순서

URL이 매칭되는 순서는 다음과 같다.

1. Host 헤더의 도메인 이름 및 포트 문자열을 등록된 모든 가상 호스트에 매칭시킨다. 매칭된 가상 호스트가 있다면 그 안에서 웹 컨텍스트를 찾는다.



가상 호스트에 설정한 호스트 이름에 "IP:Port"와 같은 형식으로 포트 정보도 매핑할 수 있다. 포트가 있는 경우에는 Host 헤더값 전체(포트 포함)를 매칭하는 작업을 수행한다.

2. 웹 컨텍스트가 발견되지 않았으면 기본 가상 호스트에서 찾는다.
3. 기본 가상 호스트에서 원하는 웹 컨텍스트가 없으면 "404 Not Found" 에러가 발생한다.

6.4. 가상 호스트 설정

콘솔 툴을 사용하여 가상 호스트를 추가, 수정 및 삭제할 수 있다.



본 절의 설정 예제에서는 편의상 이름을 "A", "B", "C"로 사용하였다. 실제 환경에서는 의미 있는 이름을 사용한다.

6.4.1. 추가

콘솔 툴을 사용하여 가상 호스트를 추가할 수 있다.

콘솔 툴 사용

콘솔 툴을 사용하여 가상 호스트를 추가하려면 다음과 같이 **add-virtual-host** 명령어를 수행한다.

```
add-virtual-host [-cluster <cluster-name> | -server <server-name>]
                  [-f, --forceLock]
                  <virtual-host-name>
                  -list <host-name-list>
                  [-tmin <The minimum number of threads>]
                  [-tmax <The maximum number of threads>]
                  [-tidle <max-idle-time>]
                  [-qs <max-queue-size>]
```

명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "add-virtual-host"를 참고한다.

6.4.2. 수정

콘솔 툴을 사용하여 가상 호스트를 수정할 수 있다.

콘솔 툴 사용

콘솔 툴을 사용하여 가상 호스트를 수정하려면 다음과 같이 **modify-virtual-host** 명령어를 수행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "modify-virtual-host"를 참고한다.

```
modify-virtual-host [-cluster <cluster-name> | -server <server-name>]
                    [-f, --forceLock]
                    <virtual-host-name>
                    [-alhn1 <access-log-enable-host-name-lookup> |
                    -aluse <use-access-log (true/false)> | -alf
                    <access-log-format> | -aluph
                    <access-log-use-parent-handler (true/false)> |
                    -alex1 <access-log-excluded-extensions>]
                    [-hnrm <host-name> | -hnadd <host-name>]
                    [-ast <attach-stacktrace-on-error>]
                    [-fhn <file-handler-name>]
                    [-fhp <file-handler-permission>]
                    [-tmin <The minimum number of threads>]
                    [-tmax <maximum-thread-num>]
                    [-tidle <max-idle-time>]
```

6.4.3. 삭제

콘솔 툴을 사용하여 가상 호스트를 삭제할 수 있다.

콘솔 툴 사용

콘솔 툴을 사용하여 가상 호스트를 삭제하려면 다음과 같이 **remove-virtual-host** 명령어를 수행한다. 명령어에 대한 자세한 내용은 JEUS Reference 안내서의 "remove-virtual-host"를 참고한다.

```
remove-virtual-host [-cluster <cluster-name> | -server <server-name>]  
                    [-f, --forceLock]  
                    <virtual-host-name>
```

7. WebSocket 컨테이너

본 장에서는 WebSocket 컨테이너의 개념과 기능, 설정 방법에 대해 설명한다.

7.1. 개요

WebSocket 컨테이너는 웹 엔진에 내부적으로 포함되어 있는 형태이며 웹 컨텍스트마다 하나씩 존재한다. JEUS에서 제공하는 WebSocket 컨테이너는 JSR 356, Java API for WebSocket을 준수한다. 따라서 서버의 WebSocket 서비스는 해당 표준에서 정의한 API에 따라 작성한다. 단, 본 장에서는 WebSocket RFC6455 및 Java API for WebSocket 표준에 대한 설명은 포함하지 않는다.



1. HTML5 WebSocket API는 클라이언트 라이브러리이므로 WebtoB 및 JEUS와는 관계없다.
2. Java API for WebSocket은 Jakarta EE 8에 포함되어 있다. 따라서 Jakarta EE 8 기반의 개발을 한다면 기본적인 Jakarta EE 8 라이브러리를 참조해서 개발한다.

WebSocket 컨테이너의 기본적인 역할은 웹 컨텍스트에 포함된 WebSocket Server Endpoint 개체들을 deploy해주고, 클라이언트로부터 WebSocket Handshake 요청이 왔을 때 이에 매핑되는 Server Endpoint 개체를 연결시켜주는 것이다. 클라이언트와 WebSocket 연결이 맺어지면 WebSocket 세션이 생성되는데 이 Session 개체에 대한 라이프 사이클도 관리해 준다.

서비스 개발자의 역할은 WebSocket Server Endpoint 클래스(`jakarta.websocket.Endpoint`) 및 Configuration 클래스(`jakarta.websocket.server.ServerApplicationConfig`)를 작성해서 웹 애플리케이션에 패키징하는 것이다. 또한 annotation을 이용한 pojo style로 개발할 수도 있다.

7.2. 제약 사항

JEUS 9에서 WebSocket 컨테이너를 사용할 때는 다음과 같은 제약 사항이 존재한다.

- 기본적으로 HTTP 리스너에서 사용 가능하다. WebtoB와 함께 사용하기 위해서는 WebtoB 매뉴얼을 참고한다.
- `jeus-web-dd.xml`의 `<websocket>` 항목에 자세한 설정이 가능하나, 해당 설정이 없어도 WebSocket을 사용할 수 있다.
- Java API for WebSocket 표준에서 제공하는 클라이언트 API는 `lib/system/jeus-websocket-client.jar`로 제공한다. 만약 다른 벤더사의 클라이언트 모듈을 사용하고 싶을 때는 위의 jar 파일의 META-INF/service/jakarta.websocket.ContainerProvider 파일 안에 타 벤더의 Provider class의 Full name을 지정하면 Service loader를 통해 사용할 수 있다.

7.3. WebSocket 컨테이너 기능

본 절에서는 WebSocket 컨테이너는 부가적인 기능에 대해 설명한다.

7.3.1. WebSocket UserProperties Failover

WebSocket 컨테이너는 WebSocket 세션별로 메모리에 데이터를 저장할 수 있는 공간을 제공한다.

`jakarta.websocket.Session.getUserProperties()` API를 호출해서 얻은 Map 객체(이하 UserProperties)가 그 공간을 나타낸다.

WebSocket 애플리케이션(Websocket Server Endpoint가 포함된 웹 컨텍스트)을 서로 다른 두 서버에 deploy한 상황이라고 가정하면 한쪽 서버에 WebSocket을 연결해서 UserProperties에 데이터를 저장하면서 사용했는데 그 서버가 비정상 종료되면 데이터가 모두 사라지게 된다. 이를 만약 다른 서버에 백업해둔다면 한쪽 서버가 비정상 종료되더라도 다른 서버로 WebSocket을 연결해서 UserProperties를 복원할 수 있다. WebSocket 서비스 사용자 입장에서는 아무 문제 없이 서비스를 사용할 수 있다. 이를 **WebSocket UserProperties Failover**나 **Distributed WebSocket UserProperties**이라고 한다.



기존의 사용자는 미리 저장해 놓은 UserProperties 안의 데이터는 가져올 수는 있으나, failover된 WebSocket Endpoint는 기존 서버와는 다른 서버에 존재하는 endpoint이고, WebSocket 세션은 새롭게 생성된다. 따라서 이 기능은 매우 한정적인 케이스에 한해서 사용되어야 한다.

주의사항

이 기능을 사용하기 위해서는 다음과 같은 조건들을 만족해야 한다.

- WebSocket UserProperties Failover는 HTTP 세션 서비스를 기반으로 동작한다. WebSocket 세션 클러스터를 지원해야 한다. 세션 클러스터에 대한 자세한 내용은 JEUS 세션 관리 안내서의 "세션 클러스터 모드"를 참고한다.
- WebSocket 애플리케이션의 `jeus-web-dd.xml` 설정 파일에 다음과 같이 기술되어야 한다.

WebSocket UserProperties Failover 설정 : `<jeus-web-dd.xml>`

```
<websocket>
  <distributed-userProperties>
    <enabled>true</enabled>
  </distributed-userProperties>
</websocket>
```

즉, 기본적으로는 WebSocket 세션과 HTTP 세션을 연동하지 않는다.

- UserProperties에 put하는 데이터는 Serializable해야 한다.
- WebSocket 연결은 HTTP 요청으로 시작하는데 이때 WebSocket Handshake 요청의 Cookie 헤더에 JSESSIONID가 있어야 한다. JSESSIONID가 가리키는 HTTP 세션은 Server Endpoint가 포함된 웹 컨텍스트에 생성되어 있어야 한다.

예를 들어 HTML 5 WebSocket API를 호출하도록 작성된 JSP를 호출하면 HTTP 세션이 생성되어 응답 Cookie 헤더로 JSESSIONID가 전달된다. 웹 브라우저에서 HTML 5 WebSocket JavaScript를 실행하면서 JEUS로 WebSocket Handshake 요청을 보내는데, 그 요청 헤더에 Cookie 헤더가 들어가야 한다. 이를 지원하지 않는 **WebSocket Client**를 사용하면 **WebSocket UserProperties Failover** 기능을 사용할 수 없다.

Firefox 30.0은 HTML 5 WebSocket API를 사용할 때 WebSocket Handshake 요청에 Cookie 헤더를 붙여준다.

```
GET /service/chat HTTP/1.1
Connection: keep-alive, Upgrade
Cookie:
JSESSIONID=FheDy8e0b0TP07KNqdeJ7Eps8j51CaQqcRWHvpYo9mdVw1BCSwlwrFiyrc1solkr.amV1czcvc2VydMvYmg==
Sec-WebSocket-Key: Csv/FCQo1g1eZfqMtPd8+g==
Sec-WebSocket-Version: 13
Upgrade: websocket
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:30.0) Gecko/20100101 Firefox/30.0
FirePHP/0.7.4
```



WebtoB Reverse Proxy와 함께 이 기능을 사용할 경우 Reverse Proxy 절에서 Session Routing 설정을 해야 한다. 이에 대한 자세한 사항은 WebtoB 매뉴얼을 참고한다.

주의사항

이 기능을 사용할 때는 다음과 같은 사항을 주의해야 한다.

- HTTP 세션의 타임아웃과 WebSocket 세션의 타임아웃은 서로 독립적이다.

특히 WebSocket 세션의 경우 HTTP 세션에 의존적으로 동작하는 것이기 때문에 WebSocket 컨테이너 차원에서 HTTP 세션의 속성을 마음대로 바꿀 수는 없다. 상황에 따라서는 UserProperties에 Serializable한 데이터를 put하는 시점에 HTTP 세션이 먼저 타임아웃 처리가 되어서 Failover가 되지 않을 수 있다. 이러한 상황을 고려해서 HTTP 세션의 타임아웃을 적절하게 조정해야 한다.



UserProperties에 Serializable한 데이터를 put하는 시점에 HTTP 세션이 타임아웃된 경우에는 WARNING 레벨의 로그를 남긴다.

- 서로 다른 웹 컨텍스트 간에 WebSocket 세션의 UserProperties는 공유되지 않는다.

7.3.2. 기타

jakarta.websocket.Session API를 통해서 얻을 수 없는 정보

Session.getUserProperties()의 리턴값인 Map<String, Object>에서 아래와 같은 정보들을 얻을 수 있다.

항목	설명
jeus.websocket.remoteAddr(String)	HTTP 요청 헤더에 Forwarded 또는 X-Forwarded-For가 있는 경우에는 해당 헤더의 값을 사용한다. 없는 경우에는 HttpServletRequest.getRemoteAddr() 값을 사용한다.

항목	설명
jeus.websocket.remoteHost(String)	<p>HTTP 요청 헤더에 Forwarded 또는 X-Forwarded-For가 있는 경우에는 해당 헤더의 값을 사용한다.</p> <p>없는 경우에는 HttpServletRequest.getRemoteHost()와 같다.</p>
jeus.websocket.remotePort(String)	<p>HTTP 요청 헤더에 Forwarded 또는 X-Forwarded-For가 있는 경우에는 해당 헤더의 값에 포함된 포트를 사용한다. 만약 헤더는 있는데 포트 정보가 없다면 이 프로퍼티는 제공하지 않는다.</p> <p>헤더가 없는 경우에는 HttpServletRequest.getRemotePort() 값을 사용하되, 0보다 큰 경우에만 유효하다.</p>

7.4. WebSocket 컨테이너 설정

WebSocket 컨테이너의 정보는 각 웹 애플리케이션의 jeus-web-dd.xml에 설정한다.

웹 컨텍스트 설정 파일 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
  <websocket>
    <max-incoming-binary-message-buffer-size>8192</max-incoming-binary-message-buffer-size>
    <max-incoming-text-message-buffer-size>8192</max-incoming-text-message-buffer-size>
    <max-session-idle-timeout-in-millis>1800000</max-session-idle-timeout-in-millis>
    <monitoring-period-in-millis>300000</monitoring-period-in-millis>
    <blocking-send-timeout-in-millis>10000</blocking-send-timeout-in-millis>
    <async-send-timeout-in-millis>30000</async-send-timeout-in-millis>
    <websocket-executor>
      <min>10</min>
      <max>30</max>
      <keep-alive-time>60000</keep-alive-time>
      <queue-size>4096</queue-size>
    </websocket-executor>
    <distributed-userProperties>
      <enabled>false</enabled>
      <use-write-through-policy>false</use-write-through-policy>
    </distributed-userProperties>
    <init-param>
      <name>name</name>
      <value>value</value>
    </init-param>
  </websocket>
  <upgrade>
    <upgrade-executor>
      <min>0</min>
      <max>30</max>
      <keep-alive-time>60000</keep-alive-time>
      <queue-size>4096</queue-size>
    </upgrade-executor>
  </upgrade>
</jeus-web-dd>
```

다음은 설정 태그에 대한 설명이다.

• <websocket>

태그	설명
<max-incoming-binary-message-buffer-size>	<p>클라이언트로부터 전달되는 바이너리 메시지를 버퍼링할 때 사용하는 버퍼의 최대값을 의미한다.</p> <p>설정된 값보다 큰 메시지가 전달되는 경우 1009 에러를 내고 WebSocket 세션을 닫는다.</p>
<max-incoming-text-message-buffer-size>	<p>클라이언트로부터 전달되는 텍스트 메시지를 버퍼링할 때 사용하는 버퍼의 최대값을 의미한다.</p> <p>설정된 값보다 큰 메시지가 전달되는 경우 1009 에러를 내고 WebSocket 세션을 닫는다.</p>
<max-session-idle-timeout-in-millis>	<p>유휴 상태의 WebSocket 세션을 언제 닫을 것인지 결정하는 값이다.</p> <p>설정된 값이 0보다 크고 1000보다 작을 경우에는 무조건 1000으로 취급한다. (기본값: 30분(1800000ms))</p>
<monitoring-period-in-millis>	<p>WebSocket 세션 타임아웃 여부를 체크하기 위한 주기를 설정한다.</p> <p>설정된 값이 1000보다 작을 경우에는 무조건 1000으로 취급한다. (기본값: 5분(300000ms))</p>
<blocking-send-timeout-in-millis>	<p>Synchronous Send을 사용하는 경우 대기할 시간을 설정한다.</p> <p>jakarta.websocket.RemoteEndpoint.Basic을 사용할 때 적용된다.</p> <p>(기본값: 10초)</p>
<async-send-timeout-in-millis>	<p>Asynchronous Send을 사용하는 경우 서버 상에서 보내지 못하고 있는 메시지에 대한 타임아웃을 나타낸다.</p> <p>jakarta.websocket.WebSocketContainer.getDefaultAsyncSendTimeout()에서 리턴된다.</p>
<websocket-executor>	<p>더이상 사용하지 않는다. 대신 <upgrade> 하위의 <upgrade-executor>를 사용한다.</p>
<distributed-userProperties>	<p>jakarta.websocket.Session.getUserProperties()에 정의된 내용에 따라 제공하는 WebSocket 세션 Failover 관련 설정이다.</p> <p>다음의 하위 항목을 설정한다.</p> <ul style="list-style-type: none"> ◦ <enabled> : WebSocket 세션의 Failover 사용 여부를 결정한다. HTTP 세션과 연동해야 하기 때문에 기본적으로는 사용하지 않는다. ◦ <use-write-through-policy> : WebSocket 세션의 UserProperties에 put/remove할 때 백업 서버로의 동기화가 끝날 때까지 기다릴 것인지 그 여부를 선택한다. 기본적으로는 기다리지 않고 백그라운드에서 동기화가 일어나도록 한다.

태그	설명
<init-param>	WebSocket Container에서 사용하는 추가 설정을 나타낸다. <name> 항목에 parameter name과 <value>에 설정할 값을 적는다

• <upgrade><upgrade-executor>

Upgrade Inbound Message를 처리하기 위해서 Container 내부적으로 사용하는 스레드 풀 관련 설정이다. 주로 Http Upgrade Handler에서 NIO Servlet 처리 시, WebSocket Endpoint 처리 시, WebSocket asynchronous send시 SendHandler를 처리 시 사용된다.

태그	설명
<keep-alive-time>	Min을 초과하는 스레드에 대해서 설정된 시간 동안 사용되지 않는다면 자동적으로 Thread Pool에서 제거된다. 0이면 제거하지 않는다. (기본값: 1분(60000ms))
<queue-size>	Thread Pool이 처리하는 Task를 저장하는 Queue의 크기를 지정한다. (기본값: 4096)

7.5. Spring WebSocket 사용

본 절에서는 JEUS에서 Spring WebSocket을 사용하는 방법에 대해서 설명한다.

Spring WebSocket은 웹 엔진에서 제공하는 WebSocket 컨테이너를 사용하면서 추가적인 기능을 제공하는 프레임워크이다. JSR 356, Java API for WebSocket을 준수하는 WebSocket 컨테이너는 동일한 인터페이스를 제공하나 WebSocket 컨테이너를 얻는 방법은 벤더별로 차이가 있다.

Spring WebSocket에서는 현재 특정 벤더에 대해서만 WebSocket 컨테이너를 얻는 방법을 지원한다. 따라서 JEUS에서는 Spring WebSocket을 사용하기 위한 별도의 라이브러리를 제공하며 이 라이브러리를 사용하기 위한 추가 설정이 필요하다.



현재 JEUS와 함께 제공되는 라이브러리는 Spring 프레임워크 4.2.0 이상의 버전을 지원한다.

애플리케이션에 Spring WebSocket 지원 라이브러리를 추가하기 위한 설정을 한다.

웹 컨텍스트 설정 파일 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
  <library-ref>
    <library-name>spring-support</library-name>
    <specification-version>
      <value>4.2.0.RELEASE</value>
    </specification-version>
    <failon-error>true</failon-error>
  </library-ref>
</jeus-web-dd>
```

Spring WebSocket 지원 라이브러리를 사용하기 위하여 websocket handshake-handler로

jeus.spring.websocket.JeusHandshakeHandler를 지정해준다.

스프링 컨텍스트 설정 파일 : <spring-context.xml>

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:websocket="http://www.springframework.org/schema/websocket"
  xmlns:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
    http://www.springframework.org/schema/websocket
    http://www.springframework.org/schema/websocket/spring-websocket-4.2.xsd">

  <websocket:handlers>
    <websocket:mapping path="/chat" handler="chatHandler"/>
    <websocket:handshake-handler ref="handshakeHandler"/>
  </websocket:handlers>

  <bean id="chatHandler" class="com.tmax.jeus.ChatHandler"/>
  <bean id="handshakeHandler" class="jeus.spring.websocket.JeusHandshakeHandler"/>
</beans>
```

8. JEUS WebCache

본 장에는 웹 애플리케이션의 성능 향상을 위한 JEUS WebCache를 적용하는 방법에 대해 설명한다.

8.1. 개요

동시 요청자 수의 증가로 인하여 응답 속도가 떨어지는 등의 웹 애플리케이션 성능 저하 문제가 발생한다면 하드웨어 측면과 소프트웨어 측면에서 해결할 수 있다.

하드웨어 측면의 해결 방법은 서버를 증설하여 계속 들어오는 요청(request)을 Load Balancing하여 부하를 분산시켜 응답 속도를 높이는 방법이다. 그러나 이 방법은 서버 추가에 대한 비용 증가와 클러스터링 등으로 인한 서버 운용 및 관리의 복잡함을 발생시킬 수 있다.

소프트웨어 측면의 해결 방법은 서버의 증설 없이 웹 애플리케이션에서 많이 사용되는 데이터를 Caching하는 것이다. 그러면 다음 요청에서는 데이터의 재생산 없이 Caching된 데이터를 이용함으로써 웹 애플리케이션의 응답시간을 줄여서 성능을 향상시킬 수 있다.

본 장에서는 JEUS 시스템에서 웹 애플리케이션의 성능 향상을 위해서 JEUS WebCache를 어떻게 적용할 수 있는지에 대해서 설명한다. 제공되는 Caching 방법은 다음과 같다.

- JSP Caching

태그 라이브러리(Tag Library)를 사용하여 JSP 내의 일부분을 Caching한다. 전체 페이지 중에서 부분 변경이 발생하는 JSP 페이지를 요청하는 경우에 적합하다.

- HTTP Response Caching

HTTP 응답 전체를 Caching한다. 정적 콘텐츠(Static Content)에 대한 요청에 적합하다.

JEUS WebCache에 Caching되는 엔트리는 SoftReference로 구현되어 있다. 그래서 심각한 메모리 증가로 인한 OutOfMemory 에러를 미연에 방지할 수 있다.

Caching 기능을 효과적으로 사용하기 위해서 빈번하게 요청되거나, 수행 로직이 복잡해서 또는 데이터베이스로부터 데이터를 가져오는 시간이 오래 걸려 응답시간이 길어질 수 있는 웹 페이지를 Caching 대상으로 선택하는 것이 바람직하다.

8.2. JSP Caching

JSP Caching은 JSP 태그 라이브러리를 사용하여 JSP 페이지 내의 일부분을 JEUS WebCache에 저장함으로써 웹 애플리케이션의 성능을 향상시키는 방법이다.

8.2.1. 기본 내용

JEUS WebCache에서는 사용자 정의(custom) 태그로 **<jeus:cache>**를 사용한다.

<jeus:cache> 내에 JSP 페이지 내에서 Caching을 원하는 콘텐츠가 있는 부분을 입력하면 첫 번째 요청에서 태그

내의 바디 콘텐츠(Body Contents)가 생성되어 브라우저에 전송되고 이 콘텐츠는 Caching된다. 다음 요청부터는 메모리에 Caching된 콘텐츠가 브라우저로 보내진다.

<jeus:cache> 태그를 사용하는 기본적인 형식은 다음과 같다.

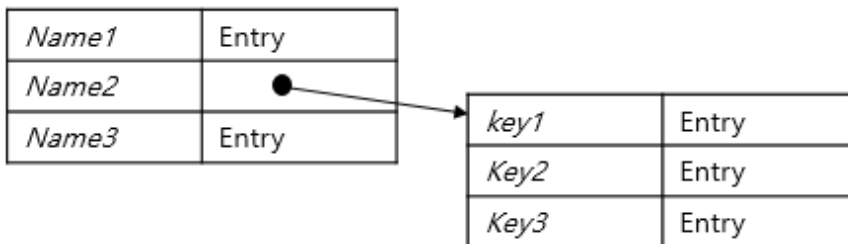
```
<%@ taglib uri="http://www.tmaxsoft.com/jeuscache" prefix="jeus" %>

<jeus:cache name="..." key="..." scope="..." timeout="..."
    size="..." async="..." df="...">
    . . . Body content to be cached. . .
</jeus:cache>
```

위 설명에서 제외된 **flush** 속성은 닫힌 태그(/>)를 사용하는데 속성에 대한 설명은 <cache> 태그를 참고한다.

JSP Caching에서 사용하는 알고리즘은 LRU이다. 그래서 JEUS WebCache 최대 허용 개수를 초과하면 LRU 알고리즘에 의해서 기존에 Caching된 엔트리가 제거된다. 그리고 TLD(Tag Library Descriptor) 파일은 jeus.jar 파일에 포함되어 배포된다. jeuscache.tld 파일에 대한 URI 정보를 JSP 엔진에 전달하기 위해서 <jeus:cache> 태그 내의 'taglib uri'를 반드시 'http://www.tmaxsoft.com/jeuscache'으로 명시해야 한다.

다음은 'name' 속성, 'name' 속성 + 'key' 속성을 사용하여 엔트리를 Caching할 때 사용하는 자료 구조이다.



엔트리 Caching에 사용되는 자료 구조

위 그림에서 Name1, Name3은 'key' 속성 없이 'name' 속성만으로 태그를 사용할 때 엔트리가 Caching되는 방식이며 'key' 속성은 물론 'name' 속성까지도 사용되지 않을 때도 이 방식이 사용된다. 그러나 'name' 속성과 'key' 속성 모두가 사용되는 태그에서는 Name2와 같은 방식으로 엔트리가 Caching된다.

8.2.2. <cache> 태그

<jeus:cache> 사용자 태그를 정의한 파일은 jeuscache.tld로 이 파일은 jeus-servlet.jar 내에 포함되며 위치는 다음과 같다.

```
jeus/servlet/cache/resource/jeuscache.tld
```

다음은 jeuscache.tld 파일의 <cache> 태그의 설정 예제이다.

<cache> 태그 설정 : <jeuscache.tld>

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.2//EN" "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```



```

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>jeuscache</short-name>
  <uri>http://www.tmaxsoft.com/jeuscache</uri>
  <display-name>JEUSCache Tag Library</display-name>
  <tag>
    <name>cache</name>
    <tag-class>jeus.servlet.cache.web.tag.CacheTag</tag-class>
    <body-content>JSP</body-content>
    <description>JEUS WebCache</description>
    <attribute>
      <name>flush</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>timeout</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>scope</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>name</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>size</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>key</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>async</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>df</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>

```

다음은 각 태그를 설정할 때 각 속성에 대한 설명이다.

속성	설명
flush	<p>Cache 내 엔트리를 제거하기 위해서 사용한다. 특정 엔트리를 제거하기 위해서는 반드시 'name' 속성이나 'name' 속성 + 'key' 속성을 지정해야 된다.</p> <p>'name' 속성에 'key' 속성을 이용해서 다수의 엔트리를 Caching한 경우 'name' 속성만을 사용하여 'flush' 속성을 수행할 때 'key' 속성을 식별자로 하는 모든 엔트리가 제거된다는 점에 주의한다. 그리고 이 속성은 다른 속성과는 다르게 바디 콘텐츠가 없는 닫힌 태그(</>)를 사용한다.</p>
timeout	<p>콘텐츠가 Caching될 기간을 설정하며 Simple Date Format 형식으로 명시된다. 숫자와 시간을 나타내는 한 문자의 결합으로 표시한다.</p> <p>기간을 명시하는 유효한 문자는 's'(seconds), 'm'(minutes), 'h'(hours), 'd'(days), 'w'(weeks)이다. 예를 들어 10s는 10초, 10d는 10일, 그리고 4w는 4주를 나타내고, 문자 없이 숫자만 설정하면 초(second) 단위로 인식한다.</p> <p>0으로 설정하면 매번 바디 콘텐츠를 생성하여 refresh 효과를 낼 수 있다. -1이 설정된다면 강제로 flush되지 않는 이상 콘텐츠는 expire되지 않는다.</p> <p>(기본값: 1시간)</p>
scope	<p>cache되는 엔트리의 운용 범위를 나타낸다.</p> <ul style="list-style-type: none"> ◦ application(기본값) ◦ session
name	<p>여러 페이지에서 Caching된 데이터를 공유하기 위해 사용되며 해당 scope 내에서 유일해야 한다.</p> <p>name 속성을 지정하지 않았을 때는 HTTP URI 정보 등을 사용하여 내부적으로 생성하여 관리된다. jeus:cache 내의 바디 콘텐츠가 공유될 필요가 없다면 이 속성을 설정하지 않는 것이 바람직하다.</p>
size	<p>Caching할 수 있는 객체의 최대 개수를 설정한다.</p> <p>설정된 값을 초과한 경우에는 LRU 알고리즘에 따라서 Cache에 저장된 객체가 제거된다. 이 값은 웹 애플리케이션 상황에 맞게 적절하게 설정되어야 한다. (기본값: Integer.MAX_VALUE)</p>

속성	설명
key	<p>Caching되는 엔트리를 식별하는 추가적인 값이다. 이 속성은 'name' 속성과 반드시 함께 사용되어야 한다. 그래서 엔트리를 식별할 때는 'name' + 'key' 값이 유일한 식별자로 사용된다.</p> <p>'key' 속성은 다음과 같은 scope를 지정하는 나열 형식으로 설정 가능하다.</p> <pre><jeus:cache name=". . ." key="[parameter page session request application].keyname" . . . ></pre> <p>위의 예에서 'keyname'의 값은 요청할 때 설정해야 한다. 'parameter', 'page', 'request'의 경우 cache 페이지와 같은 요청 페이지에서 'keyname'을 key 값으로 설정해야 하고, 'application', 'session'의 경우 다른 페이지에서 설정이 가능하다.</p> <pre>http://sample.com:8088/Sample/index.jsp?keyname=test</pre>
async	<p>하나의 Thread가 엔트리를 업데이트 중인 경우(미완료 상태)에 동일한 엔트리를 요구하는 다른 Thread에 대해서 Blocking 여부를 설정한다.</p> <ul style="list-style-type: none"> ◦ true : Thread는 Blocking되지 않고 이전 엔트리 즉, 업데이트되지 않은 엔트리를 가져간다. (기본값) ◦ false : 엔트리가 업데이트될 때까지 기다린(Blocking) 후 최신 엔트리를 가져간다.
df	<p>Overflow가 발생했을 경우 공간확보를 위해서 엔트리를 제거한다. 이때 제거되는 victim 개수를 설정된 'size' 속성에 대한 비율을 'df' 속성으로 설정할 수 있다.</p> <p>유효한 값의 범위는 '0.0 <= factor <= 1.0' 사이의 실수값이다. (기본값: 0.25)</p> <p>예를 들어 factor 1.0은 메모리에 Cache되어 있는 모든 엔트리를 제거한다는 의미이며, factor 0.0은 overflow가 발생할 경우 추가 요청에 대해 더 이상 cache하지 않는다는 의미이다.</p>

8.2.3. <jeus:cache> 사용 예

본 절에서는 예제를 사용하여 <jeus:cache> 태그 속성의 사용 방법을 설명한다.

다음은 cache.jsp 페이지를 요청하는 경우 현재 날짜와 Caching된 날짜를 비교해보는 간단한 예제이다.

<jeus:cache> 사용 예제 : <cache.jsp>

```
<%@ taglib uri="http://www.tmaxsoft.com/jeuscache" prefix="jeus" %>
<HTML>
<BODY>
  Current time: <%= new Date() %><br>
  <jeus:cache timeout="60s">
    Cached time: <%= new Date() %>
  </jeus:cache>
</BODY>
```

```
</HTML>
```

<jeus:cache> 태그의 첫 번째 요청에서는 현재 날짜를 구하여 화면에 출력하고 JEUS WebCache에 저장된다. 다음 호출부터는 Caching된 날짜가 출력될 것이다. 60초가 지난 다음에는 갱신된 날짜가 출력된다.

다음은 `<jsp:include>`를 사용하여 다른 페이지를 포함할 때 **<jeus:cache>** 태그를 사용하는 예이다.

jsp:include를 이용한 `<jeus:cache>` 사용 예제 : `<main.jsp>`

```
<HTML>
<BODY>
  <jsp:include page="cache.jsp"/>
</BODY>
</HTML>
```

main.jsp에 include된 cache.jsp 내의 `<jeus:cache>` 태그의 내용은 cache.jsp만 사용하는 첫 번째 예제와 동일한 수행 결과를 출력한다.

8.2.4. flush 기능

flush 기능은 Caching된 엔트리를 강제로 제거하는 기능이다. 대상이 되는 엔트리를 지정하는 'name' 속성 또는 'name' 속성 + 'key' 속성을 반드시 명시해야 한다.

예를 들어 다음과 같이 'name' 속성 + 'key' 속성을 사용해서 stock content를 Caching했다고 가정하자.

```
<jeus:cache name="stock" key="parameter.company" scope="application">
  . . . stock content . . .
</jeus:cache>
```

Caching된 parameter.company에 해당하는 stock content를 제거하기 위해서는 다음과 같이 설정한다.

```
<jeus:cache name="stock" key="parameter.company" scope="application" flush="true"/>
```

위와 같이 **flush** 기능을 성공적으로 수행했다면 다음 `<jeus:cache>` 태그 내의 stock content를 요청하는 경우에는 갱신된 stock content를 보게 된다. 만일 위 **flush** 속성을 다음과 같이 'key' 속성 없이 'name' 속성만을 사용한다면 parameter.company key 속성으로 저장된 모든 엔트리가 제거될 것이다.

```
<jeus:cache name="stock" scope="application" flush="true"/>
```

물론 'key' 속성을 사용하지 않고 'name' 속성만으로 엔트리를 Caching했다면 'name' 속성만 사용하면 된다.

8.2.5. refresh 기능

모든 **<jeus:cache>** 태그를 호출할 때마다 바디 콘텐츠를 생성하게 하는 refresh 기능을 제공한다. 이 기능은 **<jeus:cache>** 태그 속성을 사용하지 않는다. 대신 '_jeuscache_refresh'를 key로, true를 값으로 해서 원하는 scope에 설정할 수 있다.

애플리케이션과 세션 scope의 모든 엔트리를 refresh하기 위해서는 다음과 같이 각각 작성한다.

```
<% application.setAttribute("_jeuscache_refresh", "true"); %>
<% session.setAttribute("_jeuscache_refresh", "true"); %>
```

이 기능은 **<jeus:cache>** 태그에 접근했을 때 각각의 scope에 '_jeuscache_refresh' 값을 조사하여 true일 경우 그 바디 콘텐츠를 갱신한다. 그리고 더 이상 refresh 기능을 사용하지 않기를 원한다면 '_jeuscache_refresh'을 false로 지정한다. 'timeout', 'flush' 속성을 사용하면 하나 또는 일부분의 갱신된 엔트리를 볼 수 있는 반면, refresh기능을 사용하면 설정한 scope 내의 모든 바디 콘텐츠가 매번 갱신된다.

8.3. HTTP Response Caching

JEUS WebCache는 Servlet Filter를 사용하여 HTTP 응답 전체를 Caching하는 방법으로 HTTP Response Caching 기능을 제공한다.

페이지 내용이 동적으로 변하는 웹 페이지에는 적절하지 않으며 정적 콘텐츠에 대한 HTTP 요청에 적합한 방법이다. 이미지 파일, PDF 등의 바이너리 콘텐츠를 요구하는 HTTP 요청이 여기에 해당된다. 물론 일정 시간 동안 웹 페이지의 내용이 변경되지 않거나 변경사항이 반영되지 않아도 되는 웹 페이지에도 이 방법을 사용할 수 있다.

```
http://www.sample.com/filter/respcacheTest.jsp?key=value
```

위와 같이 key, value를 포함해서 전체 HTTP URI가 엔트리 키로 적용된다. 만일 key, value 값이 다양하게 변화한다면 각각의 URI에 해당하는 HTTP Response가 Caching된다.



HTTP Response의 상태가 200 OK(HttpServletResponse.SC_OK)인 경우에만 해당 HTTP Response가 Caching된다는 것이다. HTTP Response를 JEUS WebCache에 저장할 때 사용되는 엔트리 키로 HTTP URI가 사용된다.

본 절에서는 HTTP Response Caching을 웹 애플리케이션에 적용하는 방법에 대해 설명한다.

8.3.1. 필터 설정

웹 애플리케이션에서 HTTP Response Caching을 사용하기 위해서는 다음과 같이 web.xml에 필터를 등록한다.

다음은 <url-pattern>이 '/filter/'인 모든 HTTP 요청에 대한 HTTP 응답을 10분 동안 Caching하는 예제이다.



<filter-class>로 반드시 **jeus.servlet.cache.web.filter.CacheFilter** 클래스를 사용해야 한다.

필터 설정 : <web.xml>

```
<web-app>
. . .
<filter>
  <filter-name>CacheFilter</filter-name>
  <filter-class>jeus.servlet.cache.web.filter.CacheFilter</filter-class>
  <init-param>
    <param-name>timeout</param-name>
    <param-value>600</param-value>
  </init-param>
  <init-param>
    <param-name>lastModified</param-name>
    <param-value>on</param-value>
  </init-param>
  <init-param>
    <param-name>expires</param-name>
    <param-value>off</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>CacheFilter</filter-name>
  <url-pattern>/filter/*</url-pattern>
</filter-mapping>
. . .
</web-app>
```

다음은 필터 클래스에 전달하는 초기화 파라미터에 대한 설명이다.

파라미터	설명
timeout	<p>HTTP Response를 Caching하고 있을 시간을 설정한다.</p> <p>(기본값: 3600, 단위: 초(second))</p> <p>JSP Caching에서 사용하는 'timeout' 속성과 설정 방법은 동일하다. 단, HTTP Response Caching에서는 flush 기능이 제공되지 않기 때문에 'timeout'을 '-1'로 설정하면 웹 애플리케이션이 undeploy되지 않는 이상 expire되지 않기 때문에 주의한다.</p>
lastModified	<p>HTTP 응답으로 Last-Modified 헤더를 보낼지를 결정하기 위해서 사용된다. 이는 웹 엔진의 부하를 줄이는 목적으로 사용된다.</p> <p>브라우저는 자신이 Cache하고 있는 콘텐츠가 마지막 요청 이후에 변경되었는지 웹 엔진에 요청할 수 있다. 그러면 웹 엔진은 HTTP 요청에서 If-Modified-Since 헤더 정보와 현재 엔트리의 최종 변경시간을 비교하여 엔트리에 변경 사항이 없다는 304 상태코드(HttpServletResponse.SC_NOT_MODIFIED)를 브라우저로 전송한다.</p> <p>유효한 값은 다음과 같다.</p> <ul style="list-style-type: none">◦ on : 최종 변경시간을 Filter Chain 수행 중에 결정하여 304 상태코드를 보낸다.◦ off : HTTP 응답으로 304 상태코드를 보내지 않는다.◦ initial : 최종 변경시간을 현재시간으로 하여 304 상태코드를 보낸다. (기본값)

파라미터	설명
expires	<p>HTTP 응답으로 Expires 헤더 정보를 전송할지를 결정하기 위해서 사용된다. 브라우저에서 Cache 기능을 사용하고 있다면 Cache된 콘텐츠는 사용 기간이 만료될 때까지 유효하며 이후에 계속적인 HTTP 요청에 대해서는 브라우저 Cache에 저장된 콘텐츠를 사용할 것이다.</p> <p>만약 JEUS WebCache의 엔트리가 업데이트되었다면 새로운 엔트리는 브라우저 Cache에 저장된 콘텐츠와 다른 문제가 발생한다. 이런 경우에 웹 엔진에서 Expires 헤더 정보를 설정하여 브라우저 Cache에 저장된 콘텐츠를 기간만료시키고, JEUS WebCache의 업데이트된 엔트리를 사용하도록 해야 한다.</p> <p>유효한 값은 다음과 같다.</p> <ul style="list-style-type: none"> ◦ on : Filter Chain에서 값이 설정된다면 Expires 헤더 정보를 전송한다. ◦ off : Expires 헤더 정보를 전송하지 않는다. ◦ timeout : HTTP 응답의 last-modified 값에 상기에 기술한 timeout 파라미터 값이 더해져서 Expires 헤더 정보로 설정되어서 클라이언트로 전송된다.



위의 파라미터 외에 추가로 **scope, size, async, df** 등을 설정할 수 있는데, 이들은 [JSP Caching](#)에서 설명한 것과 동일한 의미를 가지고 있으므로 해당 부분을 참고한다.

9. 클래스 동적 반영

본 장에는 웹 애플리케이션의 개발 속도 향상을 위한 Servlet Auto Reload 기능에 대해 설명한다.

9.1. 개요

일반적으로 Jakarta EE의 개발 Lifecycle은 다음과 같은 순서를 가지고 있다.

1. 편집
2. 빌드
3. 배치
4. 테스트

개발자들이 Jakarta EE 애플리케이션, 특히 웹 애플리케이션을 개발할 때 서블릿 등의 클래스를 수정하는 경우가 많다. 이러한 개발 과정을 신속하게 수행하기 위해 많은 노력들이 진행되어져 왔고, WebLogic 10.3에서는 FastSwap을 이용하여 이러한 재배포 과정을 줄이기 위한 기능을 제공하고 있다.

Java EE 5에서는 운영 중에 클래스 로더를 내리거나 인스턴스를 종료하지 않고도 클래스를 재정의하는 기능이 소개되었지만 선언된 필드와 메소드의 변경은 불가능한 제약사항을 가지고 있었다.

JEUS 6까지는 클래스를 수정한 뒤 수정된 클래스를 적용하려면 애플리케이션을 redeploy를 하거나, Auto Reload 기능을 통해서 주기적 체크 또는 요청에 따라 클래스 로더를 새로 생성하여 기존의 클래스 로더와 교체해야 했다. 그러나 redeploy할 경우 애플리케이션의 크기가 크면 상당한 시간이 걸리는 작업이고, Auto Reload 기능 또한 클래스 로더를 재작성하기 때문에 redeploy할 때만큼 많은 부하가 걸리고 있다.

이에 따라 JEUS 7부터는 기존의 클래스 로더의 리로딩이 필요한 동적 반영(Auto Reload)을 포함하여, JDK Instrumentation Package를 이용하여 클래스 로더의 리로딩 없이 Java 클래스의 재정의가 가능한 향상된 클래스 동적 반영(Auto Reload) 기능인 **JEUS HotSwap 기능**을 제공한다. 단, 현재는 웹 애플리케이션의 클래스들만 한정하여 지원한다.



본 장에서 설명하는 JEUS의 Auto Reload(JEUS HotSwap 기능의 활성화/비활성) 기능은 운영 상황에서는 원하지 않는 부하를 발생시킬 수 있으므로, 개발 단계에서만 사용할 것을 권장한다.

9.2. 기본 설정 및 동작

본 절에서는 JEUS의 Auto Reload 기능의 세부 설정 및 동작 방식을 설명한다.

9.2.1. 서버 설정

JEUS HotSwap 기능을 사용하려면 JEUS의 서버를 시작하기 전에 시스템 옵션을 설정해야 한다(JEUS HotSwap 기능이 없는 Auto Reload는 서버 시작 시 시스템 옵션 없이 동작한다).

시스템 옵션은 jeus.server.useHotSwapAgent이고, 설정하지 않을 경우 기본값이 false이므로 기능이 동작하지 않는다.

다음은 JEUS 시작 스크립트의 설정 예제이다.

- **startDomainAdminServer 스크립트의 설정 예제**

Jeus HotSwap 설정 : <startDomainAdminServer>

```
...
"${JAVA_HOME}/bin/java" $VM_OPTION $SESSION_MEM
-Xbootclasspath/p:"${JEUS_HOME}/lib/system/extension.jar"
...
-Djeus.server.useHotSwapAgent=true
...
jeus.launcher.Launcher ${BOOT_PARAMETER}
...
```

- **startManagedServer 스크립트의 설정 예제**

Jeus HotSwap 설정 : <startManagedServer>

```
...
"${JAVA_HOME}/bin/java" $VM_OPTION $SESSION_MEM
-Xbootclasspath/p:"${JEUS_HOME}/lib/system/extension.jar"
...
-Djeus.server.useHotSwapAgent=true
...
jeus.launcher.ManagedServerLauncher ${BOOT_PARAMETER}
...
```



Auto Reload 기능을 설정하여 개발을 진행하고, 개발이 완료되면 위에서 설정한 옵션을 제거하여 운영할 때 이 기능을 사용하지 않도록 한다.

9.2.2. 애플리케이션 설정

JEUS Auto Reload 기능을 사용하려면 JEUS의 jeus-web-dd.xml에 <auto-reload>를 설정해야 한다. 그리고 JEUS HotSwap 기능을 사용하기 위해서는 <auto-reload> 하위의 <use-jvm-hotswap> 설정을 해야 한다. 이 기능들은 디렉터리 형태(Exploded Directory)인 웹 애플리케이션의 클래스 파일의 변경만 가능하다. 개발 과정에서 변경된 클래스들의 동적 반영을 위한 것이기 때문에 이미 클래스들이 노출된 디렉터리에 한정한다. 즉, 웹 애플리케이션 디렉터리의 WEB-INF/classes 디렉터리 하위에 있는 클래스의 변경만 지원한다. JEUS의 Auto Reload는 처음 애플리케이션이 배포된 후 클래스 수정이 발생한 시점을 기준으로 동작한다.

jeus-web-dd.xml에 <auto-reload>를 다음과 같이 설정하고, Auto Reload의 모니터링 주기는 domain.xml을 사용하여 설정한다. domain.xml을 통한 Auto Reload의 모니터링 주기 설정에 대한 자세한 내용은 [모니터링 설정](#)을 참고한다.

Auto Reload 설정 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
```

```

. . .
<auto-reload>
  <enable-reload>true</enable-reload>
  <use-jvm-hotswap>true</use-jvm-hotswap>
  <check-on-demand>true</check-on-demand>
</auto-reload>
. . .
</jeus-web-dd>

```

세부 동작은 각 설정에 따라 다음과 같이 동작한다.

태그	설명
<enable-reload>	Auto Reload 기능의 사용여부를 결정한다. <enable-reload>는 true로 설정되어 있어야 한다.
<use-jvm-hotswap>	<ul style="list-style-type: none"> ◦ true : 클래스 로더를 교체하지 않고, 변경된 클래스를 재정의(redefinition)하거나 클래스 재변환(retransformation)을 통해 변경 사항을 적용한다. 만약 변경된 클래스들의 재정의 또는 재변환 시도 중 불가능한 클래스가 존재한다면 JEUS HotSwap 과정을 더 진행하지 않고, 기존의 Auto Reload 과정을 수행하여 변경 사항을 반영한다. JEUS HotSwap이 불가능한 클래스들의 변경 내역은 JEUS HotSwap 제약 사항을 참고한다. ◦ false : JEUS HotSwap을 사용하지 않고, 애플리케이션의 클래스 로더를 교체하여 변경된 클래스를 적용한다.
<check-on-demand>	<ul style="list-style-type: none"> ◦ true : 요청이 들어왔을 때 바로 해당 애플리케이션의 클래스 로더를 기준으로 클래스의 마지막 수정시간을 검사하여 변경된 파일을 찾아낸다. 파일을 검사하여 변경된 클래스가 존재할 경우 <use-jvm-hotswap>의 설정 여부에 따라 Auto Reload가 수행된다. ◦ false : 웹 엔진이 'Check Class Reload' 설정에 따라 일정 주기마다(기본적으로 300초) 해당 애플리케이션 클래스 로더를 기준으로 클래스의 마지막 수정시간을 검사하여 변경된 파일을 찾아낸다. 'Check Class Reload' 설정에 대한 자세한 내용은 모니터링 설정을 참고한다. <p>파일을 검사한 후 변경된 클래스가 존재하면 <use-jvm-hotswap>의 설정 여부에 따라 Auto Reload가 수행된다.</p>

9.2.3. JEUS HotSwap으로 지원 가능한 애플리케이션 및 변환

JEUS HotSwap은 Exploded 디렉터리 내의 POJO(Plain Old Java Object), 웹 애플리케이션 클래스들을 지원한다.

JEUS HotSwap은 다음 타입의 변경을 지원한다.

- 정적 클래스 생성자 추가/제거
- 일반 클래스 생성자 추가/제거
- 정적 메소드 바디 수정
- 일반 메소드 바디 수정

다음은 JEUS HotSwap에서 지원하는 클래스 변환 리스트이다.

- Java 클래스 Instance (non-abstract)

Java Change Type	지원 여부	Notes
메소드 추가/제거	아니오	
필드 추가/제거 (1)	아니오	
메소드 바디 수정 (2)	예	
생성자 추가/제거 (3)	예	
필드 수정자(field modifier) (4)	아니오	

- Static 클래스

Java Change Type	지원 여부	Notes
메소드 추가/제거	아니오	
메소드 바디 수정	예	

- Abstract Java 클래스

Java Change Type	지원 여부	Notes
abstract 메소드 추가/제거	아니오	
Java 클래스 Instance의 1-4 변환	예	

- final Java 클래스

Java Change Type	지원 여부	Notes
Java 클래스 Instance의 1-4 변환	예	

- final Java 메소드

Java Change Type	지원 여부	Notes
Java 클래스 Instance의 1-4 변환	예	

- final Java 필드

Java Change Type	지원 여부	Notes
Java 클래스 Instance의 1-4 변환	예	

- Enum

Java Change Type	지원 여부	Notes
상수 추가/제거	아니오	
메소드 추가/제거	아니오	

- 익명 내부 클래스

Java Change Type	지원 여부	Notes
필드 추가/제거	유효하지 않음	Java 언어에서 제공되지 않는다.
메소드 추가/제거	아니오	

- 정적 내부 클래스

Java Change Type	지원 여부	Notes
Java 클래스 Instance의 1-4 변환	예	

- 일반 내부 클래스

Java Change Type	지원 여부	Notes
Java 클래스 Instance의 1-4 변환	예	

- Java 인터페이스

Java Change Type	지원 여부	Notes
메소드 추가	아니오	

- Java Reflection

Java Change Type	지원 여부	Notes
존재하는 필드/메소드 접근	예	
새로운 메소드 접근	아니오	새로운 메소드는 Reflection을 이용하면 보이지 않는다.
새로운 필드 접근	아니오	새로운 필드는 Reflection을 이용하면 보이지 않는다.

- Annotations on Classes

Java Change Type	지원 여부	Notes
메소드, 속성 Annotation 추가/제거	아니오	

- Annotation 타입

Java Change Type	지원 여부	Notes
메소드, 속성 Annotation 추가/제거	아니오	

- Exception 클래스

Java Change Type	지원 여부	Notes
Java 클래스 Instance의 1-4 변환	예	

• EJB 인터페이스

Java Change Type	지원 여부	Notes
메소드 추가/제거	아니오	Reflection과 연관되어 있는 EJB 인터페이스의 변환은 지원되지 않는다.

• EJB 3.0 Session/MDB, 구현 클래스

Java Change Type	지원 여부	Notes
메소드, 필드 추가/제거	아니오	EJB에 의해 참조되는 클래스 중 지원되는 클래스에 한해 변환이 지원된다.

• EJB 2.X Entity Bean

Java Change Type	지원 여부	Notes
메소드, 필드 추가/제거	아니오	EJB에 의해 참조되는 클래스 중 지원되는 클래스에 한해 변환이 지원된다.

• EJB Interceptors

Java Change Type	지원 여부	Notes
메소드, 필드 추가/제거	아니오	EJB에 의해 참조되는 클래스 중 지원되는 클래스에 한해 변환이 지원된다.

9.2.4. JEUS HotSwap 제약 사항

JEUS HotSwap으로 지원되는 변환과 불가능한 변환이 있다. 지원이 불가능한 변환이 있는 경우 JDK에서 `UnsupportedOperationException`이 발생하고, JEUS는 이를 받아서 다음과 같은 로그를 남긴다. 이 경우 애플리케이션의 동적 재정의는 반영되지 않지만 JEUS의 Auto Reloading 동작은 계속된다.

```
Retransforming all modified classes in the servlet context [AAA] failed.
```

지원이 불가능한 변환들을 정리하면 다음과 같다.

- Java Reflection 결과는 새롭게 변경된 필드와 메소드를 포함하지 못한다. 따라서 수정된 클래스의 API의 Reflection을 사용하는 경우 예상치 못한 동작이 나타날 수 있다.
- 이미 존재하는 클래스의 계층 변경은 지원되지 않는다. 예를 들어 클래스의 인터페이스를 구현한 리스트의 변경이나 클래스의 superclass를 변경하는 경우는 지원되지 않는다.
- Java Annotation의 추가 및 제거는 지원되지 않는다. 이는 위에 언급한 Reflection의 이유 때문이다.
- EJB 인터페이스의 메소드에 대한 추가 및 제거는 지원되지 않는다. EJB 편집은 변경에 대한 Reflect가 필요하기 때문이다.
- Enums의 상수를 추가하거나 제거하는 경우는 지원되지 않는다.

- finalize 메소드에 대한 추가나 제거의 경우는 지원되지 않는다.

10. 밸브와 필터

본 장에서는 JEUS의 밸브와 필터를 설정하는 방법을 확인한다.

10.1. 개요

밸브는 Tomcat에서 가져온 개념으로 요청을 검사 혹은 변경을 위해 추가하는 구성요소를 의미한다. 밸브의 설정은 웹엔진, 가상호스트, 컨텍스트 단위로 설정할 수 있다. 필터는 jakarta.servlet.Filter를 의미하며, 밸브와 달리 서블릿 단위로 설정할 수 있다. 자세한 내용은 jakarta.servlet.Filter 참고한다.

10.2. 밸브의 개념

본 절에서는 밸브에 대한 기본적인 개념과 사용법에 대해 설명한다.

밸브는 서블릿의 필터 개념을 서버와 가상호스트 레벨로 확장시킨 개념이다. 웹 엔진, 가상 호스트, 컨텍스트 순서로 밸브 코드를 실행시키게 되며, 같은 단위 내의 호출 순서는 설정에 지정한 순서대로 시행된다. 필터와 밸브가 모두 설정되어 있는 경우에는 밸브의 설정 모두 적용된 이후에 필터가 적용되게 된다.

jeus-web-dd.xml 설정

WEB-INF/ 디렉터리 아래에 jeus-web-dd.xml을 생성한다. 이와 관련된 자세한 설명은 [웹 컨텍스트](#)를 참고한다.

다음은 jeus-web-dd.xml 파일의 예이다. 몇몇 항목들은 생략되어 있다. 생략된 항목들은 "JEUS XML Reference"의 "13. jeus-web-dd.xml 설정"을 참고한다.

웹 컨텍스트 설정 파일 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
<context-path>/examples</context-path>
  <pipeline>
    <valve>
      <class-name>jeus.servlet.valve.RemoteAddressValve</class-name>
      <property>
        <key>deny</key>
        <value>127\.\0\.\0\1</value>
      </property>
      <property>
        <key>denyStatus</key>
        <value>403</value>
      </property>
    </valve>
  </pipeline>
</jeus-web-dd>
```

다음은 domain.xml의 web-engine에 valve를 설정하는 예시이다. 관련 없는 설정은 생략했다.

서버 설정 파일 : <domain.xml>

```
<domain xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
```

```

<web-engine>
  <pipeline>
    <valve>
      <class-name>jeus.servlet.valve.RemoteAddressValve</class-name>
      <property>
        <key>deny</key>
        <value>127\.\0\.\0\.\1</value>
      </property>
      <property>
        <key>denyStatus</key>
        <value>403</value>
      </property>
    </valve>
  </pipeline>
</web-engine>
</domain>

```

다음은 설정 태그에 대한 설명이다.

태그	설명
<valve>	사용할 valve를 설정한다.
<class-name>	적용할 밸브의 클래스 명이다. 필터의 class-name을 생각하면 된다.
<property>	밸브에 적용할 프로퍼티이다. 이 프로퍼티는 key value 쌍으로 구성되어 있고, 이는 밸브 내부의 파라미터 맵에 저장된다. 여러개 설정할 수 있다.
<key>	valve 객체 내의 파라미터맵에서 사용할 key를 의미한다. 맵 내에선 유일한 값이고, 중복되는 경우 나중에 설정된 key의 value값이 저장된다.
<value>	valve 내부의 파라미터 맵에 저장될 값이다. String으로 저장되며, 타입 캐스팅은 후술할 init() 메소드에서 진행한다.



class-name에 들어갈 class는 jeus.servlet.valve.ValveBase 클래스를 상속받아야 한다.

10.3. JEUS 제공 valve 설정

JEUS에서는 이용자 편의를 위해 밸브 구현체를 제공한다.

10.3.1. 메소드 제한 옵션 설정

method를 검사해서 특정 메소드를 막거나 허용하는 옵션을 의미한다.

메소드 제약 옵션 밸브 추가 : <jeus-web-dd.xml>

```

<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
  <context-path>/examples</context-path>
  <pipeline>
    <valve>
      <class-name>jeus.servlet.valve.MethodConstraintValve</class-name>
    </valve>
  </pipeline>
</jeus-web-dd>

```



```

        <property>
            <key>allow</key>
            <value>GET,POST</value>
        </property>
    </valve>
</pipeline>
</jeus-web-dd>

```

JEUS가 제공하는 메소드 제약 옵션에 사용할 클래스 이름은 `jeus.servlet.valve.MethodConstraintValve`이다. `key`로 `allow`와 `deny`를 줄 수 있으며, `value`로 메소드 목록을 주면 된다. 이 때, 각 http 메소드를 구분하는 구분자는 콤마(,)이다.

10.3.2. remote host/addresss valve 설정

원격 호스트/주소를 허용할지 말지 결정하는 옵션을 제공한다.

차단하고자 하면 `key`에 `deny`를, 허용하고자 하면 `key`에 `allow`를 적는다. 이 때 대소문자를 구분하므로 주의해서 `key`를 주도록 한다. `value`에는 차단 또는 허용할 remote address나 remote host를 적는다. 여기서 remote host의 값은 `servletRequest` 객체에서 `getRemoteHost()` 메소드를 호출했을 때 리턴되는 값을 사용한다. 이에 대한 자세한 내용은 [ServletRequest의 getRemoteHost\(\) 메서드](#)를 참조한다.

`value`에 줄 문자열은 Java 정규표현식을 따른다. 정규표현식 규칙은 `java.util.regex` 패키지의 규칙을 따른다. `denyStatus`는 요청을 막는 경우에 적용할 status 코드이다. 설정하지 않은 경우의 기본값은 403이다.

아래의 예시는 로컬에서의 접근만 허용하는 예제이다.

remote address 밸브 추가 : `<jeus-web-dd.xml>`

```

<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
<context-path>/examples</context-path>
    <pipeline>
        <valve>
            <class-name>jeus.servlet.valve.RemoteAddressValve</class-name>
            <property>
                <key>allow</key>
                <value>127\.\0\.\0\.\1</value>
            </property>
            <property>
                <key>denyStatus</key>
                <value>403</value>
            </property>
        </valve>
    </pipeline>
</jeus-web-dd>

```

아래 예제는 remote host valve의 사용 예제이다. remote host의 hostname이 `www.tamx.co.kr`인 경우 접근을 거부하는 예제이다.

remote host 밸브 추가 : `<jeus-web-dd.xml>`

```

<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
<context-path>/examples</context-path>

```

```

<pipeline>
  <valve>
    <class-name>jeus.servlet.valve.RemoteHostValve</class-name>
    <property>
      <key>deny</key>
      <value>www\.\tmax\.\co\.\kr</value>
    </property>
  </valve>
</pipeline>
</jeus-web-dd>

```



key와 value는 대소문자를 구분하기 때문에 옵션을 줄 때 주의하도록 한다. 또한, deny와 allow를 동시에 주지 않도록 주의한다.

10.3.3. URL rewrite 옵션 설정

URL을 재작성하기 위해 설정하는 밸브를 의미한다. Tomcat의 RewriteValve를 참고해 구현했고, 같은 규칙을 따른다. url rewrite 기능의 목적은 외부에 자신이 설정한 도메인이 노출되지 않게 하거나, 다른 곳으로 redirect하게 하는 목적이 있다. rewrite 규칙은 rewrite.config란 파일에 별도로 작성한다. rewrite 규칙에 대한 자세한 내용은 [Apache Tomcat 8.5 Rewrite 문서](#)를 참조한다.

설정은 web-engine 및 context 단위로 할 수 있다. web-engine 단위로 하는 경우에는 DOMAIN/config에 한다. app 단위로 하는 경우에는 APP_HOME/WEB-INF 에 파일을 위치시키면 된다. JEUS가 제공하는 url rewrite 옵션을 적용하기 위한 클래스 이름은 jeus.servlet.valve.rewrite.RewriteValve이다.

rewrite 밸브 추가 : <jeus-web-dd.xml>

```

<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="9">
  <context-path>/examples</context-path>
  <pipeline>
    <valve>
      <class-name>jeus.servlet.valve.rewrite.RewriteValve</class-name>
      <property>
        <key>encoding</key>
        <value>utf-8</value>
      </property>
    </valve>
  </pipeline>
</jeus-web-dd>

```

key로 encoding을 줄 수 있으며 url rewrite 시에 사용할 인코딩을 지정한다. 지정하지 않는다면 JEUS의 요청 쿼리 인코딩을 적용하도록 되어 있다. JEUS의 인코딩과 rewrite.config의 인코딩, RewriteValve의 인코딩은 일치해야 한글 사용시 정상적으로 사용 가능하다.

JEUS의 query encoding은 다음과 같은 우선순위에 따라 결정된다.

1. domain.xml 내의 forced url encoding 설정
2. forced request encoding 설정

3. client override encoding 설정
4. url mapping 의 encoding 설정
5. default encoding 설정
6. 아무것도 설정되어있지 않은 경우엔 ISO-8859-1



절대 `jeus.servlet.request.6CompatibleSetCharacterEncoding` 옵션과 함께 사용해선 안된다. 해당 설정은 하위 호환성을 위해 제공하는 옵션으로 스펙과 다르게 동작하도록 만들어 인코딩 우선순위를 엉망으로 만든다.

10.4. 사용자 정의 밸브

JEUS 개발자 또한 밸브를 상속받아 구현할 수 있다.

밸브를 추가하기 위해선 JEUS의 `jeus.servlet.valve.ValveBase`를 상속 받아 구현한다. `ValveBase`를 상속받아 사용하기 위해선 `JEUS_HOME/lib/system`의 `jakarta.servlet-api.jar`와 `jeus-servlet-engine.jar` 라이브러리가 필요하다. 빌드한 이후에는 jar로 패키징해 `JEUS_HOME/lib/thirdparty`에 추가하면 된다. 이 때, jar파일 내에 `jakarta.servlet-api.jar`와 `jeus-servlet-engine.jar`가 들어가지 않도록 주의한다.

아래는 모든 http response header에 test test를 추가해서 내보내는 valve의 예제이다.

response header에 test:test를 추가하는 예제

```
package com.test.valve;

import jeus.servlet.valve.ValveBase;
import jeus.servlet.valve.ValveException;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Map;

public class TestValve extends ValveBase {
    private String test;

    @Override public void init() throws ValveException {
        Map<String,String> paramMap = getValveParameterMap();
        test = paramMap.get("test");
    }

    @Override public void invoke(HttpServletRequest req, HttpServletResponse resp) throws
    IOException, ServletException {
        //애플리케이션 실행 전
        getNext().invoke(req, resp);
        //애플리케이션 실행 후
        resp.setHeader(test,test);
    }
}
```

`init()` 메소드의 경우에는 context 단위에 설정된 경우 deploy될 때 `domain.xml`의 경우 가상호스트나 웹 엔진이

시작할 때 호출된다. `getValveParameter()` 메소드는 `jeus-web-dd.xml`이나 `domain.xml`에서 설정한 `property`의 `map`을 가져오는 메소드이다. `invoke` 메소드의 경우에는 요청이 들어왔을 때의 `valve`의 동작을 정의한다. 기본적인 구조는 서블릿 필터와 유사하다. `getNext().invoke()` 메소드는 필터 구현에서의 `chain.doFilter()` 메소드와 같은 역할을 한다. `getNext().invoke()` 메소드 호출 이전에는 `request`를 이용해 필터 작업을 수행하고, `response`를 통해 응답의 필터링 작업을 실행한다.

JEUS_HOME/docs/api/jeus-servlet의 `jeus.servlet.valve.ValveBase`에 대한 설명이 있다.



`init()` 메소드에서 아무런 동작을 하지 않더라도 구현은 해야 한다. `invoke()` 메소드 또한 `valve`의 동작을 정의하기 때문에 구현해야 한다. 또한, `invoke` 구현시에 `getNext().invoke()` 메소드를 호출하는 것을 잊어선 안된다. 요청의 처리는 항상 JEUS에서 정의한 마지막 밸브에서 하기 때문에, 다음 밸브로 넘어가지 못하는 경우에는 요청을 처리하지 못한다.

10.5. JEUS 제공 filter 설정

JEUS에서는 이용자 편의를 위해 필터 구현체를 제공한다.

10.5.1. samesite 제한 옵션 설정

`samesite`는 `context` 별로 설정할 수 있지만, `user-agent`를 식별해 다르게 동작할 수는 없다. 특정 클라이언트에서는 `samesite`를 처리하지 못하는 경우가 있어, JEUS에서 `user-agent`를 식별해 `samesite`를 삭제하는 필터를 제공한다.

`samesite` 제한 옵션 필터 추가 : `<web.xml>`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
  metadata-complete="false"
  version="5.0">

  <filter>
    <filter-name>NoSameSiteFilter</filter-name>
    <filter-class>jeus.servlet.filters.NoSameSiteFilter</filter-class>
    <init-param>
      <param-name>agent</param-name>
      <param-value>^Chrome/\w$</param-value>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>NoSameSiteFilter</filter-name>
    <servlet-name>NoSameSiteServletTest</servlet-name>
  </filter-mapping>

</web-app>
```

JEUS가 제공하는 samesite 제한 옵션에 사용할 클래스 이름은 jeus.servlet.filters.NoSameSiteFilter이다. init-param에 name에는 agent로 설정해 주어야 하며, value로 samesite를 삭제하고자 하는 user-agent의 정규 표현식으로 표현한다.

10.5.2. Forwarded Header 옵션 설정

RFC 7239에 정의되어 있는 Forwarded Header와 de facto("사실 상 표준") 헤더들을(ex. X-Forwarded)를 식별하여, jakarta.servlet.HttpServletRequest API에서 proxy가 아닌 client 정보를 얻어 올 수 있도록 하는 필터를 제공한다.

필터 클래스 이름은 jeus.servlet.filters.ForwardedFilter이고, 필터를 추가하면 자동으로 forwarded 헤더를 찾고 적용한다. 하지만, forwarded는 스펙상 "for, host, proto, by" 4가지만 찾아서 적용한다. 자세한 내용은 RFC 7239를 참고한다.

de facto 헤더도 같이 고려하기 위해서는 필터의 InitParameter로 찾고자 하는 헤더를 적용하면 된다. InitParameter의 key는 총 6가지가 있는데, Forwarded-For, Forwarded-Host, Forwarded-Proto, Forwarded-By, Forwarded-Port, Forwarded-Server 가 있다. InitParameter value에는 적용하고자 하는 헤더들을 콤마(,)로 구분해서 넣으면 된다. 예를 들어, 아래와 같이 정의할 수 있다.

```
<filter>
  <filter-name>ForwardedFilter</filter-name>
  <filter-class>jeus.servlet.filters.ForwardedFilter</filter-class>
  <init-param>
    <param-name>Forwarded-For</param-name>
    <param-value>X-Forwarded-For,Proxy-Client-IP,WL-Proxy-Client-IP,HTTP_CLIENT_IP,HTTP_X_FORWARDED_FOR</param-value>
  </init-param>
</filter>
```

적용되는 우선순위는 Forwarded 헤더가 가장 높고, 그 다음에는 param-value에 적힌 순서대로 우선순위가 높다. 그리고, key와 전혀 관련없는 헤더들을 적용할 경우(Forwarded-For에 host와 관련된 헤더들을 넣을 경우) API를 호출할 때 잘못된 값이 나올 수 있으니 주의해야 한다.

필터에서 필터링된 후보 헤더와 값을 확인하기 위해서는, 다음 아래의 값으로 HttpServletRequest.getAttribute() API를 호출하면 확인 가능하다. 리턴값은 Map<String, List<String>>으로 첫번째 String은 헤더 이름이고, List<String>은 헤더 값들이다.

```
jeus.servlet.filters.ForwardedFilter.For
jeus.servlet.filters.ForwardedFilter.Host
jeus.servlet.filters.ForwardedFilter.Proto
jeus.servlet.filters.ForwardedFilter.By
jeus.servlet.filters.ForwardedFilter.Port
jeus.servlet.filters.ForwardedFilter.Server
```

관련된 HttpServletRequest의 API는 getRemoteAddr(), getServerName(), getServerPort(), getScheme(), isSecure() 이다.

Forwarded Header 옵션 필터 추가 : <web.xml>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
         metadata-complete="false"
         version="5.0">

  <filter>
    <filter-name>ForwardedFilter</filter-name>
    <filter-class>jeus.servlet.filters.ForwardedFilter</filter-class>
    <init-param>
      <param-name>Forwarded-For</param-name>
      <param-value>X-Forwarded-For,Proxy-Client-IP,WL-Proxy-Client-
IP,HTTP_CLIENT_IP,HTTP_X_FORWARDED_FOR</param-value>
    </init-param>
    <init-param>
      <param-name>Forwarded-Host</param-name>
      <param-value>X-Forwarded-Host</param-value>
    </init-param>
    <init-param>
      <param-name>Forwarded-Proto</param-name>
      <param-value>X-Forwarded-Proto, WL-Proxy-SSL</param-value>
    </init-param>
    <init-param>
      <param-name>Forwarded-By</param-name>
      <param-value>X-Forwarded-By</param-value>
    </init-param>
    <init-param>
      <param-name>Forwarded-Port</param-name>
      <param-value>X-Forwarded-Port</param-value>
    </init-param>
    <init-param>
      <param-name>Forwarded-Server</param-name>
      <param-value>X-Forwarded-Server</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>ForwardedFilter</filter-name>
    <servlet-name>ForwardedFilterServletTest</servlet-name>
  </filter-mapping>
</web-app>
```

JEUS가 제공하는 Forwarded Header 옵션에 사용할 클래스 이름은 jeus.servlet.filters.ForwardedFilter이다.

11. 따라하기

다음의 단계를 따라하면 JEUS를 간단하게 체험할 수 있다. 다음의 예에서는 JEUS의 서버명이 "server1"이라고 가정한다. 사용자는 이 서버명을 JEUS가 설치된 머신의 이름으로 변경해야 한다.

1. 시스템에 JEUS가 제대로 설치되었는지 여부와 경로 및 시스템 변수들이 적절히 설정되어 있는지 확인한다(특히, 시스템 경로에 "JEUS_HOME/bin/" 디렉터리가 포함되어 있는지 확인한다).



1. 본 안내서에서 언급하는 "JEUS_HOME"은 JEUS의 실제 설치 루트 디렉터리를 의미한다.

(예: "/home/user/jeus9")

2. JEUS 설치에 대한 자세한 내용은 "JEUS 설치 및 시작하기"를 참고한다.

2. MS(Managed Server) 내의 웹 엔진(서블릿 엔진)에 적어도 1개의 웹 커넥션이 설정되어 있어야 한다. 웹 커넥션 설정은 [웹 커넥션 관리](#)를 참고한다.

간단하게 [HTTP 리스너 설정](#)에서와 같이 포트가 '8088'로 설정된 동일한 HTTP 리스너를 추가하여 간단하게 실행해 볼 수 있다.

3. 설정이 완료되면 콘솔 화면을 실행하고 다음과 같이 명령을 입력하면 JEUS Launcher가 실행된다. 여기서 -u와 -p는 JEUS를 설치할 때 설정한 관리자 이름과 패스워드를 의미한다.

아래 예제에서는 관리자 이름과 패스워드를 'jeus'로 설정했다고 가정한다.

```
startMasterServer -domain domain1 -server server1 -u jeus -p jeus
```

4. 다른 콘솔 화면을 열고, 이번에는 '**jeusadmin -u jeus -p jeus**' 명령을 실행한다(여기서 -u, -p를 입력하지 않았다면 다시 한 번 'login -u jeus -p jeus'을 입력하여 로그인 할 수 있다).

```
$ jeusadmin
JEUS 9 Administration Tool
To view help, use the 'help' command.
offline>login -u jeus -p jeus
Attempting to connect to 127.0.0.1:9736.
The connection has been established to JEUS Master Server [adminServer] in the domain [domain1].
[MASTER]domain1.adminServer>
```

5. 다음과 같이 명령을 실행하면 웹 엔진을 모니터링하고 제어할 수 있는 명령어들의 목록이 출력된다.

```
[MASTER]domain1.adminServer>help -g Web
[ Web]-----
add-ajp-listener      Add AJP listener.
add-backup-webtob     Add backup WebtoB server.
add-http-listener     Add HTTP listener.
add-response-header   Add an HTTP response custom header.
```

add-tcp-listener	Add TCP listener.
add-tmax-connector	Add Tmax Connector.
add-valve	add a new valve configuration.
add-virtual-host	Add virtual host.
add-web-cookie-policy	Add the cookie policy configuration.
add-web-encoding	Add web engine charset encoding.
add-web-properties	Add web engine properties.
add-webtob-connector	Add the WebtoB Connector.
clear-thread-local	Change the on/off setting for clearing ThreadLocal
clear-web-statistics	Resets the web engine statistics.
list-session	Show session list sorted by idle time.
modify-jsp-engine	Modify JSP engine configurations.
modify-response-header	Modify the HTTP response custom header.
modify-session-configuration	Modifies the session configuration.
modify-tmax-connector	Modify the thread pool number of the tmax-connector.
modify-virtual-host	Modify the access log format of a virtual host dynamically. The access log must be enabled.
modify-web-cookie-policy	Modify the cookie policy configuration.
modify-web-encoding	Modify the web engine charset encoding configurations.
modify-web-engine-configuration	Modify some parts of the web engine configuration dynamically.
modify-web-listener	Modify the thread pool number of the web listener (http-listener, tcp-listener, orajp13-listener).
modify-web-properties	Modify web engine properties.
modify-webtob-connector	Modify the thread pool number of the webtob-connector.
notify-auto-scale	Notify completed.
reload-web-context	Forcibly reloads the servlet context.
remove-backup-webtob	Remove backup WebtoB server.
remove-response-header	Remove the HTTP response custom header.
remove-session	Remove time exceeded session.
remove-tmax-connector	Remove the tmax-connector.
remove-valve	Remove the valve configuration.
remove-valve-property	Remove the property of the valve.
remove-virtual-host	Remove virtual host.
remove-web-cookie-policy	Remove the HTTP cookie policy configuration.
remove-web-encoding	Remove the web engine charset encoding configurations.
remove-web-listener	Remove a web listener (http-listener, tcp-listener, or ajp13-listener).
remove-web-properties	Remove web engine properties.
remove-webtob-connector	Remove webtob-connector.
resume-web-component	Temporarily resumes the given component (servlet element of context or given web-connection name).
set-valve-property	Set property of the valve. If the corresponding key does not exist, a key=value pair is added to the valve.
show-request-processing-flow	Shows the request processing flow of mapped URL patterns and the specified host name.
show-session-configuration	Shows the session configuration.
show-session-server-backup-table	Shows the session server backup table

<code>show-web-engine-configuration</code>	Show web engine configurations, including the monitoring period and access-logs.
<code>show-web-statistics</code>	Shows the web engine statistics.
<code>show-webtob-connector</code>	Show the WebtoB Connector Information.
<code>suspend-web-component</code>	Temporarily suspends the servlet.
<code>precompile-jsp</code>	Precompile JSP files for a deployed web module. Connect to JEUS Master Server or a server.

To show detailed information for a command, use 'help [COMMAND_NAME]'.
ex) help connect

6. 웹 엔진 제어 명령어나 모니터링 명령어를 수행한다. 웹 엔진 명령어는 JEUS Reference 안내서의 "웹 엔진 관련 명령어"를 참고한다.

위의 과정에서 문제가 발생했다면 JEUS 환경 설정을 확인하고 올바르게 재설정해야 한다. 문제점의 원인을 좀 더 자세히 알아보려면 JEUS 관리자 콘솔 윈도우 로그에 남겨진 정보를 참고하는 것도 좋은 방법이다.



JEUS 환경설정에 대한 자세한 내용은 "JEUS 설치 및 시작하기"와 "JEUS Server 안내서"를 참고한다.