

# Jakarta Batch 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. 소개	1
1.1. Jakarta Batch 기본 개념	1
1.2. Jakarta Batch 구성 요소	1
1.2.1. JobOperator	2
1.2.2. Job	2
1.2.3. Step	3
1.2.4. ItemReader	4
1.2.5. ItemWriter	4
1.2.6. ItemProcessor	5
1.2.7. Checkpoints	5
2. Batch 스레드 풀 환경설정	6
2.1. 개요	6
2.2. DD 설정	6
2.2.1. 컴포넌트 DD 설정	7
2.2.2. Application DD 설정	7
2.3. 로깅	7
3. Jakarta Batch 예제	9
3.1. 개요	9
3.2. DataSource 설정	9
3.3. jeus-web-dd.xml 설정	10
3.4. Job 정의	10
3.5. JBatchServlet	11
3.6. ItemReader	12
3.7. ItemWriter	13
3.8. ItemProcessor	14

# 1. 소개

본 장에서는 Jakarta Batch의 개념과 구성 요소에 대해서 설명한다.

## 1.1. Jakarta Batch 기본 개념

배치 처리방식은 널리 사용되는 워크로드 패턴으로, 벌크 지향적이고 상호호환을 하지 않으며 백그라운드에서 실행되는 특징이 있다. 일반적으로 오래 실행되고 계산량이 많은 작업들에 적합하고, 병렬적으로 처리하거나 순차적으로 처리해야하는 작업들에 모두 이용 가능하다. 다양한 방식으로 배치 처리를 시작할 수 있는데, 예를 들면 ad-hoc, 스케줄, on-demand 방식으로 실행될 수 있다. 배치 애플리케이션들의 로깅, 체크포인팅, 병렬화 등에 대한 요구를 만족시키고, 배치 워크로드에 대해 관제할 수 있는 기능이 제공되어 작업의 게시, 중지/재시작같은 상호작용을 가능하게 한다.

JEUS에서는 스펙에서 제공하는 RI를 기반으로 동작하고, 작업이 실행될 스레드 풀을 사용자가 명세할 수 있도록 기능을 제공한다.

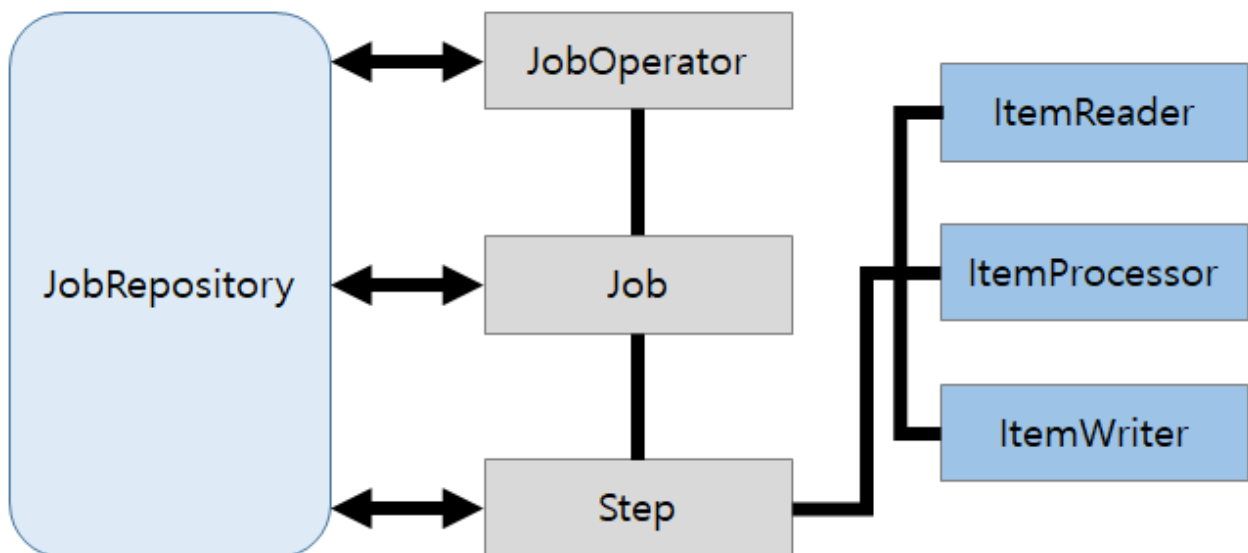


Jakarta Batch에 대한 상세한 내용은 스펙을 참조하기를 권한다.

## 1.2. Jakarta Batch 구성 요소

본 절에서는 Jakarta Batch에서 정의하고 있는 작업을 구성하는 요소들에 대한 개념을 간략하게 설명한다.

다음은 Jakarta Batch를 구성하는 핵심적인 요소들 간의 관계이다.



Jakarta Batch를 구성하는 핵심적인 요소들 간의 관계

사용자 애플리케이션에서 JobOperator를 통해서 Job을 제어(실행 혹은 중지)하고, Job은 수행할 작업에 대한 명세를 가지고 있다. Step은 Job의 하위에 속하며, 하나의 Job은 여러 Step으로 이루어질 수 있다. 마지막으로 step은 ItemReader, ItemProcessor, ItemWriter의 처리 모듈이 분리되어 있다. 이 모든 것은 Jakarta Batch 구현체 내의 JobRepository에 메타정보가 저장된다.

## 1.2.1. JobOperator

JobOperator는 Job의 처리에 대한 모든 상태 정보를 조회/제어할 수 있는 기능을 제공하고, 사용자 애플리케이션에서 배치 작업을 실행시켜주는 인터페이스 역할을 수행한다. 구체적으로 예를 들면 시작, 재시작, 종료에 대한 제어와 StepExecution을 가지고 오는 명령어를 제공하기도 한다.

## 1.2.2. Job

Job은 배치 프로세스 전체를 캡슐화한 정보이고 여러 Step으로 구성될 수 있고, 모든 Step을 전역적으로 관리하는 환경 설정을 할 수 있다. Job의 환경설정으로 Job의 이름, Step의 순서 정의, Job의 재시작 기능 설정 유무를 명시할 수 있다.

- JobInstance

JobInstance는 개념적인 Job의 실행을 의미한다.

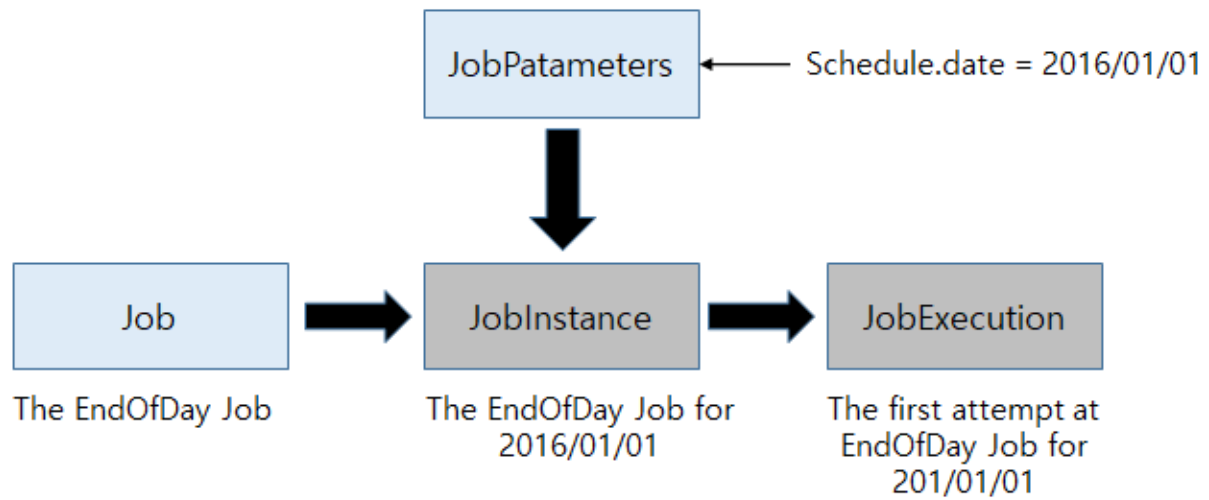
매일 특정 시간에 동작하는 Job을 정의했다고 가정하면 이 Job은 매일 특정 시간에 동작한다는 논리적인 JobInstance 하나를 가지게 된다. 만약에 1월 1일에 해당 Job이 실행되었다가 실패한 경우 다음 날이 되었을 때 실패했던 1월 1일의 JobInstance를 다시 실행한다. 또한, 1월 2일의 JobInstance도 새롭게 생성하여 실행한다. 따라서 각 JobInstance는 여러 번 실행될 수 있고, 여러 번 실행되는 주체를 JobExecution라고 한다. 실행할 때마다 JobExecution 인스턴스가 새롭게 생성된다. 새로운 JobInstance를 사용한다는 것은 처음부터 Job을 실행하겠다는 것을 의미하고, 이미 정의된 JobInstance를 사용하는 것은 Checkpoint 기능으로 처리해야 할 부분부터 시작하는 것이다.

JobInstance 자체는 데이터에 대해서 무지하다. 데이터에 대한 책임은 데이터를 item 단위로 읽어들이어서 적재하는 ItemReader 구현체에게 있다.

- JobParameters

JobInstance를 다른 JobInstance와 구별하기 위해서는 JobParameters를 이용해야 한다. 물론 다른 JobInstance에 동일한 JobParameters를 사용할 수도 있다. 구체적으로 JobParameters는 Java SE에서 제공하는 java.util.Properties 클래스이다.

예를 들면 1월 1일의 JobInstance는 JobParameters로 "schedule.date = 2016/01/01"를 가지고, 1월 2일의 JobInstance는 JobParameters로 "schedule.date = 2016/01/02"를 가진다. Job은 여러 JobInstance를 가질 수 있고, 각 JobInstance는 각자의 JobParameters로 구별될 수 있다. JobExecution은 하나의 JobInstance 내에서 여러 번 실행할 수 있다.



Job, JobInstance, JobParameters, JobExecution 간의 관계

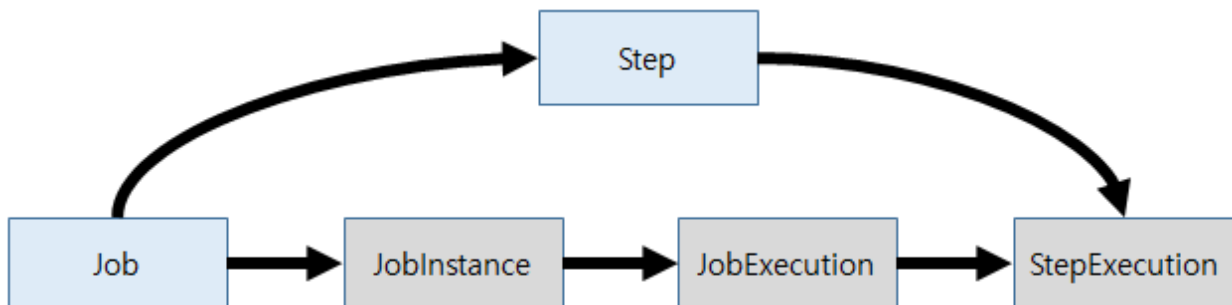
- JobExecution

Job의 실행을 한 번 시도하는 도구이다. 실행의 결과는 성공 혹은 실패가 될 수 있다. JobInstance는 해당 JobExecution이 성공적으로 끝났을 때만 성공한 것으로 간주된다. 예를 들면 매일 특정 시간에 동작하는 Job을 정의했을 때 해당 Job을 실제로 실행하는 주체는 JobExecution이 된다. 만약 해당 1월 1일에 수행한 JobExecution이 실패한다면, 다음 날 수행하는 시간에 동일한 1월 1일에 생성된 JobInstance에서 새로운 JobExecution을 정의하여 다시 작업을 수행하게 된다.

### 1.2.3. Step

Step은 연속된 단계로 처리해야 하는 배치 작업을 각 작업들 간에 관계를 독립적으로 캡슐화한 정보이다. 모든 Job은 여러 개의 Step으로 구성될 수 있고, Step은 실제 배치 처리에 대한 모든 정보와 제어하기 위해 필요한 정보들을 담고 있다. Step의 간단한 예로 파일을 읽어서 데이터베이스로 넣는 작업을 예로 들 수 있다. Job과 JobExecution의 관계처럼 Step도 StepExecution을 가지고 있다.

다음 그림은 Job과 Step의 관계를 나타낸다. Job과 Step이 one-to-many 관계이고, 각 step은 상황에 따라 StepExecution을 여러 번 만들어 실행시킬 수 있다. JobExecution과 StepExecution 관계 역시 one-to-many 관계이고, 하나의 Job이 상황에 따라 여러 StepExecution을 생성하여 실행하는 관계가 될 수 있다.



Jakarta Batch에서 Job과 Step의 관계

- StepExecution

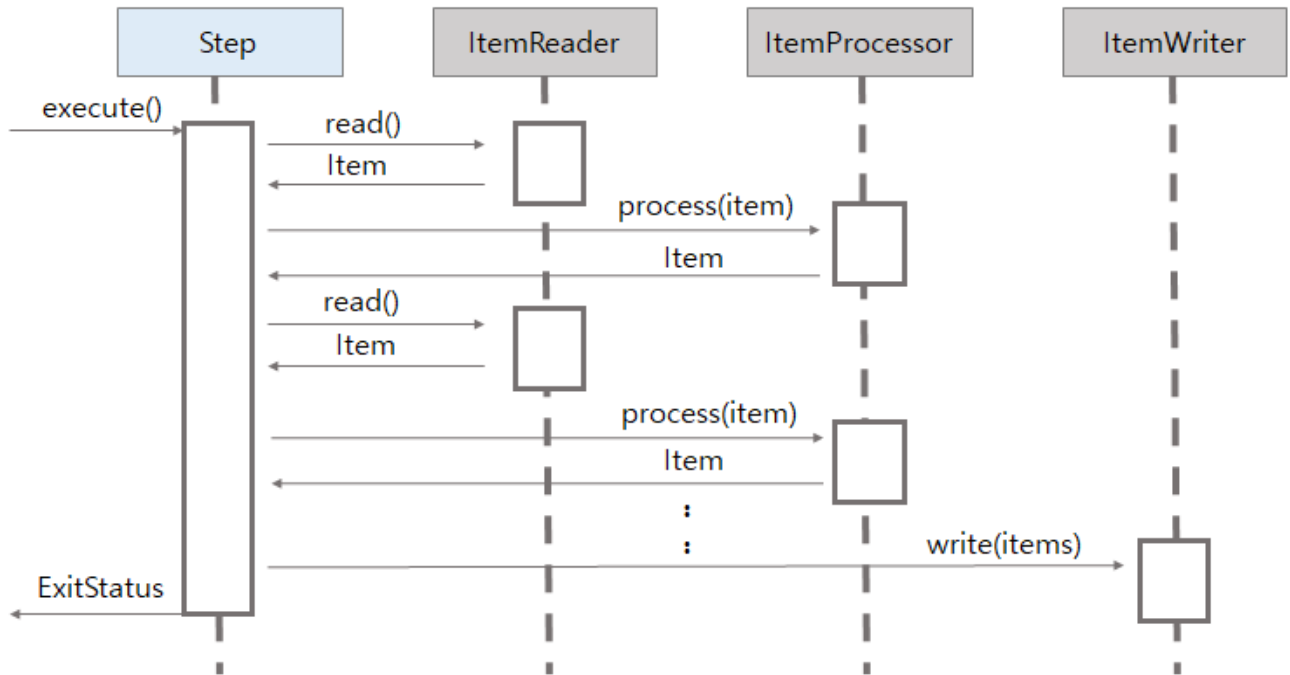
Step의 실행을 한 번 시도하는 도구이다. Step이 실행될 때마다 StepExecution이 생성이 된다. Step은 Chunk나 Batchlet으로 이루어지고, 서로 상호 배제(mutually exclusive)한 관계로 동시에 사용될 수 없다.



- Chunk

Chunk는 item-oriented한 배치 Step에 적절하며, reader-processor-writer 패턴으로 구현되고 트랜잭션의 scope으로 실행된다. 주기적으로 처리 과정을 Checkpoint로 기록한 후 새로운 트랜잭션을 실행한다.

다음은 chunk 기반의 item을 reader-processor-writer 패턴으로 처리하는 과정을 간략하게 나타낸 그림이다.



Jakarta Batch에서 Step을 처리할 때 Reader-Processor-Writer의 동작 과정

Step이 시작되면 ItemReader에서 입력 데이터를 item 단위로 읽어들이어 ItemProcessor에게 넘겨주고, ItemProcessor는 각 item을 처리한다. 이 과정을 반복하다가 처리 완료된 item 리스트를 ItemWriter에게 전달한다.

- Batchlet

Batchlet은 task-oriented한 배치 Step을 의미하고, item-oriented하지 않은 작업에 유용하게 이용될 수 있다. 구체적으로 예를 들면 파일을 전송하는 작업이나 특정 명령어를 실행하는 작업 등이 있다.

## 1.2.4. ItemReader

ItemReader는 한 스텝을 구성하는 item들을 읽어들이는 도구이다. 읽어들이는 item을 ItemProcessor에게 하나의 item씩 전달해주는 역할을 한다. 간단한 예로 파일을 열어서 처리해야 하는 작업이 있고 하나의 라인이 레코드 단위로 의미가 있다고 할 때, item을 하나의 라인으로 정의하여 전달할 수 있다.

뒤에서 설명할 Checkpoint 기능을 ItemReader에서 사용할 수 있는데 마지막으로 성공적으로 읽어온 위치를 Checkpoint로 기록을 해놓으면, 오류가 나거나 재시작이 되었을 때 Checkpoint부터 다시 시작할 수 있다. 위에서 예로든 파일 내의 레코드의 경우 마지막으로 읽어들이어진 라인수를 Checkpoint로 기록해 놓을 수 있다.

## 1.2.5. ItemWriter

ItemWriter는 한 chunk에 대한 결과물을 출력하는 도구이다. 현재 들어온 item에 대한 정보만 알 수 있으며, 다음에 들어올 입력에 대한 정보는 알 수가 없다. ItemProcessor를 거친 여러 items들이 리스트로 전달될 수 있다.

## 1.2.6. ItemProcessor

ItemProcessor는 Item을 실제 처리하는 도구이다. ItemReader가 하나의 item을 읽어 온 것을 ItemProcessor가 처리하는 로직을 수행하게 되고, 처리 결과를 ItemWriter가 출력하는 역할을 담당한다. 배치 애플리케이션 개발자는 Item에 대한 처리 로직을 ItemProcessor에 구현해서 수행한다.

## 1.2.7. Checkpoints

아주 오랜시간 동안의 처리시간이 필요한 거대한 데이터에 대한 배치 애플리케이션은 일반적으로 checkpoint/restart 기능을 요구한다.

Checkpoint는 StepExecution의 진행상태를 주기적으로 bookmark하여 나중에 재시작될 때 기록된 마지막 Checkpoint 부분부터 실행할 수 있도록 해주는 기능이다. Checkpoint를 할 때 lock을 잡고 기록하기 때문에 과도하게 Checkpoint 기능이 자주 동작할수록 전체 시스템의 성능에 크게 영향을 줄 수 있다. 따라서, Checkpoint 주기를 적절하게 설정하는 것은 성능에 중요한 튜닝 포인트이다.

## 2. Batch 스레드 풀 환경설정

본 장에서는 JEUS에서 Jakarta Batch에서 정의한 작업이 실행될 스레드 풀을 설정하는 방식에 대해서 기술한다.

### 2.1. 개요

스레드 풀은 Web, EJB의 DD에 설정하거나 그것을 포괄하는 Application DD에 설정할 수 있다.

일반적인 경우에는 Web에만(또는 EJB에만) 설정하거나 애플리케이션에만 batch-thread-pool을 설정하여 우선순위를 따질 것 없이 명시한 batch-thread-pool 정보를 기반으로 동작한다.



JEUS는 Web DD와 Application DD에 스레드 풀 설정이 모두 존재하거나 EJB DD, Application DD가 같이 존재하는 경우 **Web DD(또는 EJB DD)의 스레드 풀 설정을 기반으로 설정된다.**

### 2.2. DD 설정

DD에 batch-thread-pool에 대한 설정을 하지 않는 경우 기본값으로 설정된 스레드 풀에서 작업이 진행된다.

다음은 스레드 풀을 설정하는 항목에 대한 설명이다.

항목	설명
min	정의한 작업을 실행될 스레드 풀에서 관리하는 스레드 수의 최솟값이다. (기본값: 0)
max	정의한 작업을 실행될 스레드 풀에서 관리하는 스레드 수의 최댓값이다. (기본값: 0)
keep-alive-time	정의한 작업을 실행될 스레드 풀에서 관리하는 스레드의 수가 Max 이하이면서 Min 이상인 경우 설정된 시간 동안 아무 작업을 수행하고 있지 않은 스레드는 자동적으로 스레드 풀에서 제거된다.  스레드의 수가 0이면 제거하지 않는다. (기본값: 60000, 단위: ms)
queue-size	스레드 풀에서 정의한 작업들이 저장될 Queue의 크기를 지정한다. (기본값: 4096)

DD에서 스레드 풀에 대한 튜닝을 할 경우 DD 간의 우선순위에 따라 스레드 풀이 설정된다. 본 절에서는 각 DD별 설정 방법에 대해서 설명한다.

### 2.2.1. 컴포넌트 DD 설정

컴포넌트는 Servlet이나 EJB 등을 의미한다. 스레드 풀 설정들 간에 경쟁을 해야 할 경우 가장 높은 우선순위를 가지고 있다. jeus-web-dd.xml과 jeus-ejb-dd.xml에 설정한다.

다음은 /WEB-INF/jeus-web-dd.xml에 batch-thread-pool을 설정한 예이다.

컴포넌트 DD 설정 : <jeus-web-dd.xml>

```
<jeus-web-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="8.0">
  <batch-thread-pool>
    <min>10</min>
    <max>20</max>
    <keep-alive-time>60000</keep-alive-time>
    <queue-size>4096</queue-size>
  </batch-thread-pool>
</jeus-web-dd>
```

다음은 /META-INF/jeus-ejb-dd.xml에 batch-thread-pool을 설정한 예이다.

컴포넌트 DD 설정 : <jeus-ejb-dd.xml>

```
<jeus-ejb-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="8.0">
  <batch-thread-pool>
    <min>10</min>
    <max>20</max>
    <keep-alive-time>60000</keep-alive-time>
    <queue-size>4096</queue-size>
  </batch-thread-pool>
</jeus-ejb-dd>
```

### 2.2.2. Application DD 설정

다음은 /META-INF/jeus-application-dd.xml에 batch-thread-pool을 설정한 예이다.

Application DD 설정 : <jeus-application-dd.xml>

```
<application xmlns="http://www.tmaxsoft.com/xml/ns/jeus" version="8.0">
  <batch-thread-pool>
    <min>10</min>
    <max>20</max>
    <keep-alive-time>60000</keep-alive-time>
    <queue-size>4096</queue-size>
  </batch-thread-pool>
</application>
```

## 2.3. 로깅

Jakarta Batch의 로거는 JEUS 로거를 사용하지 않는다. 따라서 로거 설정도 JEUS 로거 설정으로는 동작하지 않는다. Jakarta Batch 로거는 Java 로거를 사용한다. 따라서 Jakarta Batch에 대하여 로깅을 하기 위해서는 Java 로거

설정을 해야 한다. 이에 대한 설정은 JEUS Server 안내서의 "프로퍼티 설정"을 참고한다.



Jakarta Batch 로거는 `com.ibm.jbatch` 패키지를 기준으로 로그 레벨 설정을 해야 한다.

## 3. Jakarta Batch 예제

본 장에서는 JEUS에서 Jakarta Batch를 활용한 간단한 애플리케이션에 예제를 설명한다.

### 3.1. 개요

본 예제에서는 JEUS에서 Servlet 요청에 대해 배치 Job을 수행하고, 수행 결과를 클라이언트에게 응답으로 주는 간단한 애플리케이션을 소개한다.

다음은 Jakarta Batch를 활용한 서블릿 예제의 War 파일 구조이다.

```
Example.war
|--META-INF
|   |--MANIFEST.MF
|--WEB-INF
|   |--[X]jeus-web-dd.xml
|   |--classes
|       |--META-INF
|           |--batch-jobs
|               |--[X]simple_file_batch.xml
|           |--task
|               |--[T]personList.txt
|           |--tmaxsoft
|               |--jbatch
|                   |--test
|                       |--[C]jBatchServlet.class
|                       |--handler
|                           |--[C]FileItemReader.class
|                           |--[C]LogWriter.class
|                           |--[C]Person.class
|                           |--[C]StringToPersonProcessor.class

* 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
```

### 3.2. DataSource 설정

Jakarta Batch 애플리케이션을 JEUS에서 사용하려면 반드시 Jakarta Batch 작업을 수행할 때 필요한 데이터소스가 정의되어 있어야 한다. 현재 JEUS에서는 Apache derby DB에서만 동작한다.

Linux의 경우 JEUS\_HOME/bin/startderby를 이용하여 derby를 실행할 수 있다. 데이터소스의 export name으로 "jdbc/batch"를 정의한다.



현재는 Derby만 정상적으로 지원되고 있다. Derby 외의 DB 지원은 정상적으로 검증이 이루어지지 않았으므로 사용을 권하지 않는다.

다음은 데이터소스가 정의된 domain.xml의 예이다.

데이터소스 설정 : <domain.xml>

```
<domain>
  <data-source>
    ...
    <database>
      <data-source-id>jdbc/batch</data-source-id>
      <data-source-class-name>
        org.apache.derby.jdbc.ClientConnectionPoolDataSource
      </data-source-class-name>
      <data-source-type>ConnectionPoolDataSource</data-source-type>
      <server-name>localhost</server-name>
      <port-number>1527</port-number>
      <database-name>derbyDB</database-name>
      <user>app</user>
      <password>app</password>
      <property>
        <name>ConnectionAttributes</name>
        <type>java.lang.String</type>
        <value>;create=true</value>
      </property>
    </database>
  </data-source>
</domain>
```

### 3.3. jeus-web-dd.xml 설정

Job을 실행시킬 스레드 풀에 대한 설정을 한다. jeus-web-dd.xml에서 batch-thread-pool에 대한 명세를 하지 않는 경우 default 값으로 설정된 스레드 풀이 생성된다. 해당 상세 설명은 [컴포넌트 DD 설정](#)을 참조한다.

### 3.4. Job 정의

Job을 명세하는 xml에 Job, Step, ItemReader, ItemWriter, ItemProcessor를 등록한다. 해당 Job을 정의한 xml 파일은 META-INF/batch-jobs/ 폴더 아래에 두어야 한다.



War에서 META-INF는 구조적으로 /WEB-INF/classes 하위에 있어야 동작한다.

다음은 /WEB-INF/classes/META-INF/batch-jobs/simple\_file\_batch.xml 파일의 설정 예이다.

Job 정의 : <simple\_file\_batch.xml>

```
<job id="demo" xmlns="http://xmlns.jcp.org/xml/ns/jakartaee" version="1.0">
  <step id="step1">
```

```

<chunk>
  <reader ref="tmaxsoft.bhkim.test.filebatch.FileItemReader">
    <properties>
      <property name="filePath" value="#{jobParameters['filePath']}" />
    </properties>
  </reader>
  <processor ref="tmaxsoft.bhkim.test.filebatch.StringToPersonProcessor" />
  <writer ref="tmaxsoft.bhkim.test.filebatch.LogWriter" />
</chunk>
</step>
</job>

```

## 3.5. JBatchServlet

Job에 대해서 설정한 xml을 JobOperator에게 전달한다. Job을 실행할 때 필요한 파라미터를 정의해서 같이 operator에게 전달할 수 있다.

다음 예제는 filePath를 key로 하는 프로퍼티로 처리할 데이터를 가지고 있는 personList.txt 파일의 경로를 넘겨주는 예제이다.

JBatchServlet 예제

```

@WebServlet("/JBatchServlet")
public class JBatchServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final String FILE_PATH = "META-INF/task/personList.txt";

    public JBatchServlet() {
        super();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        Properties jobParams = new Properties();
        URL personListURL = Thread.currentThread().getContextClassLoader().getResource(FILE_PATH);
        jobParams.setProperty("filePath", personListURL.getPath());

        JobOperator operator = BatchRuntime.getJobOperator();
        // xml file name without format
        long id = operator.start("simple_file_batch", jobParams);

        waitForEnd(operator, id);

        response.getWriter().println("File Batch is successfully precessed.");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }

    public static void waitForEnd(final JobOperator jobOperator, final long id) {
        final Collection<BatchStatus> endStatuses = Arrays.asList(BatchStatus.COMPLETED,
BatchStatus.FAILED);
        do {

```



```

        try {
            Thread.sleep(100);
        } catch (final InterruptedException e) {
            return;
        }
    } while (!endStatuses.contains(jobOperator.getJobExecution(id).getBatchStatus()));
}
}

```

FILE\_PATH에서 정의하고 있는 파일의 내용은 다음과 같다.

```

Name1,27,Computer Science
Name2,22,English Education
Name3,23,Electronic Engineering

```

## 3.6. ItemReader

FileItemReader 예제를 살펴보면 ItemReader를 구현하면서 4개의 함수 open, close, readItem, checkpointInfo를 구현해야 한다. ItemReader에서는 item 단위로 파일에 접근해서 처리해야 하고, 본 예제에서는 item의 단위를 파일의 한 라인으로 정의하고 있다.

open 함수의 매개변수로 Serializable을 넘겨주는데, 이것은 Checkpoint를 확인하는 기능으로 이전에 비정상적으로 종료되어 다시 시작하는 경우에 마지막으로 기록된 Checkpoint에서 실행되도록 도와주는 기능이다.



Jakarta Batch 애플리케이션을 작성할 때 Servlet, EJB 외에 순수 Batch 로직에서 injection이 동작하지 않으므로 주의해야 한다.

다음 예제에서는 readItem을 할 때마다 recordNumber를 기록해줬다가, 나중에 작업이 비정상 종료되어 다시 실행되었을 때 파일을 해당 recordNumber부터 시작할 수 있도록 건너뛰는 작업을 open에서 수행한다.

FileItemReader 예제

```

public class FileItemReader implements ItemReader {

    @Inject
    @BatchProperty
    private String filePath; // JobProperties in xml setting is injected to the field

    private BufferedReader reader = null;

    private int recordNumber;

    @Override
    public void open(Serializable checkpoint) throws Exception {
        if (filePath == null)
            throw new RuntimeException("Can't find any input");

        final File file = new File(filePath);
        if (!file.exists())
            throw new RuntimeException("A file '" + filePath + "' doesn't exist");
    }
}

```

```

        reader = new BufferedReader(new FileReader(file));

        if (checkpoint != null) {
            assert (checkpoint instanceof Integer);
            recordNumber = (Integer) checkpoint;

            // Pass over latest checkpoint record
            for (int i = 1; i < recordNumber; i++)
                reader.readLine();
        }
    }

    @Override
    public void close() throws Exception {
        if (reader != null)
            reader.close();
    }

    @Override
    public Object readItem() throws Exception {
        Object item = reader.readLine();

        // checkpoint line update
        recordNumber++;
        return item;
    }

    @Override
    public Serializable checkpointInfo() throws Exception {
        return recordNumber;
    }
}

```

## 3.7. ItemWriter

LogWriter 예제는 ItemWriter를 구현하면서 4개의 함수 open, close, writeItems, checkpointInfo를 구현한다. writeItem에서 처리해야할 item을 벌크(리스트)로 넘겨 받게 된다.



Jakarta Batch 애플리케이션을 작성할 때 Servlet, EJB 외에 순수 Batch 로직에서 injection이 동작하지 않으므로 주의해야 한다.

다음 예제에서는 간단하게 넘겨 받은 item 리스트에 대해서 내부에서 정의한 logger로 출력하는 작업만 수행한다.

ItemWriter 예제

```

public class LogWriter implements ItemWriter {
    private static final Logger logger = Logger.getLogger(LogWriter.class.getSimpleName());
    int writeRecordNumber;

    @Override
    public void open(Serializable checkpoint) throws Exception {

```

```

    }

    @Override
    public void close() throws Exception {
    }

    @Override
    public void writeItems(List<Object> items) throws Exception {
        // simply print passed item to logger
        for (Object o: items) {
            logger.info("writeItems > " + o.toString());
        }
    }

    @Override
    public Serializable checkpointInfo() throws Exception {
        return null;
    }
}

```

## 3.8. ItemProcessor

ItemProcessor는 ItemReader에서 읽어들이는 데이터를 가공하는 작업을 수행한다.



Jakarta Batch 애플리케이션을 작성할 때 Servlet, EJB 외에 순수 Batch 로직에서 injection이 동작하지 않으므로 주의해야 한다.

다음 예제에서는 읽어들이는 파일의 라인 문자열을 item 파라미터로 전달받았고, 문자열을 가공하여 Person 객체로 저장한다. 후에 특정 주기가 되면 가공된 정보들이 ItemWriter로 전달된다.

ItemProcessor 예제

```

public class StringToPersonProcessor implements ItemProcessor {

    @Override
    public Object processItem(Object item) throws Exception {
        // Parameter로 넘어온 item은 "name, age, department" 형식.
        final String[] line = String.class.cast(item).split(",");

        if (line == null || line.length != 3)
            return null;

        return new Person(line[0], Integer.parseInt(line[1]), line[2]);
    }
}

```

Processor 내에서 사용할 Person 객체는 다음과 같이 간단하게 정의된다.

Person 객체

```

public class Person {

```

```
private String name;
private int age;
private String department;

public Person(String name, int age, String department) {
    this.name = name;
    this.age = age;
    this.department = department;
}

@Override
public String toString() {
    return name + "," + age + "," + department;
}
}
```