



A Quick Guide To Enterprise Batch Processing With Jakarta Batch



Contents

Guide Updated: **April 2024**

Introduction	3
Jakarta Batch	3
Batch Job.....	3
Step.....	5
Chunk.....	5
Reader.....	6
Processor.....	6
Writer.....	7
Configuring the Chunk: Size Matters.....	8
Error Handling in Chunks.....	8
Retrying After Failures.....	9
Checkpointing for Consistency.....	9
Optimizing Performance.....	10
Task-Oriented Processing with Batchlets.....	10
Introducing Batchlets.....	10
Defining a Batchlet.....	10
Monitoring Batch Jobs.....	11
Job Operator API.....	11
Listeners.....	12
Metrics.....	14
Managing the Lifecycle of Batch Jobs.....	15
Exception Handling.....	15
Job Restartability.....	15
State Management.....	15
Understanding Checkpoints.....	15
When Custom Checkpointing is Necessary.....	16
Implementing Custom Checkpoints.....	16
Important Considerations.....	17
Transaction Management in Jakarta Batch.....	17
Key Optimization Techniques.....	18
Practical Example.....	18
Summary	19

Introduction

Batch processing is an integral part of enterprise applications, as such runtimes can support reading, processing and storing large volumes of data. For example, inventory processing, payroll, report generation, invoice/statement generation, data migration and data conversion are all tasks that can benefit from batch processing.

Batch processing typically involves the division of data loads into smaller "chunks," which are subsequently broken down into even smaller units. Through this systematic approach, each unit of data is processed individually, without requiring human intervention. facilitating streamlined execution of batch processing tasks.

As a result, the handling of significantly large volumes of data is streamlined, improving efficiency and speed. Batch processing can also be parallelized to take advantage of the hardware capabilities of the underlying computer. This quick guide will show you how to create batch processing tasks on the Jakarta EE Platform. By the end of the document, you will have a solid foundation in using the Jakarta Batch specification to create all kinds of batch jobs.

Jakarta Batch

On the Jakarta EE platform, Jakarta Batch is the standard specification for creating any batch processing tasks, whether they are simple or highly sophisticated. To create a batch job, you should first understand how Jakarta Batch is structured. Let's start by looking at the various components.

Batch Job

A batch job in Jakarta Batch is an encompassing instruction that contains everything to run a given batch task. For instance, the task for a bank may be sending account statements to clients at the end of every week. This task can be encapsulated, or described in a Jakarta Batch job. This job will have a name, e.g. MailClientStatements, the steps needed to get the job done, whether some exceptions should be skipped, any listeners, job level properties and anything that the job needs to run. Remember a typical batch job runs end-to-end with no human intervention. This means that a job should describe in advance everything that the given batch job will need.

A batch job is described in XML using the Jakarta Job Specification Language. The MailClientStatements job can be described as in the following XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="mail-client-statements" xmlns="https://jakarta.ee/xml/ns/jakartaee"
    version="2.0">
  <properties>
    <property name="propertyName" value="propertyValue"/>
    <property name="anotherProperty" value="anotherValue"/>
    <property name="defaultFileType" value="pdf"/>
  </properties>
  <listeners>
    <listener ref="jobLevelListener"></listener>
  </listeners>

  <step id="computeBalance" next="createFile">
    <properties>
      <property name="firstStepProperty" value="firstStepValue"/>
    </properties>
    <listeners>
      <listener ref="firstStepListener"></listener>
    </listeners>
    <chunk>
      <reader ref="myReader"></reader>
      <processor ref="myProcessor"></processor>
      <writer ref="myWriter"></writer>
    </chunk>
  </step>

  <step id="createFile">
    <batchlet ref="myBatchlet"></batchlet>
  </step>

  <step id="mailStatement">
    <batchlet ref="mailerBatchlet"></batchlet>

    <end on="COMPLETED"/>
  </step>
</job>
```

A job descriptor, as shown above, is first identified by an id. In this example, this is mail-client-statements. This is the unique ID of the job across all other batch jobs. The XML file can be called anything. But the convention is to call the file by the name of the job, so in this case, the XML file containing

the above job descriptor is in a file called `mail-client-statements.xml`. For applications packaged as WAR files, the job descriptor files are found under `WEB-INF/classes/META-INF/batch-jobs` and `META-INF/batch-jobs` for applications packaged as JAR files.

A job can have job-level or global properties that will be available to all units specified in the job. In the above example, the `job-level` properties element defines three properties, `propertyName`, `anotherProperty` and `defaultFileType`. These will be available to all specified units of the job. A batch job will be carried out primarily as a collection of one or more steps, each carrying out an atomic unit of the larger job.

Step

A step is a logical unit of work in a job. A step can be any self-contained unit of work that should be carried out as part of the larger job. In our mail customer statement job example, a given step can be one where client balances are computed on a specified date. Another step can be reading the customer statement for a given period and converting it to a PDF file. Yet another step can be taking the created file in the previous step and mailing it to the customer. Another auxiliary step could be reading the same PDF file and uploading to some document storage service.

All these steps are self-contained units of work that can be described separately in a batch job. A step must have an ID and, optionally, a `next` attribute that specifies which step comes after the execution of the current step. A step can also have properties, which will be scoped just to artefacts in that step. A step can also have step specific listeners, which again, will be scoped to that particular step. The mail customer statement job above has three steps - `computeBalance`, `createFile` and `mailStatement` steps.

The `computeBalance` step has its `next` set to `createFile`. This means that when the execution of the `computeBalance` is completed, the batch runtime will traverse to executing the `createFile` step. The `computeBalance` step defines its own property, called `firstStepProperty`, with the value `firstStepValue`. This property will only be available to artefacts in this step. The step also defines a listener, called `firstStepListner`. For the actual processing of carrying out the unit of a job in a step, a step can have either a chunk or a task.

Chunk

A chunk is one of two atomic units of work that can be carried out within a step. The other one is a task (covered later). A chunk specifies a unit of work that can be carried out on a given number of items. Remember a batch job, at its most atomic level, processes batch items one at a time. A chunk is used to specify the two required, and one optional operation that can be carried out on a single item in the batch processing. These are `reader`, `writer` and optionally, `processor`. The `computeBalance` step in the mail client statement batch description above has a chunk that specifies all three operations. A chunk can have a number of attributes that configure how the batch runtime manages each instance of the chunk. For now, let's look at the three operations that can be carried out in a chunk, starting with the `reader`.

Reader

A reader, in Jakarta Batch, is a Java class that implements the `jakarta.batch.api.chunk.ItemReader` interface. This interface has four methods that must be implemented. However, if the given batch job does not need all four methods, the reader can extend the `jakarta.batch.api.chunk.AbstractItemReader` class, which has empty placeholder implementations for three of the methods. By extending this class, a given reader will only have to implement the `readItem()` method from the `ItemReader` interface. The `MyReader` class for the mail client statement batch is shown below.

```
@Named
public class MyReader extends AbstractItemReader {

    @Override
    public Object readItem() throws Exception {
        return null;
    }
}
```

The `@Named` Jakarta CDI qualifier makes this class available for reference in the Expression Language used to define the batch job. By default, without specifying any name, the `@Named` annotation makes the annotated class available in the EL space as `myReader`, as used in the job description above. The `readItem` method is used to read a unit of whatever data that is being processed. In our mail client statement batch job, the reader can be used to read the client statement for each client. A reader reads a single item, one at a time. By default, the method returns `java.lang.Object`. You can change this to any concrete Java type in your application.

The reader can be implemented to read the batch items from anywhere - databases, files (JSON, XML etc and transformed), other modules or microservices as well as externally from another system. The batch runtime doesn't really prescribe where the data will come from. It only specifies that an item should be returned. It will keep calling the `readItem` method, as long as the method returns a non-null value. A null value signals the end of items to read to the batch runtime, indicating there is no longer the need to call the reader to read anything.

Where does the returned item go, you ask? Depending on how you configured your chunk, it can go to the processor, or straight to the writer. In the mail client statement job, it goes to the processor.

Processor

A processor in Jakarta Batch is a Java class that implements the `jakarta.batch.api.chunk.ItemProcessor`. This interface has a single method, `processItem(Object item)`, that takes the item returned from the reader. Remember the reader returns one batch item at a time. This item is

automatically passed as a parameter to the `processItem` method. This method can then do any kind of business-specific processing on the returned item. In our mail client statements batch job, for example, the processor can compute the balance of a client based on the client account received. This balance can be based on a specific date range.

Jakarta Batch leaves the actual details of how a given item is processed to your business domain. All it does is take the item from the reader and hand it over to your processor for processing. The `MyProcessor` class for the mail client statement batch job description is shown below.

```
@Named
public class MyProcessor implements ItemProcessor {
    @Override
    public Object processItem(Object item) throws Exception {
        return null;
    }
}
```

Similar to the reader, the processor also returns a unit of the processed item. As stated above, the processor is free to "process" the received item from the reader according to the business domain. Also, like the reader, the processor can return null to signify to the batch runtime that the given item as received from the reader should not be part of the next steps of the batch processing. This allows the processor to filter out items according to the business domain requirements.

For instance, the mail client statement batch processor can filter out client accounts that have not had any banking transaction in the last six months. The processor can return null for a received client account that falls within that category. Alternatively, the processor can return a File object for each client with transactions on their account in the last three months, returning null for clients that didn't have any. Where does the returned item from the processor go? It goes to the writer.

Writer

A writer in Jakarta Batch is any Java class that implements the `jakarta.batch.api.chunk.ItemWriter` interface. This interface, similar to the `ItemReader` interface, has four methods. But for brevity, a writer can extend the `jakarta.batch.api.chunk.AbstractItemWriter` class instead. This class only requires the implementation of the `writeItems(List<Object> items)` method. The `MyWriter` used in the mail client statement batch description is shown below.

```
@Named
public class MyWriter extends AbstractItemWriter {
    @Override
    public void writeItems(List<Object> items) throws Exception {

    }
}
```

The `writeItems` method takes a list of objects. These objects represent a collection of the concrete Java type passed from the reader to the processor and ultimately to the writer. But why does the `writeItem` method take a list? The batch runtime is designed to read a single item, process it and then collect all the items into a list up to a predefined count. When this count is hit, the list is passed to the writer. This list could be empty if the processor filters out all items, though this is not very common.

The writer can do different types of “writing” activities within the given business context. In the mail client statement, if the processor returns a `File` object for each processed client, the writer receives a collection of `Files` for “writing,” which could mean uploading said files to a given storage area.

Configuring the Chunk: Size Matters

One of the critical configurations of a chunk is its size, as this determines how many items the batch job processes before sending them to the writer. It's essential to understand that the right chunk size can significantly impact the performance of your batch job. If the size is too small, you could encounter inefficient use of computing resources. If it's too large, memory constraints or transaction timeouts could become a problem.

The following XML snippet illustrates how you might specify a chunk size in your job XML:

```
<chunk checkpoint-policy="item" item-count="100">
  <reader ref="myItemReader" />
  <processor ref="myItemProcessor" />
  <writer ref="myItemWriter" />
</chunk>
```

In this example, `item-count="100"` specifies that the job processes 100 items before invoking the writer. Knowing the ideal chunk size comes down to you measuring and finding out the average item number to expect, based on your workload.

Error Handling in Chunks

Error handling is another crucial aspect of chunk configuration. In batch processing, it's not uncommon to encounter a situation where a particular item fails to process due to a data issue or a transient system error. Jakarta Batch provides mechanisms to handle such errors gracefully.

You can specify a `skippable-exception-classes` element in the chunk to define which exceptions should not cause the job to fail but rather skip the problematic item:


```
<chunk>
  <skippable-exception-classes>
    <include class="jakarta.persistence.NoResultException"/>
  </skippable-exception-classes>
</chunk>
```

In this setup, if a `NoResultException` is thrown, the item is skipped, and the job continues, processing the next item.

Retrying After Failures

Sometimes, failures are not due to the item itself but rather temporary issues, like a network outage. Jakarta Batch allows for retrying such items:

```
<chunk>
  <retryable-exception-classes>
    <include class="java.net.SocketTimeoutException"/>
  </retryable-exception-classes>
</chunk>
```

Here, if a `SocketTimeoutException` occurs, the job will retry processing the item before deciding if it can't be processed.

Checkpointing for Consistency

Checkpointing is a strategy to ensure that a job can recover from a failure without having to start over from the beginning. By default, the checkpoint occurs after each chunk (defined by the `item-count``). However, you can also use a custom checkpoint policy if your business logic requires it:

```
<chunk checkpoint-policy="custom" item-count="100">
</chunk>
```

This level of control can be crucial when dealing with large datasets, where restarting a job from the beginning can be time- and resource-intensive.

Optimizing Performance

Consider the transactional behavior and the impact on performance. Using a persistent step-scoped or job-scoped data repository can minimize transaction times and optimize the performance of your batch job.

For instance, using an in-memory database for intermediate processing steps can drastically reduce the I/O time, making the chunk processing much faster.

Task-Oriented Processing with Batchlets

Up until now, our focus has been on chunks, which are ideal for iterative processing of datasets using the read-process-write pattern. However, batch processing is not always about dealing with large volumes of data that need to be processed in an iterative manner. Sometimes, the requirement is to perform a one-off task that doesn't fit into the chunk model. This is where the concept of a batchlet becomes crucial.

Introducing Batchlets

A batchlet is a specialized component within the Jakarta Batch framework designed for tasks that require a single execution step. It is a Java class that implements the `jakarta.batch.api.Batchlet` interface and is particularly suited for non-iterative operations, such as performing clean-up, executing a standalone script or initiating a single data migration task.

Defining a Batchlet

In the context of a batch job, you define a batchlet operation as a step in your job XML. Here's how you can declare a batchlet-based task:

```
<step id="cleanupResources">
  <batchlet ref="myResourceCleanupBatchlet"/>
</step>
```

In this example, `myResourceCleanupBatchlet` would be a Java class like the following that implements the `Batchlet` interface, which is tasked with executing the cleanup when this step is run.

```
@Named("myResourceCleanupBatchlet")
public class ResourceCleanupBatchlet implements Batchlet {

    @Override
    public String process() throws Exception {
        // Logic to clean up resources
        System.out.println("Cleaning up temporary files...");
        // ... (code to delete temporary files)

        System.out.println("Closing database connections...");
        // ... (code to close database connections)

        return "COMPLETED"; // Or "FAILED" if there were errors
    }

    @Override
    public void stop() throws Exception {
        // Optional. Add additional cleanup actions during cancellation
    }
}
```

Monitoring Batch Jobs

To ensure that your batch jobs run smoothly, effective monitoring is essential. Jakarta Batch offers several tools to keep track of job execution.

Job Operator API

The Job Operator API is a powerful feature that allows you to control the batch job lifecycle programmatically. With it, you can start, stop and restart jobs, as well as inquire about their current statuses. This API can be seamlessly integrated with your application's monitoring systems, providing high levels of control and visibility. For example, the JobOperator could be used as follows:

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
long executionId = jobOperator.start("myBatchJob", null); // Start the job
JobExecution jobExecution = jobOperator.getJobExecution(executionId);

System.out.println("Job Status: " + jobExecution.getBatchStatus());
```

Listeners

Jakarta Batch listeners provide a way to hook into various events throughout the lifecycle of a job or step. They enable you to capture vital information, trigger actions or integrate with external monitoring systems. Jakarta Batch provides several types of listeners.

Types of Listeners

- Job Listeners: Receive notifications of job-level events (before and after job execution, on job failure etc.).
- Step Listeners: Respond to events within a step (before and after a step, on step failure etc.).
- Reader Listeners: React to events related to the ItemReader (before and after read, on read error).
- Processor Listeners: Similar to reader listeners, but for the ItemProcessor.
- Writer Listeners: Receive event notifications for the ItemWriter.

Implementing Custom Listeners

To create a custom listener, you implement one or more of the listener interfaces from the `jakarta.batch.api.listener` package. For example, a job listener:

```
public class MyJobListener implements JobListener {
    @Override
    public void beforeJob() throws Exception {
        // Logic to perform before job starts
    }

    @Override
    public void afterJob() throws Exception {
        // Logic to perform after job completes
    }
}
```

Attaching Listeners to Jobs and Steps

You associate your listeners with a job or step in your job XML:

```
<job ...>
  <listeners>
    <listener ref="myJobListener" />
    <listener ref="myStepListener" />
  </listeners>
  ...
</job>
```

Advanced Use Cases

Let's look at some use cases for advanced listener techniques.

Detailed Auditing and Logging:

```
public class AuditStepListener implements StepListener {
    @Override
    public void beforeStep() throws Exception {
        logger.info("Step {} starting...", stepContext.getStepName());
    }

    @Override
    public void afterStep() throws Exception {
        logger.info("Step {} completed, items processed: {}",
            stepContext.getStepName(), stepContext.getMetrics().
getReadCount());
    }
}
```

Error Notifications and Retries:

```
public class ErrorNotificationListener implements WriteListener {
    @Override
    public void onWriteError(Exception ex) throws Exception {
        if (isRetryableError(ex)) {
            retryService.scheduleRetry(failedItems);
        } else {
            notificationService.sendFailureAlert(ex);
        }
    }
}
```

Integration with Monitoring Systems:

```
public class MetricsListener implements JobListener {
    @Inject
    private MetricsService metricsService;

    @Override
    public void afterJob() throws Exception {
        metricsService.recordJobExecution(jobContext.getJobName(), jobContext.
getExitStatus());
    }
}
```

Important Considerations

- Performance: Design listeners to be efficient. Avoid heavy processing that might slow down the batch job.
- Error Handling: Implement robust error handling within your listeners to prevent failures in the monitoring logic from disrupting the job.
- Contextual Information: Listeners have access to rich metadata via JobContext and StepContext objects. These can offer valuable insights into job state and execution metrics.

Metrics

Jakarta Batch provides built-in metrics that provide an overview of your batch jobs' performance. These metrics track data such as the number of items processed, skipped, or retried. They can be accessed as follow:

JMX (Java Management Extensions): If you have a JMX-compatible monitoring tool, you can configure your application runtime like Payara Server to expose Jakarta Batch metrics through JMX. This allows you to integrate the metrics with your monitoring dashboards.

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
long executionId = ...; // Get the execution ID of the job
JobExecution jobExecution = jobOperator.getJobExecution(executionId);
List<StepExecution> stepExecutions = jobExecution.getStepExecutions();
for (StepExecution stepExecution : stepExecutions) {
    for (Metric metric : stepExecution.getMetrics()) {
        System.out.println(stepExecution.getStepName() + " - " + metric.
getType() + ": " + metric.getValue());
    }
}
```

The common metric types available out of the box are:

- READ_COUNT: The total number of records read.
- WRITE_COUNT: The total number of records written.
- COMMIT_COUNT: The number of transaction commits.
- ROLLBACK_COUNT: The number of transaction rollbacks.
- READ_SKIP_COUNT: The number of records skipped during the reading process.
- WRITE_SKIP_COUNT: The number of records skipped during the writing process.

Managing the Lifecycle of Batch Jobs

Beyond starting and monitoring, effectively managing a batch job's lifecycle includes robust exception handling, job restart capabilities and state management.

Exception Handling

Jakarta Batch allows for fine-grained control over how your jobs handle exceptions. By specifying which exceptions should prompt a job to stop or cause a rollback, you can ensure that your batch jobs are resilient in the face of unexpected conditions.

Job Restartability

Making jobs restartable is particularly beneficial for long-running processes. If a job fails, it can be restarted from the last successful checkpoint rather than from the beginning, saving time and resources. You can enable this feature by setting the restartable attribute in the job XML:

```
<job id="myLongRunningJob" restartable="true">
</job>
```

State Management

Maintaining the state across job executions is very important, especially when jobs are interrupted or deal with partial processing. Jakarta Batch provides execution contexts that can persist state information, allowing for continuity when a job is restarted.

Understanding Checkpoints

By default, checkpoints occur in Jakarta Batch after each chunk (determined by the item-count). However, for particularly complex or long-running jobs, this automatic checkpointing may not always be the most optimal or robust strategy. This is where custom checkpoints come into play.

When Custom Checkpointing is Necessary

Consider these scenarios where a custom checkpoint policy might be beneficial:

- **Large Chunk Sizes:** If your chunks process huge datasets, the automatic checkpoint after each chunk might lead to excessive overhead or performance issues due to the serialization of state.
- **Non-Idempotent Operations:** When processing steps cannot be safely repeated without side effects, a custom checkpoint allows more granular control over exactly when the state is saved.
- **Fine-Grained Recovery:** In case of unexpected job interruptions, custom checkpoints give you greater flexibility to resume precisely where the job failed, minimizing wasted processing.

Implementing Custom Checkpoints

You'll need to implement a Java class extending the `jakarta.batch.api.checkpoint.CheckpointAlgorithm` interface. The primary method to override is `isCheckpoint()`, where you'll define the logic determining when a checkpoint should occur.

A custom algorithm can implement checkpoints based on a specific time interval or after successfully processing a defined number of records.

In your job XML, modify the chunk element to reference your custom algorithm:

```
<chunk checkpoint-policy="custom">
    <checkpoint-algorithm ref="timeBasedCheckpointAlgorithm"/>
    ...
</chunk>

public class TimeBasedCheckpointAlgorithm implements CheckpointAlgorithm {

    @Inject
    private JobContext jobContext;

    private long checkpointInterval;
    private long lastCheckpointTime;

    @Override
    public boolean isCheckpoint() throws Exception {
        if (checkpointInterval == 0) { // Initialize on first call
```



```
        checkpointInterval = Long.parseLong(jobContext.getProperties().
getProperty("checkpointInterval"));
        lastCheckpointTime = System.currentTimeMillis();
    }

    long currentTime = System.currentTimeMillis();
    if (currentTime - lastCheckpointTime >= checkpointInterval) {
        lastCheckpointTime = currentTime;
        return true;
    } else {
        return false;
    }
}
}
```

In this example, the algorithm checks if the elapsed time since the last checkpoint exceeds a configurable interval (`checkpointInterval` in job properties).

Important Considerations

- **Performance Overhead:** Design custom checkpoint algorithms carefully to avoid excessive overhead—the more frequent the checkpoints, the more performance can be affected.
- **State Persistence:** Ensure that the job context information you need for restarts is being serialized and persisted correctly.
- **Balance:** Strike a balance between checkpoint frequency and the cost of job restarts in the event of failures.

Transaction Management in Jakarta Batch

Before we dive into advanced optimization in Jakarta Batch, let's do a quick recap on how transactions work in Jakarta Batch:

- **Implicit Transaction Scope:** Each chunk or batchlet execution implicitly runs within its own transaction. These transactions are typically container-managed, e.g. JTA transactions if you are running within an application server.
- **Commit/Rollback:** The batch runtime commits the transaction at the end of a successful chunk or batchlet step. If an exception occurs, the transaction is rolled back.

Key Optimization Techniques

1. Transaction Boundaries:

- Carefully consider whether each chunk or batchlet truly requires its own transaction. Analyze if steps can be logically grouped to reduce individual transaction overhead.
- For read-heavy steps, explore alternatives like read-only transactions or relaxing transaction isolation levels if your business logic permits.

2. Batching Writes:

- Accumulate changes within a step and perform database writes in batches instead of individual operations. This significantly reduces transaction-related overheads.
- Be mindful of batch sizes. Excessively large batches can strain memory or lead to transaction timeouts.

3. Asynchronous Operations:

- If possible, offload non-critical updates or operations to asynchronous processes outside of the core batch job transactions. This shortens the duration of your main transactions, reducing the chance of timeouts while improving commit performance.

4. Data Source Configuration:

- Tune connection pools and database settings for optimal performance under high load.
- Consider database-specific optimizations, like bulk inserts or optimized indexing strategies.

5. Parallel Processing:

- Use Jakarta Batch's partitioning capabilities to split your workload across multiple threads or instances. These can distribute the transactional load and improve overall throughput, but they require careful coordination if data consistency is crucial.

Practical Example

Let's imagine a batch job that processes bank transactions. Here's an optimization approach:

- **Batching:** Instead of updating account balances with each transaction, accumulate changes in memory and flush them periodically in batches of 500 or 1000 updates.
- **Asynchronous Logging:** Send audit logs to a message queue for asynchronous processing, rather than including them in the main transaction.
- **Partitioning:** If the workload is sizeable, partition by account number ranges to run multiple workers in parallel.

Important Considerations

- **Business Logic:** Transaction boundaries must align with your business requirements for data integrity and consistency.
- **Error Handling:** Design robust error-handling mechanisms with retries and back-off strategies to handle transient failures without compromising the entire transaction.
- **Monitoring:** Measure transaction durations, commit/rollback rates and potential bottlenecks. Use these metrics to guide your optimization decisions.

Additional Tips

- **Database Choice:** For extremely high-volume scenarios, consider NoSQL databases or specialized data stores optimized for fast writes and scale-out capabilities.
- **In-Memory Caches:** Utilize caches, if feasible, to reduce database read load.

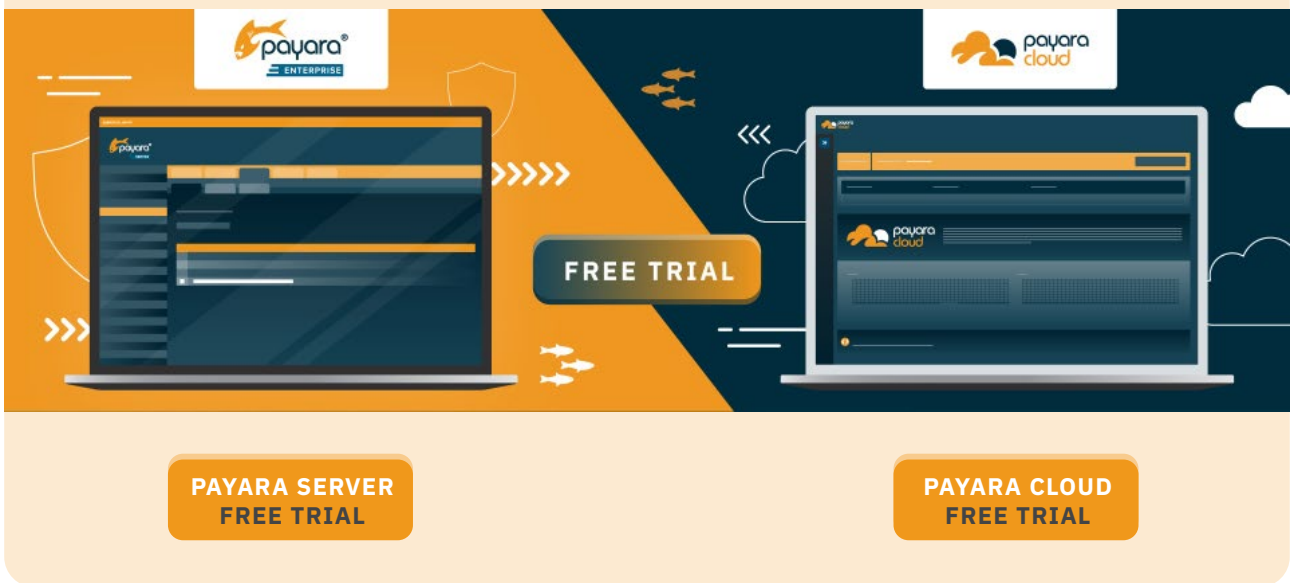
Summary

Throughout this guide, we've seen how Jakarta Batch simplifies the development of batch applications within the Jakarta EE environment. Key advantages include:

- **Standardization:** Jakarta Batch offers a well-defined specification for batch processing, promoting portability and maintainability of your batch jobs.
- **Flexibility:** From simple batchlets to complex, multi-step jobs, Jakarta Batch provides the components to model a wide range of batch processing scenarios.
- **Resilience:** Features like retries, exception handling and checkpoints help you build robust batch jobs that gracefully handle unexpected situations.
- **Monitoring:** Built-in metrics and integration with JMX provide visibility into your batch jobs' performance.

If you're ready to take your batch processing to the next level, start experimenting with Jakarta Batch by downloading [Payara Community](#). The power to streamline and optimize your background workloads is at your fingertips! Happy Coding!

Interested in Payara? Try Before You Buy



**PAYARA SERVER
FREE TRIAL**

**PAYARA CLOUD
FREE TRIAL**



sales@payara.fish



UK: +44 800 538 5490
Intl: +1 888 239 8941



www.payara.fish

Payara Services Ltd 2024 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ