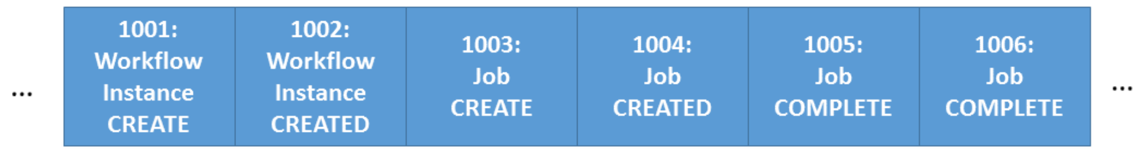
Camunda exporters

As Zeebe processes jobs and workflows, or performs internal maintenance (for example, Raft failover), it produces an ordered stream of records.

note

Exporters are not available in Camunda 8 Software-as-a-Service (SaaS).



Although clients can't directly inspect this stream, Zeebe can load and configure user-defined code, known as an **exporter**, to process each record.

An **exporter** provides a single entry point to handle every record written to the stream. Exporters can be used for various purposes:

- Persist historical data by pushing it to an external data warehouse
- Export records to visualization tools (e.g., zeebe-simple-monitor)

Zeebe loads exporters only if they are configured via the main Zeebe YAML configuration file.

Once configured, the exporter starts receiving records the next time Zeebe is restarted. Exporters are guaranteed to see only records produced after they're configured.

A reference implementation is available via the Zeebe-maintained Elasticsearch exporter.

Exporters reduce the need for Zeebe to store data indefinitely. Once data is no longer required internally, Zeebe queries its exporters to determine if it can be safely deleted. If so, it is permanently removed, reducing disk usage.

note

If no exporters are configured, Zeebe automatically deletes data when it's no longer needed. To retain historical data, you **must** configure an exporter to stream records to an external system.

All exporters—whether loaded from an external JAR or not—interact with the broker through the exporter interface.

Loading

Exporters are loaded during broker startup, before any processing begins.

The broker validates each exporter configuration during loading and will fail to start if:

- An exporter ID is not unique
- The exporter references a non-existent or inaccessible JAR
- The specified class does not exist or can't be instantiated
- The exporter throws an exception in its Exporter#configure method

This validation step allows exporters to perform lightweight configuration checks. During this phase, the context provides a partition ID value of Context#NULL_PARTITION_VALUE. At runtime, this will be replaced with the actual partition ID.

note

Zeebe instantiates the exporter for validation and then discards it. Exporters should avoid heavy computations during instantiation.

Metrics

The Micrometer MeterRegistry is available via the Exporter#configure(Context) method for exporters to record metrics:

```java
public class SomeExporter implements Exporter {
  @Override
  public void configure(final Context context) {
    // ...
    registry = context.getMeterRegistry();
    // ...
  }

  public void flush() {
    try (final var ignored = Timer.resource(registry, "meter.name")) {
      exportBulk();
    }
  }
}
```

When an exporter is validated, it receives an in-memory register that is discarded afterward.

note

Zeebe creates an isolated class loader for each JAR referenced in exporter configurations. If the same JAR is used by multiple exporters, they will share the same class loader.

This design allows different exporters to depend on the same third-party libraries without concerns about version conflicts or class name collisions.

System classes and those bundled with the Zeebe JAR are loaded via the system class loader.

Exporter-specific configuration is defined in the [exporters.args] nested map. This map is passed as a Map<String, Object> to the exporter's Exporter#configure(Configuration) method using the Configuration object.

Configuration takes place in two phases: once during broker startup and again each time a partition elects a new leader.

Processing

At any given time, there is exactly one leader node for each partition.

When a node becomes the leader for a partition, it starts an instance of the exporter stream processor.

This stream processor creates exactly one instance of each configured exporter and forwards every record on the stream to each exporter in sequence.

note

This means there is exactly one instance of each exporter per partition. For example, if you have four partitions and four processing threads, potentially four instances of your exporter may run simultaneously.

Zeebe guarantees **at-least-once** delivery semantics. This means that each record will be seen by an exporter at least once, but possibly more. Duplicate delivery can occur in scenarios such as:

- Reprocessing after Raft failover (i.e., leader re-election)
- Errors occurring before the exporter updates its position

To reduce duplicates, the stream processor tracks the position of the last successfully exported record for each exporter. Because the stream is an ordered sequence of records with monotonically increasing positions, tracking the position is sufficient. Exporters set this position once they can ensure the corresponding record was exported successfully.

note

Although Zeebe minimizes duplicate record delivery, exporters must be designed to handle duplicates. Export operations must be **idempotent**. This can be implemented within the exporter, but if exporting to an external system, it's recommended to handle deduplication there to minimize load on Zeebe. Refer to the exporter-specific documentation for implementation details.

Error handling

If an error occurs during the Exporter#open(Context) phase, the stream processor fails and is restarted. This may resolve transient issues automatically. In the worst case, no exporters will run until the errors are resolved.

If an error occurs during the Exporter#close phase, it is logged, but other exporters are still allowed to finish their work and shut down gracefully.

If an error occurs during record processing, the same record is retried continuously until the error no longer occurs. In the worst case, a single failing exporter can block all exporters for that partition. Currently, exporters are expected to implement their own retry and error-handling strategies—though this behavior may evolve in future Zeebe versions.

Performance impact

Each loaded exporter introduces some performance overhead. A slow exporter will slow down all other exporters for the same partition and, in extreme cases, may block a processing thread entirely.

To avoid performance bottlenecks, exporters should be kept as simple and lightweight as possible. Any heavy data transformation or enrichment should be delegated to external systems.

Purging

The data purge feature allows you to delete all historical (and runtime) data from your cluster. Therefore every exporter needs to implement the Exporter#purge method.

When a purge is happening, the Exporter#purge method is called. This method:

- Deletes all data exported so far.
- Is blocking and only returns when all data has been deleted.
- May be retried and therefore **must** be idempotent

When the purge cluster operation is executed, the following steps are taken:

- All nodes leave existing partitions resulting in a cluster with no partitions. This means all exporters are closed. At this point all runtime data is deleted.
- For all previously configured exporters, the following steps are executed sequentially for each exporter:
    - It is configured via Exporter#configure(Context)
    - The Exporter#purge method is called.
    - The exporter is closed via Exporter#close.

- Partitions are bootstrapped and nodes rejoin the partitions with the same configuration as before the purge.

note

Exporter#open(Context) is not called during the purge operation.

When an exporter is purged, it is expected to delete all data, but not schemas. In the case that an exporter exports to a database, only the records are deleted, not the tables themselves.

note

All resources required for purging need to be closed afterwards to avoid memory leaks.

Custom exporter to filter specific records

The exporter interface supports record filtering through the Context#RecordFilter interface.

- This interface provides methods to filter records based on record type, value type, and intent.
- Valid record types and value types can be found in the protocol definition, while intents are listed in the Intent enum class.

For example, you can implement a custom exporter that only exports records with:

- Record type: EVENT
- Value type: JOB
- Intent: CREATED

```
public class CustomExporterFilter implements RecordFilter {

 @Override
 public boolean acceptType(RecordType recordType) {
  return recordType == RecordType.EVENT;
 }

 @Override
 public boolean acceptValue(ValueType valueType) {
  return valueType == ValueType.JOB;
 }

 @Override
 public boolean acceptIntent(Intent intent) {
  return intent == JobIntent.CREATED;
  }
}
```

You can then set this filter in the Exporter#configure method of your custom exporter:

```
public class CustomExporter implements Exporter {

  // ...
  private Controller controller;

  @Override
  public void open(final Controller controller) {
    this.controller = controller;
    // ...
  }

  @Override
  public void configure(final Context context) {
    // ...
    context.setFilter(new CustomExporterFilter());
  }

  @Override
  public void export(final Record<?> record) {
    // ...
    // after handling the record, acknowledge the position
    controller.updateLastExportedRecordPosition(record.getPosition());
  }

  // ...
}
```

note

- After handling the record, you must acknowledge the position by calling controller.updateLastExportedRecordPosition(record.getPosition()). If the position is not acknowledged, the log compaction does not occur and results in out of disk space.
- Filters are applied as an AND condition. This means all conditions must be met for a record to be accepted by the exporter.

Listen to expired messages with a custom filter

You can also create a custom filter to listen to expired messages. This can be useful if you want to take specific actions on messages that have expired, such as logging them or re-publishing them.

For example, if you want to allow exporting only message events with EXPIRED intent, follow the steps below:

1. Implement the RecordFilter interface:

```
public class MessageExpiredExporterFilter implements RecordFilter {

@Override
public boolean acceptType(RecordType recordType) {
   return recordType == RecordType.EVENT;
}

@Override
public boolean acceptValue(ValueType valueType) {
   return valueType == ValueType.MESSAGE;
}

@Override
public boolean acceptIntent(Intent intent) {
   if (intent instanceof MessageIntent messageIntent) {
   return messageIntent == MessageIntent.EXPIRED;
   }

   return true;
}
}
```

note

This filter will only accept records of type EVENT, value type MESSAGE, and intent EXPIRED. To accept more record types, value types, and intents, modify the acceptType, acceptValue, and acceptIntent methods accordingly.

2. Set the MessageExpiredExporterFilter filter in the Exporter#configure method of your custom exporter:

```
public class MessageExpiredExporter implements Exporter {
```

```java
// ...
private Controller controller;

@Override
public void open(final Controller controller) {
  this.controller = controller;
  // ...
}

@Override
public void configure(final Context context) {
  // ...
  context.setFilter(new MessageExpiredExporterFilter());
}

@Override
public void export(final Record<?> record) {
  // ...
  // after handling the record, acknowledge the position
  controller.updateLastExportedRecordPosition(record.getPosition());
}

// ...
}
```

info

Messages with zero TTL will also be exported with this filter. If a message with zero TTL is republished after expiration, it will immediately expire again, causing the exporter to receive it again.

This creates an infinite loop of republishing and expiring the same message, potentially leading to the engine blocked from processing other records. To avoid this, check the message TTL before republishing it.

3. (Optional) By default, the exporter will not receive the full message body, only the message key with the empty message body is exported. To receive the full message body with the expired message, enable it via YAML configuration or environment variable.

    - **Environment variables**
    - **YAML configuration**

```
ZEEBE_BROKER_EXPERIMENTAL_FEATURES_ENABLEMESSAGEBODYONEXPIRED=true
```

caution

Enabling the full message body for expired messages can impact the performance of message expiration.

- When this feature flag is enabled, every deleted message is appended to the Zeebe engine's record stream **including the full message body**.
- Because each expired message now carries its entire payload, the expiration checker's write buffer fills up faster. As a result, the checker requires more time (or more roundtrips) to process the same number of expired messages.
- This can lead to an increasing backlog of messages waiting to be expired.

Schema

In earlier Camunda versions, some upgrades required manual data migrations. These migrations often introduced:

- Operational downtime
- Risk of human error
- Complex backup and rollback procedures

Starting with **Camunda 8.8**, exporters are designed to support upgrades without requiring data migrations.
This approach reduces complexity, minimizes downtime, and enables faster, more reliable releases.

Schema compatibility guidelines (Elasticsearch/OpenSearch)

As new features are added, the indices storing data evolve. To maintain **zero required data migrations** across versions, follow these schema guidelines.

Schema changes to avoid

The following **breaking changes** require data migrations and must be avoided:

- **Field removal**: Deleting existing fields from index mappings
- **Data type changes**: Changing the data type of existing fields (for example, text → keyword)
- **Required field additions**: Adding mandatory fields without default values
- **Record structure changes**: Modifying the structure of exported records in incompatible ways

Safe schema evolution

The following changes are considered **backwards compatible** and do **not** require data migrations:

- **Additive changes**: Adding optional fields with default values
- **New indices**: Creating new indices for new features
- **Index settings**: Updating index settings in ways that do not affect existing data

Integration testing

Schema compatibility is verified through the SchemaUpdateIT integration test.

This test runs on both **Elasticsearch** and **OpenSearch**. Any incompatible schema change causes the test to fail, preventing breaking changes from being introduced.