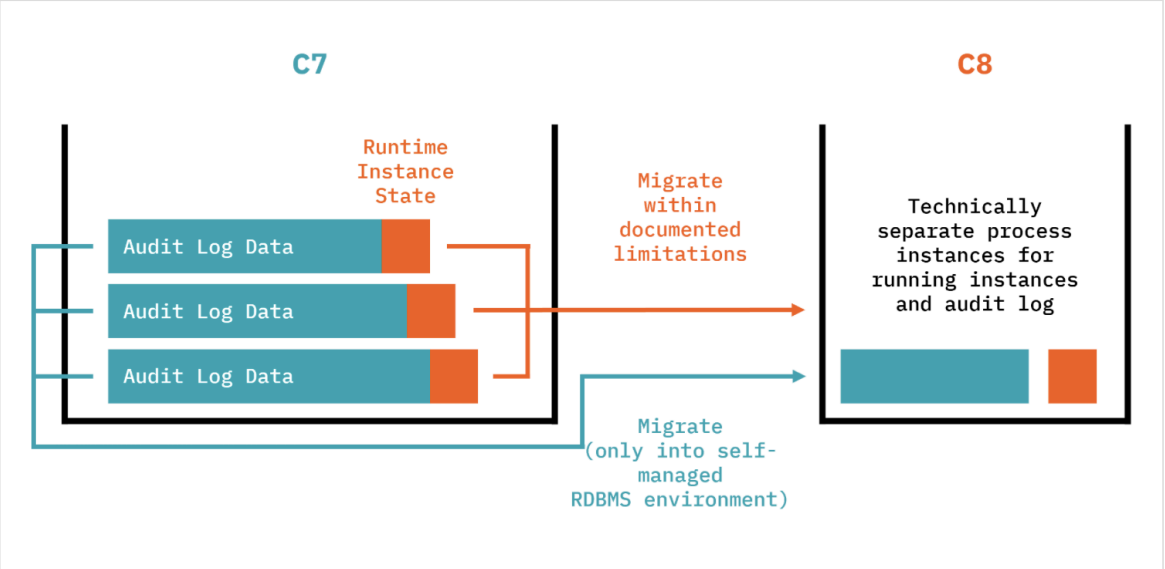
Data Migrator

Use the Data Migrator to copy runtime and audit data from Camunda 7 to Camunda 8.
PRODUCTION READINESS STATUS

- History migration: Experimental and not intended for production usage.



Modes of operation

The Data Migrator offers two modes of operation:

- Runtime migration: Migrate running process instances and continue execution in C8. Production-ready with Camunda 8.8.
- History migration: Copy audit (history) data to Camunda 8 (experimental). Not intended for production, currently an experimental feature.

Migration details are summarized as follows:

What is migrated	What is NOT migrated
• Running process instances (state-preserving). • Process variables and their values. • Execution history (experimental mode available).	• BPMN process models (use the Migration Analyzer & Diagram Converter). • Custom code or integrations (use Code Conversion Utilities). • Users, groups, tenants, and authorizations. • Task assignments and states (due date, priority, etc.).

Key Features

- State-preserving migration: Maintains exact execution state of running process instances
- Variable data migration: Converts and migrates process variables with proper type handling
- Validation and verification: Pre-migration validation to ensure successful migration
- Skip and retry capabilities: Handle problematic instances gracefully with retry options
- Detailed logging and reporting: Comprehensive logging for monitoring migration progress
- Database flexibility: Support for multiple database vendors (H2, PostgreSQL, Oracle)

Typical choreography of runtime and history migration

As described in the roll-out phase of the migration journey, you will typically use the following sequence of tasks when applying both data migrations (while keeping downtimes to a minimum):

1. Stop the Camunda 7 solution (normally shut down your application).
2. Start the Data Migrator in "running instance migration mode".
3. Wait until running instance migration is completed.
4. Start the new Camunda 8 solution immediately so migrated process instances can continue right away.
5. Start the Data Migrator in "history migration mode".
6. The migrator runs until all history data is migrated while Camunda 8 process execution continues in parallel.

With this approach, the duration of history migration doesn't block big bang migrations.

Customization

You might need to customize the data migration, especially if you used complex data formats in Camunda 7 (for example, Java objects) that need to be converted to something Camunda 8 can handle (for example, JSON).

As part of this step, you might also need to extract big payloads and binaries (like documents) into an external data store and reference it from the process (using, for example, upcoming document handling possibilities).

Repository

You can track progress and releases in the repository:
https://github.com/camunda/camunda-7-to-8-data-migrator

Cockpit Plugin

The Cockpit plugin provides a web-based interface to view information about skipped and migrated data during the migration process (experimental).

Version Compatibility

Each version of the Data Migrator is bound to a specific version of Camunda 7 and 8. Only these default combinations are recommended (and supported) by Camunda.

Data Migrator Version	Camunda 7 Version	Camunda 8 Version
`0.1.x`	`7.24.x`	`8.8.x`

Install and quick start

Install the Data Migrator and run your first data migration.

Prerequisites

- Java 21+
- Running Camunda 8 (SaaS or self-managed)
- Access to Camunda 7 database
- Models migrated and deployed to C8

Installation

1. Download the latest release from the GitHub releases page:
    - https://github.com/camunda/camunda-7-to-8-data-migrator/releases
2. Extract the archive to your preferred directory.
3. Navigate to the extracted directory.

Quick start

1. Make sure Camunda 8 is up and running, and all process models to migrate are deployed.
   To be used with the Runtime Data Migrator, every process model requires:

    - A blank start event (you must add one if the process model doesn't have one already).
    - An execution listener at the end of your blank start event with the job type `migrator`. You have to add this manually, or let it be added by the Migration Analyzer & Diagram Converter.
2. <bpmn:startEvent id="StartEvent_1">
3. <bpmn:extensionElements>
      <zeebe:executionListeners>
       <zeebe:executionListener eventType="end" type="migrator" />
      </zeebe:executionListeners>
     </bpmn:extensionElements>
    </bpmn:startEvent>

    - Hint: For automatic resource deployment, you can also drop your BPMN files into the `configuration/resources` folder.
4. Drop your JDBC driver into the `configuration/userlib` folder (for example, `postgresql-42.2.5.jar` for PostgreSQL).
5. Prepare your configuration file (`configuration/application.yml`):
6. camunda.client:
7. mode: self-managed
     grpc-address: http://localhost:26500
     rest-address: http://localhost:8088

    camunda.migrator.c7.data-source:
     jdbc-url: jdbc:postgresql://localhost:5432/camunda7
     username: your-username
     password: your-password

8. Run the Data Migrator:
9. *# On Linux/macOS*
10. ./start.sh --help

    # On Windows
    start.bat --help

11. Monitor the migration progress in the console output and log files.

Runtime migration

Migrate currently running process instances.

About runtime migration

Running refers to process instances in Camunda 7 are not yet ended and are currently waiting in a wait-state. This state is persisted in the database, and a corresponding data entry must be created in Camunda 8, so the process instance can continue from that state in the new solution.

Requirements and limitations

The following requirements and limitations apply:

- The Runtime Data Migrator needs to access the Camunda 7 database.
- The Runtime Data Migrator needs to access Orchestration Cluster APIs (which means you can also use it when running SaaS).
- You must be familiar with the Data Migrator limitations.

If you need to adjust your process models before migration, you can use process version migration in the Camunda 7 environment to migrate process instances to versions that are migratable to Camunda 8. An interesting strategy can be to define dedicated migration states you want your process instances to pile up in. Another common strategy is to use process instance modification in the Camunda 7 environment to move out of states that are not migratable (for example, process instances within a multiple instance task).

To use the Runtime Data Migrator, every Camunda 8 process model must have:

- A process-level None Start Event, and
- An execution listener on that None Start Event with the type `migrator` (or a validated job type, see below).

Example:

```
<bpmn:startEvent id="StartEvent_1">
 <bpmn:extensionElements>
  <zeebe:executionListeners>
   <zeebe:executionListener eventType="end" type="migrator" />
  </zeebe:executionListeners>
 </bpmn:extensionElements>
</bpmn:startEvent>
```

Choreography

The runtime migration typically follows these phases:

1. Preparation

- Stop Camunda 7 process execution to avoid starting new instances during migration.
- Migrate BPMN models using the Migration Analyzer & Diagram Converter.
- Add required `migrator` execution listeners to None Start Events in Camunda 8 models.
- Adjust Camunda 8 models to comply with migration limitations.
- Test migrated models in a Camunda 8 environment.
- Back up your Camunda 7 database before migration.

2. Migration

- Deploy Camunda 8 process models and resources to the target environment.
- Configure the migrator with proper database connections and settings.
- Start the migrator and monitor progress through logs.
- Verify results in Camunda 8 Operate.
- Handle skipped instances by reviewing and addressing validation failures.
- After successful migration, clean up models if needed:
    - Remove `migrator` execution listeners from Camunda 8 models.
    - Revert temporary model changes.
    - Migrate instances to the latest version of Camunda 8 models if appropriate.

3. Validation

- Check migrated instances in Camunda 8 Operate.
- Verify variable data and migrated state.
- Test process continuation by completing some migrated instances.
- Monitor system performance and resource usage.
- Validate business logic continues to work as expected.

Validation and skip reasons

The migrator validates each process instance before migration and will skip instances that fail validation for the following reasons:

Skip reason	Condition (why it is skipped)
Missing Camunda 8 process definition	No corresponding Camunda 8 process definition is found for the Camunda 7 process ID.
Multi-instance activities	The process instance has active multi-instance activities.
Missing flow node elements	The Camunda 7 instance is at a flow node that does not exist in the deployed Camunda 8 model.
Missing None Start Event	The Camunda 8 process definition does not have a process-level None Start Event.
Missing `migrator`execution listener	The Camunda 8 process definition does not have an execution listener of type `migrator` on the None Start Event.
Multi-tenancy	The tenant ids are not configured in the Data Migrator.

When a process instance is skipped:

- The skipped process instance is logged.

- The instance is marked as skipped in the migration database.
- You can list skipped instances.
- You can retry migration of skipped instances after fixing the underlying issues.

Common resolution steps

1. Deploy the missing Camunda 8 process definition
2. Wait for multi-instance activities to complete
3. Ensure all active flow nodes in the Camunda 7 process have corresponding elements in the Camunda 8 process
4. Modify process instance to a supported state

Usage examples

```
# Run runtime migration
./start.sh --runtime

# List all skipped process instances
./start.sh --runtime --list-skipped

# Retry skipped process instances
./start.sh --runtime --retry-skipped
```

Job type configuration

During migration, the Data Migrator starts new Camunda 8 process instances and sets a special `legacyId` variable to link them to their original Camunda 7 process instances. The migrator uses execution listeners on start events for its internal migration logic.

However, if users manually start new Camunda 8 process instances on models that still have these migration execution listeners, those instances won't have the `legacyId` variable. This creates a problem:

- The migrator would try to migrate process instances that don't need migration
- This could cause errors or unexpected behavior

The `validation-job-type` feature solves this by allowing you to use a FEEL expression to create different job types for externally started process instances versus process instances started by the migrator. This ensures only instances that truly need migration are processed by the migrator.

The migrator supports two job type configurations with fallback behavior:

- `job-type`: Used for actual job activation.
  - This is the primary job type used when activating jobs in Camunda 8.
  - It is required for the migrator to function correctly.
- `validation-job-type`: Used for validation purposes (optional).
  - You can define a FEEL expression that provides different job types based on the process instance context.
  - The BPMN process will be validated to contain a start event execution listener with the respectively defined FEEL expression.

Default Behavior: When `validation-job-type` is not defined, `job-type` is used for both validation and activation.

Basic Configuration:
```
camunda.migrator:
 job-type: migrator # Used for both validation and activation
```

Separate Validation and Activation:
```
camunda.migrator:
 job-type: migrator # Used for activation
 validation-job-type: '=if legacyId != null then "migrator" else "noop"' # Used for validation with FEEL expression
```

Important notes:

- If `validation-job-type` is not defined, `job-type` is used for both purposes.
- The `job-type` is always used for job activation.
- The `validation-job-type` can be a FEEL expression (starts with =).
- Set `validation-job-type` to `DISABLED` to disable job type validation entirely.
- Use FEEL expressions only for validation, not for job activation since on job activation, the FEEL expression is already evaluated to a static value.
- You can use variables like the special variable `legacyId` in FEEL expression.

FEEL Expression Requirements: When using FEEL expressions in the `validation-job-type` property, you must also specify the same expression in the execution listener of your BPMN process start events. The migrator configuration alone is not sufficient.

Example BPMN configuration for FEEL expression validation:
```
<bpmn:startEvent id="StartEvent_1">
```

```
 <bpmn:extensionElements>
  <zeebe:executionListeners>
   <zeebe:executionListener eventType="end" type="=if legacyId != null then
&amp;quot;migrator&amp;quot; else &amp;quot;noop&amp;quot;" />
  </zeebe:executionListeners>
 </bpmn:extensionElements>
</bpmn:startEvent>
```

Externally Started Process Instances

Migrator jobs for externally started process instances (process instances not started by the Data Migrator) are activated but not further processed by the Data Migrator since these process instances do not contain the `legacyId` variable that the migrator uses to identify instances that need migration. After the default lock timeout the jobs will be available again for activation.

When using FEEL expressions such as `=if legacyId != null then "migrator" else "noop"` in the execution listener, externally started process instances will generate jobs with the type noop instead of `migrator`. To handle these jobs properly, you must implement a noop job worker that simply activates and completes these jobs without performing any migration logic.

Example noop job worker implementation:

```
@JobWorker(type = "noop")
public void handleNoopJobs(ActivatedJob job) {
  // Simply complete the job without any processing
  // This allows externally started process instances to continue normally
}
```

This approach ensures that:

- Process instances started by the Data Migrator are handled by the `migrator` job worker.
- Externally started process instances continue their normal execution flow through the noop job worker.
- Both types of instances can coexist in the same Camunda 8 environment.

Tenants

- Camunda 7 process instances assigned to no tenant (`tenantId=null`) are migrated to Camunda 8 with `<default>` tenant.

- The default behavior is to migrate only process instances without assigned tenant.
- When migrating process instances, the migrator can be configured to handle specific tenants throughout the migration process. Defining tenant IDs ensures that only process instances associated with those tenants are migrated.
    - Make sure to create the tenants in Camunda 8 before starting the migration.
    - Add `Authentication configuration for the client` and assign the user to the all of the tenants

How Multi-Tenancy Works

1. Validation: Pre-migration validation ensures target tenant deployments exist in Camunda 8
2. Tenant Preservation: Process instances maintain their original tenant association during migration
3. Job Activation: The migrator fetches jobs only for the configured tenants and the default tenant when activating jobs

Example

Use the `camunda.migrator.tenant=ids` property to specify which tenants should be included in the migration process. This property accepts a comma-separated list of tenant identifiers.

```
camunda:
 migrator:
  tenant-ids: tenant-1, tenant-2, tenant-3
```

With this configuration, only process instances associated with `tenant-1`, `tenant-2`, `tenant-3`, and the default tenant will be created and migrated. Instances associated with other tenants will be skipped.

Dropping the migration mapping schema

The migrator uses the `{prefix}MIGRATION_MAPPING` table to keep track of instances.

If you wish to drop this table after migration is completed, you can use the `--drop-schema` flag when starting the migrator. This will drop the migration mapping schema on shutdown if the migration was successful (no entities were skipped).

 # Migrate and drop the migration mapping schema on shutdown if migration was succcesful
./start.sh --runtime --drop-schema

If you wish to drop the table regardless of the migration status, you can use the `--force` flag in combination with `--drop-schema`. This will perform the drop in all cases.
 # Migrate and force drop the migration mapping schema on shutdown
./start.sh --runtime --drop-schema --force
WARNING

Using `--force` can lead to data loss. Use with caution.

History migration (experimental)

Use the History Data Migrator to copy process instance audit data to Camunda 8.
INFO

The history migration mode of the Data Migrator will not be released before Camunda 8.9 (April 2026). You can check the current state and track progress in the GitHub repository.

About history migration

Process instances leave traces, referred to as History in Camunda 7. These are audit logs of when a process instance was started, what path it took, and so on.

It is important to note that audit data can exist for ended processes from the past, but is also available for currently still running process instances, as those process instances also left traces up to the current wait state.

The History Data Migrator can copy this audit data to Camunda 8.

Audit data migration might need to look at a huge amount of data, which can take time to migrate. In early measurements, migrating 10,000 process instances took around 10 minutes, but the number varies greatly based on the amount of data attached to a process instance (for example, user task instances, variable instances, and so on).

You can run audit data migration alongside normal operations (for example, after the successful big bang migration of runtime process instances) so that it doesn't require downtime and as such, the performance might not be as critical as for runtime instance migration.

Requirements and limitations

The following requirements and limitations apply:

- The History Data Migrator needs to access the Camunda 7 database.
- The History Data Migrator can only migrate to Camunda 8 if a relational database (RDBMS) is used, a feature planned for Camunda 8.9.
- The History Data Migrator needs to access the Camunda 8 database (which means you can only run this tool in a self-managed environment).
- If runtime and history data are migrated at the same time, you will end up with two instances in Camunda 8: a canceled historic process instance and an active new one in the runtime. They are linked by process variables.

Usage examples

```
# Run history migration (experimental)
./start.sh --history

# List all skipped history entities
./start.sh --history --list-skipped

# List skipped entities for specific types
./start.sh --history --list-skipped HISTORY_PROCESS_INSTANCE
HISTORY_USER_TASK

# Retry skipped history entities
./start.sh --history --retry-skipped
```

Entity types

Entity type	Description

`HISTORY_PROCESS_DEFINITION`	Process definitions
`HISTORY_PROCESS_INSTANCE`	Process instances
`HISTORY_INCIDENT`	Process incidents
`HISTORY_VARIABLE`	Process variables
`HISTORY_USER_TASK`	User tasks
`HISTORY_FLOW_NODE`	Flow node instances
`HISTORY_DECISION_INSTANCE`	Decision instances
`HISTORY_DECISION_DEFINITION`	Decision definitions

History cleanup

The history cleanup date is migrated if the Camunda 7 instance has a removal time.

Tenants

- Camunda 7's `null` tenant is migrated to Camunda 8's `<default>` tenant.
- All other `tenantIds` will be migrated as-is.
- For details, see multi-tenancy in Camunda 8.

Variables

The Data Migrator automatically handles the transformation of Camunda 7 variables to Camunda 8 compatible formats during migration.
INFO

The handling and intercepting of variables described on this page is currently only supported for the Runtime Data Migrator.

About variables

This section documents which variable types are supported and how they are transformed.

For complete details on Camunda 7 variable types, see the official Camunda 7 documentation.

Supported Types

The following table shows how different Camunda 7 variable types are handled during migration:

Camunda 7 Type	Example Value	Migration Behavior	Camunda 8 Result	Interceptor Type

String	`"hello world"`	Direct migration	String value	`StringValue`, `PrimitiveValue`
Boolean	`true`, `false`	Direct migration	Boolean value	`BooleanValue`, `PrimitiveValue`
Integer	`42`, `1234`	Direct migration	Number value	`IntegerValue`, `PrimitiveValue`
Long	`123456789L`	Direct migration	Number value	`LongValue`, `PrimitiveValue`
Double	`3.14159`	Direct migration	Number value	`DoubleValue`, `PrimitiveValue`
Short	`(short) 1`	Direct migration	Number value	`ShortValue`, `PrimitiveValue`

Null	`null`	Direct migration	Null value	`NullValueImpl`
Date	`new Date()`	Converted to ISO format	String (ISO 8601)	`DateValue`, `PrimitiveValue`
Java Object serialized as JSON	Serialized JSON	Converted to Map	JSON object	`ObjectValue`, `SerializableValue`
Spin JSON	`SpinJsonNode`	Converted to Map	JSON object	`SpinValue`, `SerializableValue`
Spin XML	`SpinXmlElement`	Converted to String	String (raw XML)	`SpinValue`, `SerializableValue`

Java Object serialized as XML	XML serialized object	Converted to String	String (raw XML)	`ObjectValue`, `SerializableValue`

Unsupported Types

When a process instance contains unsupported variable types, the migrator will:

- Skip the entire process instance
- Log a detailed error message indicating the variable type that caused the skip
- Mark the instance as skipped for potential retry after manual intervention

The following Camunda 7 variable types are not supported and will cause the process instance migration to be skipped:

Camunda 7 Type	Example	Interceptor Type
Byte Array	`"hello".getBytes()`	`BytesValue`, `PrimitiveValue`
File	`FileValue` objects	`FileValue`

Java Serialized Objects	Java objects serialized as `application/x-java-serialized-object` like `List`, `Set`, `Map`, `float`, `byte`, `char` or custom types	`ObjectValue`, `SerializableValue`

Transformation

Variable transformations are handled by built-in transformers that run in a specific execution order. Validators run first (Order: 1-3) to reject unsupported types, followed by transformers (Order: 10-20) that convert supported types.

Date

- Input: Java `Date` objects from Camunda 7
- Output: ISO 8601 formatted strings (`yyyy-MM-dd'T'HH:mm:ss.SSSZ`)
  - Example: `2024-07-25T14:30:45.123+0200`
- Timezone: Uses the JVM's default timezone setting

JSON

JSON variables are handled differently depending on their origin:

Spin JSON Variables and JSON Object Variables (serialized with `application/json`):

- Deserialized into Map structures for Camunda 8
- Maintains nested object structure
- Example: `{"name": "John", "age": 30}` becomes a Map object

Invalid JSON: If JSON cannot be parsed, the process instance is skipped.

XML

Spin XML Variables and XML Object Variables (serialized with `application/xml`):

- Raw XML string content is preserved
- No parsing or transformation applied

Name Compatibility

The migrator handles variable names that are invalid in FEEL expressions:

- Names starting with numbers (e.g., `1stVariable`)
- Names with spaces (e.g., `my variable`)
- Names with special characters (e.g., `var/name`, `var-name`)
- Reserved keywords (e.g., `null`)

These variables are migrated as-is, but may require special handling in FEEL expressions using bracket notation.

Disabling Built-in Interceptors

You can disable any built-in transformer or validator using the `enabled` configuration property. Use the class names from the tables above:

```
camunda:
 migrator:
  # Variable interceptor plugins configuration
  interceptors:
   # Disable date transformation
   - class-name: io.camunda.migrator.impl.interceptor.DateVariableTransformer
     enabled: false
```

You can find a complete list of built-in interceptors in the property reference.

Custom Transformation

The `VariableInterceptor` interface allows you to define custom logic that executes whenever a variable is accessed or modified during migration. This is useful for auditing, transforming, or validating variable values.

Custom interceptors are enabled by default and can be restricted to specific variable types.

How to Implement a `VariableInterceptor`

1. Create a new Maven project with the provided `pom.xml` structure
2. Add a dependency on `camunda-7-to-8-data-migrator-core` (scope: `provided`)
3. Implement the `VariableInterceptor` interface

4. Add setter methods for any configurable properties
5. Package as JAR and deploy to the `configuration/userlib` folder
6. Configure in `configuration/application.yml`

Creating a Custom Variable Interceptor

Here's an example of a custom variable interceptor which is only called for string variables:

```
public class MyVariableInterceptor implements VariableInterceptor {

  /**
   * Restrict this interceptor to only handle string variables.
   */
  @Override
  public Set<Class<?>> getTypes() {
     return Set.of(StringValue.class);
  }

  @Override
  public void execute(VariableInvocation invocation) {
    // ...
  }

}
```

Type Restrictions

Variable interceptors can be restricted to specific variable types using the `getTypes()` method. You can find a complete list of available variable types for restriction as subinterfaces of the `TypedValue`interface in the JavaDoc.

```
@Override
public Set<Class<?>> getTypes() {
  // Handle only specific types
  return Set.of(
     PrimitiveValue.class,    // String, Integer, Boolean, etc.
     DateValue.class,         // Date variables
     ObjectValue.class        // JSON, XML, Java serialized objects
  );
}
// Or handle all types (default behavior)
@Override
public Set<Class<?>> getTypes() {
```

```
    return Set.of(); // Empty set = handle all types
}
```

Configuring Custom Interceptors

Configure your custom interceptors in `application.yml`:

```
# Variable interceptor plugins configuration
# These plugins can be packaged in JARs and dropped in the userlib folder
camunda:
 migrator:
  interceptors:
   - class-name: com.example.migrator.AuditVariableInterceptor
     enabled: true
     properties:
       prefix: "CUSTOM_PREFIX_"
       enableLogging: true
```

Deployment

1. Package your custom interceptor as a JAR file
2. Place the JAR in the `configuration/userlib/` folder
3. Configure the interceptor in `configuration/application.yml`
4. Restart the Data Migrator

The `enabled` property is supported for all interceptors (both built-in and custom) and defaults to `true`.

See example interceptor.

Execution Order

- Custom interceptors configured in the `application.yml` are executed in their order of appearance from top to bottom
    - Built-in interceptors run first, followed by custom interceptors
- In a Spring Boot environment, you can register interceptors as beans and change their execution order with the `@Order` annotation (lower values run first)

Error Handling

When variable transformation fails:

- The entire process instance is skipped

- Detailed error messages are logged with the specific variable name and error cause
- The instance is marked for potential retry after fixing the underlying issue
- You can use `--list-skipped` and `--retry-skipped` commands to manage failed migrations

Cockpit plugin

EXPERIMENTAL FEATURE

- The Cockpit plugin is an experimental feature and we don't recommend using it in production environments.
- You can read more about limitations in the limitations page.

The Cockpit plugin provides a web-based interface for viewing information about skipped and migrated runtime and history data. It integrates with Camunda 7 Cockpit to give you visibility into which runtime process instances or history data (like variables, flow nodes, etc.) were successfully migrated or skipped during migration and the reasons why.

For more information on Camunda 7 plugins, see the Camunda 7 documentation.

Prerequisites

- Database Configuration: The plugin can only access databases that Camunda 7 is connected to, so the migration schema needs to be created in the Camunda 7 database (default behavior), or both Camunda 7 and Camunda 8 databases must point to the same database instance. For an example, see the Data Migrator setup example.
- Camunda 7 Webapps are deployed, running, and accessible.
- Migration schema is available in Camunda 7 database.
- The Cockpit plugin requires running the migrator with `save-skip-reason` enabled.
    - The plugin doesn't show skip reasons without this setting because they are not stored.
- To use the Cockpit plugin, run the migrator with the following setting:
- camunda.migrator:
- save-skip-reason: true

HEADS-UP

- Saving the skip reason could result in a large amount of data being stored additionally in your database, depending on the order of magnitude of the data to be migrated or potentially skipped.
- We recommend testing your migration in a QA environment before running the Data Migrator against your production database.

Installation

1. Download the latest release from the releases page.
2. Deploy the plugin to your Camunda 7 installation by copying the generated JAR file into the Camunda 7 plugins directory. For example the paths are:
    - For Tomcat: `./camunda-bpm-ee-tomcat-<camunda-7-version>-ee/server/apache-tomcat-<tomcat-version>/webapps/camunda/WEB-INF/lib/`.
    - For Run: `./camunda-bpm-run-ee-<camunda-7-version>-ee/configuration/userlib/`.
3. Inspect skipped and migrated data in Cockpit once the plugin has been deployed.

Using the Cockpit plugin

After installation and configuration, the Cockpit plugin provides:

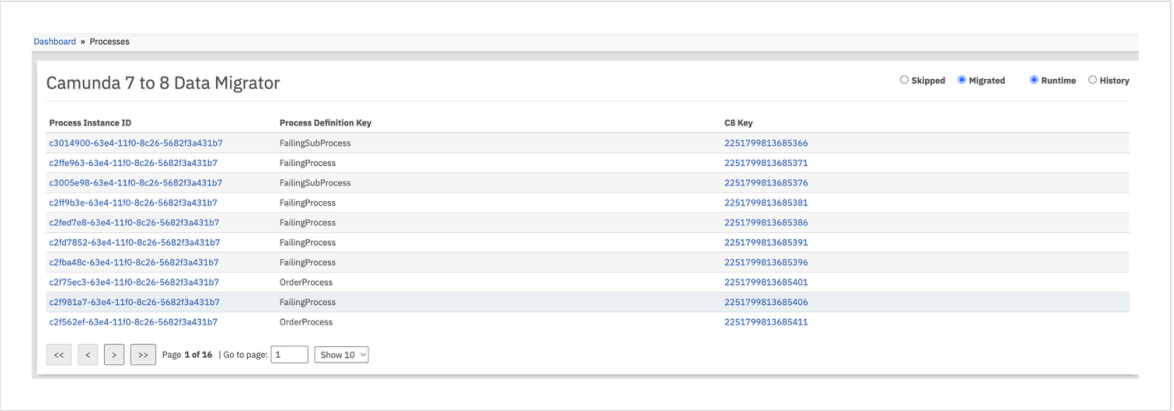
- Skipped entity overview: View all entities that were skipped during migration.
- Detailed skip reasons: Understand why specific entities were not migrated.
- Migration status tracking: See data that has been migrated successfully.

Screenshots

The following screenshots demonstrate the Cockpit plugin interface and functionality:

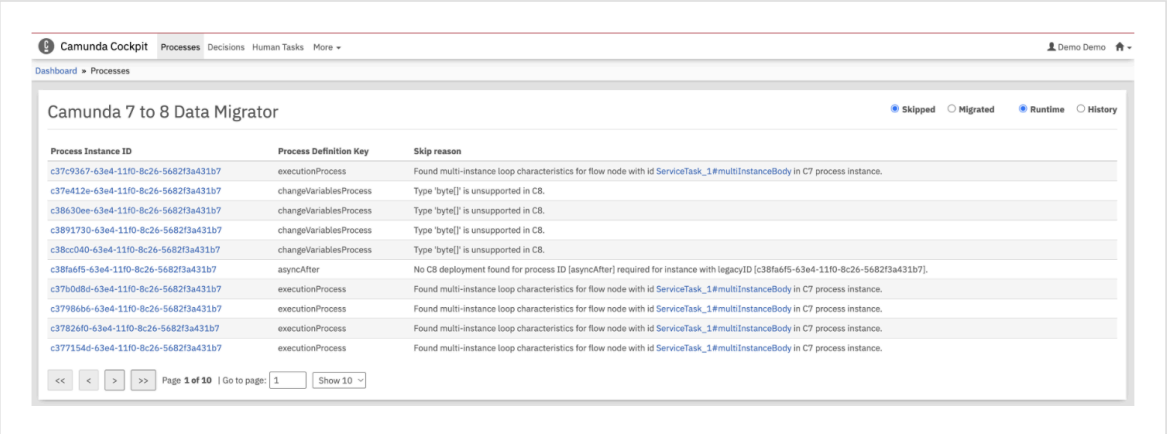
Migrated process instances view

Shows a table of successfully migrated process instances from Camunda 7 to Camunda 8, including the process instance ID, process definition key, and the corresponding Camunda 8 key.



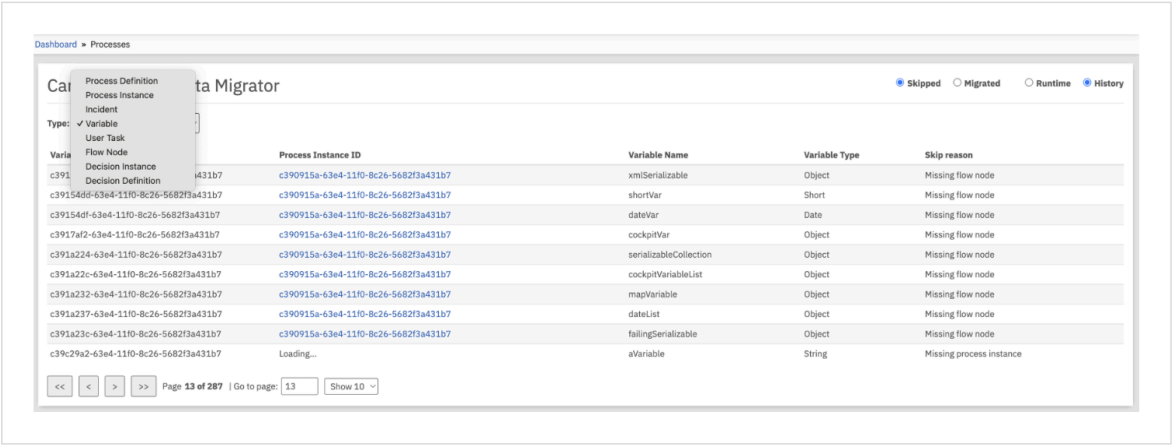
Skipped process instances overview

Displays process instances that were skipped during migration, allowing users to identify which instances failed and need further attention.



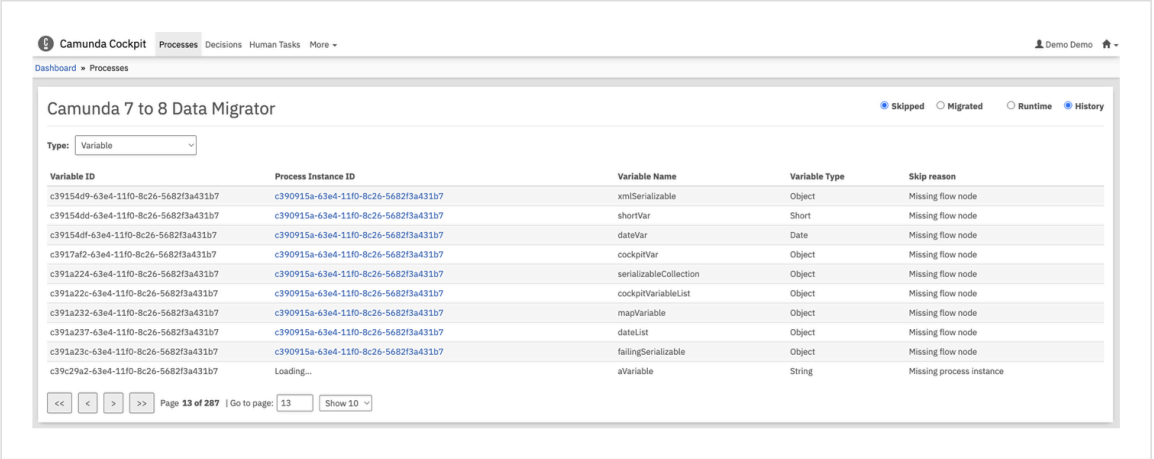
Entity type selection

Allows filtering historic entities by type (process instances, variables, tasks, etc.) to simplify analysis of migration issues.



Variable-specific skip analysis

When viewing historic variable data, the type and value of primitives are retrieved to provide additional context.



Limitations

An overview of the current limitations of the Camunda 7 to Camunda 8 Data Migrator, covering general limitations as well as specific limitations related to variables and BPMN elements.

Runtime

The runtime migration has the following limitations.

General limitations

- To migrate running process instances, the historic process instance must exist.
    - You cannot migrate running instances when you have configured history level to `NONE` or a custom history level that doesn't create historic process instances.
    - The minimum supported history level is `ACTIVITY`.
- You must add an execution listener of type `migrator` to all your start events.
- Migration of users, groups, or tenants as well as authorizations is currently not supported.
    - You must ensure that the users, groups, and authorizations are already migrated to Camunda 8 before migrating process instances.
    - See https://github.com/camunda/camunda-bpm-platform/issues/5175
- Data changed via user operations
    - Data set via user operations like setting a due date to a user task cannot be migrated currently.
    - See https://github.com/camunda/camunda-bpm-platform/issues/5182

Variables

- Proper handling and intercepting of variables is currently only supported for the Runtime Data Migrator.
- Unsupported Camunda 7 types.
- Camunda 8 variable name restrictions.
    - Variables that do not follow the restrictions will cause issues in FEEL expression evaluation.
- Variables set into the scope of embedded sub-processes are not supported yet and will be ignored.
    - See https://github.com/camunda/camunda-bpm-platform/issues/5235

INFO

To learn more about variable migration, see variables.

BPMN elements

Some BPMN elements and configurations supported in Camunda 7 are not supported in Camunda 8 or have specific limitations during migration. Below is an overview of these limitations and recommendations to address them.

Elements supported in Camunda 7 but not supported in Camunda 8

See the BPMN documentation for more details on element support in Camunda 8, and adjust your models accordingly before migration.

Start events

- It is required that a process instance contains a single process level None Start Event to run the data migrator.
- If a process definition only has event-based start events (for example, Message, Timer), it is required to add a temporary None Start Event. This change must be reverted after the data migration is completed.
- Example adding a None Start Event:

```
 <bpmn:process id="Process_1fcbsv3" isExecutable="true">
  <bpmn:startEvent id="StartEvent_1">
    <bpmn:outgoing>Flow_FromEventStartEvent</bpmn:outgoing>
    <bpmn:messageEventDefinition id="MessageEventDefinition_1yknqqn" />
  </bpmn:startEvent>
  <bpmn:sequenceFlow id="Flow_FromEventStartEvent" sourceRef="StartEvent_1"
targetRef="ActivityId" />
+   <bpmn:startEvent id="NoneStartEvent">
+     <bpmn:outgoing>Flow_FromNoneStartEvent</bpmn:outgoing>
+   </bpmn:startEvent>
+   <bpmn:sequenceFlow id="Flow_FromNoneStartEvent"
sourceRef="NoneStartEvent" targetRef="ActivityId" />
  <bpmn:task id="ActivityId">
    <bpmn:incoming>Flow_FromEventStartEvent</bpmn:incoming>
    <bpmn:incoming>Flow_FromNoneStartEvent</bpmn:incoming>
    <bpmn:outgoing>Flow_1o2i34a</bpmn:outgoing>
  </bpmn:task>
  <bpmn:endEvent id="EndEvent">
    <bpmn:incoming>Flow_1o2i34a</bpmn:incoming>
  </bpmn:endEvent>
  <bpmn:sequenceFlow id="Flow_1o2i34a" sourceRef="ActivityId"
targetRef="EndEvent" />
 </bpmn:process>
```

Async before/after wait states

Camunda 8 does not support asynchronous continuation before or after any kind of wait state. Service-task-like activities are executed asynchronously by default in Camunda 8 - so for example a service task waiting for asynchronous continuation before will be correctly migrated. But if you need to migrate an instance currently

waiting asynchronously at other elements in a Camunda 7 model, such as a gateway for example, this instance would just continue without waiting in the equivalent Camunda 8 model. You might need to adjust your model's logic accordingly prior to migration.

Message events

- Only message catch and throw events are supported for migration.
- Depending on your implementation, you may need to add a correlation variable to the instance pre-migration.

Message and signal start events

- If your process starts with a message/signal start event, no token exists until the message/signal is received and hence no migration is possible until that moment.
- Once the message/signal is received, the token is created and moved down the execution flow and may be waiting at a migratable element inside the process. However, due to how the migration logic is implemented, at the moment the data migrator only supports processes that start with a normal start event.

Triggered boundary events

- Camunda 7 boundary events do not have a natural wait state.
- If the process instance to be migrated is currently at a triggered boundary event in Camunda 7, there may still be a job associated with that event, either waiting to be executed or currently running. In this state, the token is considered to be at the element where the job is created: typically the first activity of the boundary event's handler flow, or technically the point after the boundary event if asyncAfter is used.
- During migration to Camunda 8, the token will be mapped to the corresponding target element. However, if that element expects input data that is normally produced by the boundary event's job (for example, setting variables), this data may be missing in the migrated instance.
- Recommendation: To ensure a consistent migration, allow boundary event executions to complete before initiating the migration.

Call activity

To migrate a subprocess that is started from a call activity, the migrator must set the `legacyId` variable for the subprocess. This requires propagating the parent variables. This can be achieved by updating the Camunda 8 call activity in one of the following ways:

- Set `propagateAllParentVariables` to `true` (this is the default) in the `zeebe:calledElement` extension element.
- Or, if `propagateAllParentVariables` is set to `false`, provide an explicit input mapping:

```
<zeebe:ioMapping>
 <zeebe:input source="=legacyId" target="legacyId" />
</zeebe:ioMapping>
```

Multi-instance

Processes with active multi-instance elements can currently not be migrated. We recommend to finish the execution of any multi-instance elements prior to migration.

Parallel gateways

Process instances with active joining parallel gateways cannot currently be migrated. The migrator will skip these instances during migration.

- This limitation occurs when some execution paths have completed and reached the joining parallel gateway, but other paths are still waiting at activities before the gateway.
- Recommendation: Ensure no token waits in a joining parallel gateway.
- See https://github.com/camunda/camunda-bpm-platform/issues/5461

Timer events

- Timer start events: prior to migration, you must ensure that your process has at least one none start event. Processes that only have a timer start event cannot be migrated.
- If your model contains timer events (start and other), you must ensure that no timers fire during the migration process.
    - Timers with date: ensure the date lies outside the migration time frame.
    - Timers with durations: ensure the duration is significantly longer than the migration time frame.
    - Timers with cycles): ensure the cycle is significantly longer than the migration time frame and/or use a start time that lies outside the migration time frame.
- Note that during deployment and/or migration, the timers may be restarted. If business logic requires you to avoid resetting timer cycles/duration, you need to apply a workaround:
    - Timers with cycles:

            - Add a start time to your cycle definition that is equal to the moment in time when the currently running Camunda 7 timer is next due.
            - You must still ensure that the start time lies outside the migration time frame.
    - Timers with durations:
        - Non-interrupting timer boundary events:
            - Switch to cycle definition with a start time that is equal to the moment in time when the currently running Camunda 7 timer is next due and add a "repeat once" configuration.
            - This way, for the first post migration run, the timer will trigger at the start time.
            - For all subsequent runs, the defined cycle duration will trigger the timer. The "repeat once" instruction ensures it only fires once, similar to a duration timer.
            - You must still ensure that the start time lies outside the migration time frame.
        - Interrupting boundary and intermediate catching events:
            - Add a variable to your Camunda 7 instance that contains the leftover duration until the next timer is due.
            - In your Camunda 8 model, adjust the timer duration definition to use an expression: if the variable is set, the value of this variable should be used for the duration. If the variable is not set or does not exist, you may configure a default duration.
            - This way, for the first post migration run the variable will exist and the timer will set its duration accordingly.
            - For all subsequent runs, the variable will not exist and the default duration will be used.
            - Again, you must ensure the leftover duration for the first post migration run lies outside the migration time frame.

Event subprocesses

- Event subprocesses with interrupting start events can cause unexpected behavior during migration if triggered at the wrong moment. This includes timer, message, and signal start events.
- What can go wrong:
    - A task that already ran in Camunda 7 might run again in Camunda 8.
    - The process might end up in the wrong state after migration — for example, being one step behind what you see in Camunda 7.
- When could it happen:

    - This can occur when a process instance is already inside an event subprocess in Camunda 7, and the start event of that same subprocess is accidentally triggered again in Camunda 8 during migration.
- How to prevent it:
    - Don't correlate messages or send signals during migration.
    - Temporarily adjust timer start events in event subprocesses to ensure they do not trigger during migration (see the section on timer events for more details).
    - If above suggestions are not feasible in your use case make sure service tasks are idempotent — so repeating them does not cause issues.

History

The history migration has the following limitations.

Process instance

- Process instance migration doesn't populate the `parentElementInstanceKey` and `tree` fields.
- This means that the history of subprocesses and call activities is not linked to their parent process instance.
- As a result, you cannot query for the history of a subprocess or call activity using the parent process instance key.
- See https://github.com/camunda/camunda-bpm-platform/issues/5359

DMN

- The Data Migrator only migrates instances which are linked to process definition business rule tasks.
- The properties `evaluationFailure` and `evaluationFailureMessage` are not populated in migrated decision instances.
- Decision instance `inputs` and `outputs` are not yet migrated.
    - See https://github.com/camunda/camunda-bpm-platform/issues/5364

Cockpit plugin

The Cockpit plugin has the following limitations:

- The migration schema has no authorization mechanism. Anyone with authenticated access to the Camunda 7 Cockpit can see the Cockpit Plugin and read the migration schema.
- If the migration of a process instance or any other entity is skipped for multiple reasons, only one reason is stored and displayed.
    - See https://github.com/camunda/camunda-bpm-platform/issues/5389
- For historic data migration the skip reason is currently only stored for the initial migration attempt. If migration fails again after retry, the skip reason is not updated.
    - See https://github.com/camunda/camunda-bpm-platform/issues/5390
- There are currently some UI inconsistencies. See:
    - https://github.com/camunda/camunda-bpm-platform/issues/5422
    - https://github.com/camunda/camunda-bpm-platform/issues/5423
    - https://github.com/camunda/camunda-bpm-platform/issues/5424
- The Cockpit plugin doesn't have extensive test coverage yet so we cannot guarantee a high level of stability and therefore don't claim it to be production-ready.
    - See https://github.com/camunda/camunda-bpm-platform/issues/5404

Database

Database configuration is required for both Camunda 7 and Camunda 8 (RDBMS history) data sources. The Data Migrator uses JDBC to connect to these databases.

Setup

1. Include the appropriate JDBC driver in the classpath by dropping the JAR into `configuration/userlib`.
2. Configure connection details in `configuration/application.yml`.
3. Set table prefixes if your installation uses them.
4. Verify connectivity before starting migration.
5. Ensure sufficient disk space for migration data.

INFO

The database vendor is automatically detected but can be overridden using the `database-vendor`property.

Transaction isolation level

The required isolation level to run the Data Migrator with is `READ COMMITTED`. Other transaction isolation levels are not supported and might lead to unexpected behavior.

Compatibility

The migrator supports the following SQL databases:

Database	Version	JDBC Driver	Notes
H2	2.3.232	`org.h2.Driver`	Default, good for testing
PostgreSQL	17	`org.postgresql.Driver`	Recommended for production
Oracle	23ai	`oracle.jdbc.OracleDriver`	Recommended for production

The migrator supports migration only within the same database vendor:

Migration Path	Status

PostgreSQL → PostgreSQL	✅ Supported
Oracle → Oracle	✅ Supported
H2 → H2	✅ Supported
PostgreSQL → Oracle	❌ Not supported
Oracle → PostgreSQL	❌ Not supported

Configuration examples

Configuration examples for the Data Migrator's `configuration/application.yml`.

Camunda 8 Client

```
camunda.client:
 mode: self-managed # Operation mode: 'self-managed' or 'cloud'
 grpc-address: http://localhost:26500 # The gRPC API endpoint
 rest-address: http://localhost:8088 # The REST API endpoint
```

Data Migrator

```
camunda.migrator:
 page-size: 500 # Number of records to process in each page
 job-type: migrator # Job type for actual job activation (used for validation and activation unless validation-job-type is defined)
 validation-job-type: '=if legacyId != null then "migrator" else "noop"' # Job type for validation (optional - falls back to job-type if not defined)
 auto-ddl: true # Automatically create/update database schema
 table-prefix: MY_PREFIX_ # Optional table prefix for migrator schema
```

```
 data-source: C7 # Choose if the migrator schema is created on the data source of 'C7' or 'C8'
 interceptors:
   - class-name: com.example.MyCustomInterceptor # Custom interceptor class
   - class-name: com.example.AnotherInterceptor # Another custom interceptor class
```

Datasource

Configure the Camunda 7 and Camunda 8 datasources. You can use the same or different databases for Camunda 7 and Camunda 8.

Camunda 7 (runtime and history)

```
camunda.migrator.c7.data-source:
 table-prefix: MY_PREFIX_ # Optional prefix for Camunda 7 database tables
 auto-ddl: true # Automatically create/update Camunda 7 database schema
 jdbc-url: jdbc:h2:./h2/data-migrator-source.db
 username: sa # Database username
 password: sa # Database password
 driver-class-name: org.h2.Driver
```

You can apply any HikariCP property (for example, pool size) under `camunda.migrator.c7.data-source`.

Camunda 8 RDBMS (history)

```
camunda.migrator.c8.data-source:
 table-prefix: MY_PREFIX_ # Optional prefix for Camunda 8 RDBMS database tables
 auto-ddl: true # Automatically create/update Camunda 8 RDBMS database schema
 jdbc-url: jdbc:h2:./h2/data-migrator-target.db
 username: sa # Database username
 password: sa # Database password
 driver-class-name: org.h2.Driver
```

You can apply any HikariCP property (for example, pool size) under `camunda.migrator.c8.data-source`.

Logging

```
logging:
 level:
   root: INFO # Root logger level
```

```
  io.camunda.migrator: INFO # Migrator logging
  io.camunda.migrator.RuntimeMigrator: DEBUG # Runtime migration logging
  io.camunda.migrator.persistence.IdKeyMapper: DEBUG # ID mapping logging
file:
  name: logs/camunda-7-to-8-data-migrator.log
```

Configuration property reference

Reference for all Data Migrator configuration properties, set in the `configuration/application.yml`file.

`camunda.client`

Prefix: `camunda.client`
INFO

Read more about Camunda Client configuration options.

Property	Type	Description
`.mode`	`string`	Operation mode of the Camunda 8 client. Options: `self-managed` or `cloud`. Default: `self-managed`
`.grpc-address`	`string`	The gRPC API endpoint for Camunda 8 Platform. Default: `http://localhost:26500`

`.rest-address`	`string`	The REST API endpoint for Camunda 8 Platform. Default: `http://localhost:8088`

`camunda.migrator`

Prefix: `camunda.migrator`

Property	Type	Description
`.page-size`	`number`	Number of records to process in each page. Default: `100`
`.job-type`	`string`	Job type for actual job activation. Default: `migrator`.
`.validation-job-type`	`string`	Job type for validation purposes. Optional: falls back to `job-type` if not defined. Set to `DISABLED` to disable job type execution listener validation entirely.
`.auto-ddl`	`boolean`	Automatically create/update migrator database schema. Default: `false`

`.table-prefix`	`string`	Optional prefix for migrator database tables. Default: *(empty)*
`.data-source`	`string`	Choose if the migrator schema is created in the `C7` or `C8` data source. Should not be changed during migration, as it can lead to duplicate data migration. Default: `C7`
`.tenantIds`	`string`	Comma-separated list of tenant ids for which process instances should be migrated during runtime migration. For more information: Tenant in Runtime Default: *(empty)* (migrate only process instances without assigned tenant id)
`.database-vendor`	`string`	Database vendor for migrator schema. Options: `h2`, `postgresql`, `oracle`. Default: Automatically detected.
`.interceptors`	`array`	List of variable interceptors (built-in and custom) to configure during migration.
`.save-skip-reason`	`boolean`	Whether to persist skip reasons for entities that could not be migrated. Required when using the Cockpit plugin. Default: `false`.

camunda.migrator.interceptors.[n]

Prefix: camunda.migrator.interceptors.[n]

Property	Type	Description
class-name	string	Required. Fully qualified class name of the interceptor (built-in or custom).
enabled	boolean	Whether the interceptor is enabled. Default: true for all interceptors.
properties	map	Custom properties (key:value pairs) to configure the interceptor. Properties call setter methods on the interceptor class and pass the value.

Built-in interceptors

The following built-in interceptors are available and can be disabled:

Validators (reject unsupported types):

- io.camunda.migrator.impl.interceptor.ByteArrayVariableValidator
- io.camunda.migrator.impl.interceptor.FileVariableValidator
- io.camunda.migrator.impl.interceptor.ObjectJavaVariableValidator

Transformers (convert supported types):

- `io.camunda.migrator.impl.interceptor.PrimitiveVariableTransformer`
- `io.camunda.migrator.impl.interceptor.NullVariableTransformer`
- `io.camunda.migrator.impl.interceptor.DateVariableTransformer`
- `io.camunda.migrator.impl.interceptor.ObjectJsonVariableTransformer`
- `io.camunda.migrator.impl.interceptor.ObjectXmlVariableTransformer`
- `io.camunda.migrator.impl.interceptor.SpinJsonVariableTransformer`
- `io.camunda.migrator.impl.interceptor.SpinXmlVariableTransformer`

`camunda.migrator.c7.data-source`

Prefix: `camunda.migrator.c7.data-source`

Property	Type	Description
`.table-prefix`	`string`	Optional prefix for Camunda 7 database tables. Default: *(empty)*
`.auto-ddl`	`boolean`	Automatically create/update Camunda 7 database schema. Default: `false`
`.database-vendor`	`string`	The database vendor is automatically detected and can currently not be overridden.

`.*`		You can apply all `HikariConfig` properties.
`.jdbc-url`	`string`	JDBC connection URL for the source Camunda 7 database. Default: `jdbc:h2:mem:migrator`
`.username`	`string`	Username for Camunda 7 database connection. Default: `sa`
`.password`	`string`	Password for Camunda 7 database connection. Default: `sa`
`.driver-class-name`	`string`	JDBC driver class for Camunda 7 database. Default: `org.h2.Driver`

`camunda.migrator.c8`

Prefix: `camunda.migrator.c8`

Property	Type	Description

`.deployment-dir`	`string`	Define directory which resources like BPMN processes are automatically deployed to C8. In case multi-tenancy is enabled, please perform a manual deployment.

`camunda.migrator.c8.data-source`

Prefix: `camunda.migrator.c8.data-source`

If the `c8.data-source` configuration is absent, the RDBMS history data migrator is disabled.
INFO

Heads-up: History Data Migrator is experimental.

Property	Type	Description
`.table-prefix`	`string`	Optional prefix for Camunda 8 RDBMS database tables. Default: *(empty)*
`.auto-ddl`	`boolean`	Automatically create/update Camunda 8 RDBMS database schema. Default: `false`
`.database-vendor`	`string`	Database vendor for Camunda 8 schema. Options: `h2`, `postgresql`, `oracle`. Default: Automatically detected.

`.*`		You can apply all `HikariConfig` properties. For example:
`.jdbc-url`	`string`	JDBC connection URL for the target Camunda 8 RDBMS database. Default: `jdbc:h2:mem:migrator`
`.username`	`string`	Username for Camunda 8 database connection. Default: `sa`
`.password`	`string`	Password for Camunda 8 database connection. Default: `sa`
`.driver-class-name`	`string`	JDBC driver class for Camunda 8 database. Default: `org.h2.Driver`

`logging`

Prefix: `logging`

Property	Type	Description

`.level.root`	`string`	Root logger level. Default: `INFO`
`.level.io.camunda.migrator`	`string`	Migrator logging level. Default: `INFO`
`.file.name`	`string`	Log file location. Set to: `logs/camunda-7-to-8-data-migrator.log`. If not specified, logs are output to the console.

Troubleshooting

Troubleshooting information for common issues when running the Data Migrator.

Migration fails to start

- Verify Java 21+: `java -version`.
- Check database connectivity and credentials.
- Ensure Camunda 8 is running and accessible.
- Review your `configuration/application.yml` configuration.

Process instances are skipped

- Ensure Camunda 8 process definitions are deployed.
- Verify `migrator` execution listeners are added to None Start Events.
- Ensure flow nodes exist in both Camunda 7 and Camunda 8 models.
- Review skipped instance logs for exact reasons.

List and retry skipped instances:
./start.sh --runtime --list-skipped
./start.sh --runtime --retry-skipped

Performance issues

- Adjust `camunda.migrator.page-size`.
- Ensure database resources are sufficient.
- Check network latency between components.
- Monitor CPU/memory/disk usage.

Variable migration errors

- Check Camunda 8 variable name restrictions.
- Verify variable types are supported.
- Implement a custom `VariableInterceptor` if needed.

Debug logging

Increase logging levels to get more detail:

```
logging:
 level:
  root: INFO
  io.camunda.migrator: DEBUG
  io.camunda.migrator.RuntimeMigrator: TRACE
 file:
  name: logs/camunda-7-to-8-data-migrator.log
```

Cockpit plugin

Plugin not visible in Cockpit

- Ensure the plugin JAR is placed in the correct Camunda 7 plugins directory.
- Check Camunda 7 logs for any plugin loading errors.
- Restart Camunda 7 after deploying the plugin.

No skip data displayed

- Confirm `save-skip-reason: true` is set in the migrator configuration.
- Verify migration has been run with this setting enabled.
- Check database connectivity between the plugin and the migrator database.

Getting help

- Use the Camunda forum.
- Search existing issues: https://github.com/camunda/camunda-bpm-platform/issues.
- When creating a new issue, include: logs, config, environment, steps to reproduce.