DATA EXTRACTION PLATFORM

DUMONT

Connectors & Data Pipeline

DOCUMENTATION

v 2026.1

March 27, 2026

viglet viglet.org

VIGLET PRODUCT DOCUMENTATION

Dumont DEP

Data Extraction Platform

PRODUCT	Dumont DEP — Data Extraction Platform
VERSION	2026.1
GENERATED	March 27, 2026
WEBSITE	viglet.org
SOURCE	github.com/openviglet/dumont
LICENSE	Apache License 2.0

Dumont DEP is a data extraction platform that connects content sources — websites, databases, file systems, AEM, and WordPress — to search engines through a reliable, asynchronous processing pipeline. This document covers architecture, installation, configuration, connectors, and the developer guide.

Table of Contents

What is Dumont DEP?

Viglet Dumont DEP is an open-source data extraction platform. It connects your content — wherever it lives — to Viglet Turing ES for indexing and search.

Think of Dumont DEP as the bridge between your content sources and the search engine. It crawls websites, queries databases, scans file systems, reads AEM repositories, and pulls from WordPress — then delivers every document to Turing ES through a reliable, asynchronous pipeline.

What can you do with Dumont DEP?

Crawl websites

Point the **Web Crawler** at a starting URL and let it discover and extract content from your entire site — pages, articles, documentation. It handles authentication, URL filtering, locale detection, and incremental updates.

Index databases

The **Database Connector** runs any SQL query against Oracle, PostgreSQL, MariaDB, MySQL, or any JDBC-compatible database and turns each row into a searchable document. Complex joins, custom transformations, and batch processing are all supported.

Scan file systems

The **FileSystem Connector** walks a directory tree, extracts text from PDFs, Word documents, spreadsheets, presentations, and images (via OCR), and indexes everything with full metadata — file size, extension, modification date.

Connect to AEM

The **AEM Connector** indexes content from Adobe Experience Manager author and publish instances — supporting content fragments, delta tracking, locale mapping, and custom extension points for your specific AEM setup.

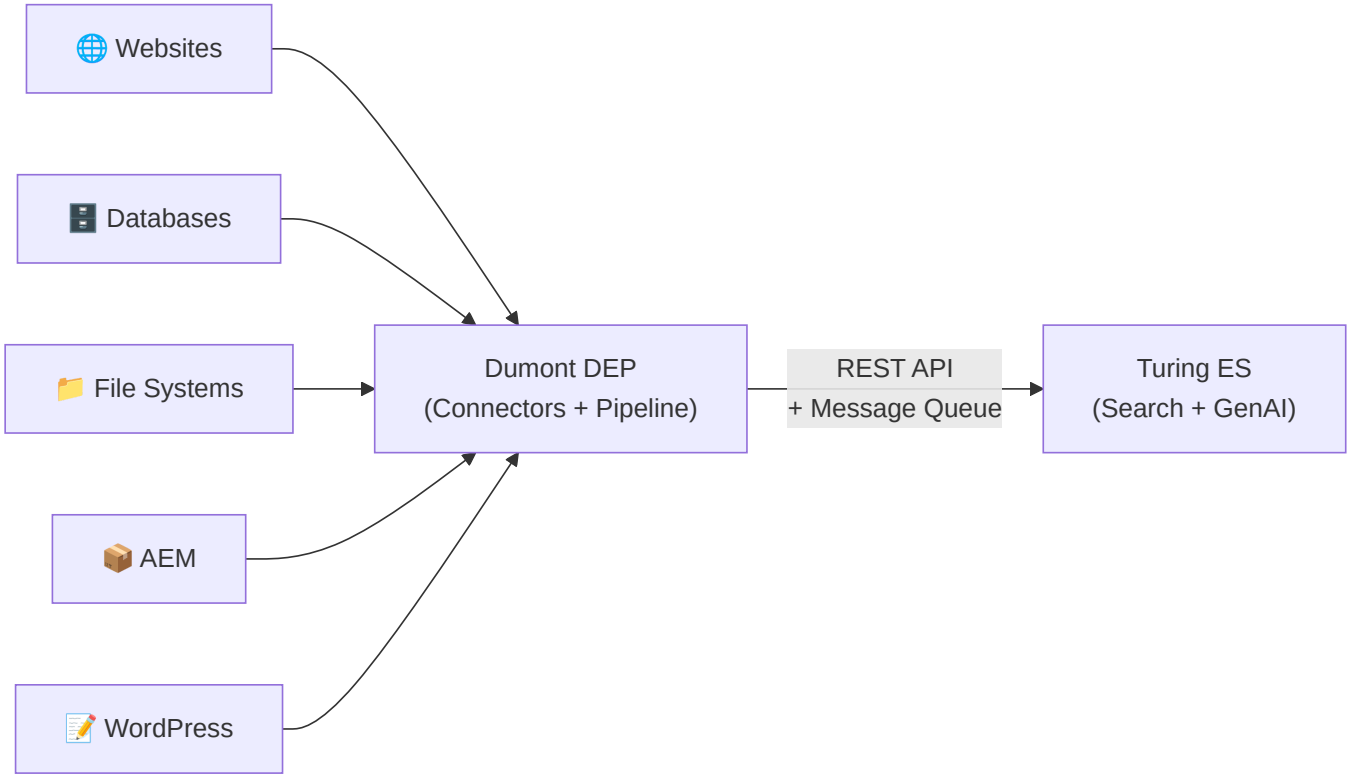
Integrate with WordPress

The **WordPress Connector** indexes posts, pages, and custom content types from WordPress installations directly into Turing ES.

Send to any search engine

Beyond Turing ES, Dumont DEP can deliver content directly to **Apache Solr** or **Elasticsearch** via pluggable indexing adapters — no Turing ES required.

How it works at a glance



Content flows from its original source through Dumont DEP connectors, into an internal message queue for reliable processing, and out to the search engine — where it becomes immediately searchable.

Key concepts

These are the main building blocks you will work with in Dumont DEP. You do not need to understand all of them before getting started — come back to each one as you need it.

CONCEPT	WHAT IT IS	LEARN MORE
Connector	A component that extracts content from a specific source type (web, database, files, AEM, WordPress)	Connectors
Indexing Plugin	The output adapter that delivers documents to a search engine — Turing ES (default), Apache Solr, or Elasticsearch	Indexing Plugins
Job Item	A single document being processed through the pipeline — with fields, metadata, and an action (index, de-index, update)	Core Concepts
Batch Processor	Groups individual job items into configurable batches (default: 50) for efficient queue delivery	Core Concepts
Message Queue	Apache Artemis (embedded) — decouples connector extraction from indexing delivery for reliability and throughput	Architecture
Processing Strategy	Priority-based rules that determine how each document is handled — index, re-index, de-index, ignore, or skip unchanged	Core Concepts
Indexing Rules	Regex-based filters that skip documents matching specific attribute patterns before they reach the queue	Core Concepts

Where to go next

Not sure where to start? Here is a suggested path depending on what you want to do:

I want to understand how Dumont DEP works → Read Core Concepts first, then Architecture.

I want to install Dumont DEP → Go to the Installation Guide.

I want to crawl a website → Go to Web Crawler Connector.

I want to index a database → Go to Database Connector.

I want to index files from a directory → Go to FileSystem Connector.

I want to connect AEM to Turing ES → Go to AEM Connector.

I want to index WordPress content → Go to WordPress Connector.

I want to send content to Solr or Elasticsearch directly → Go to Indexing Plugins.

I want to contribute or build from source → Go to the Developer Guide.

Core Concepts

This page explains the fundamental concepts of Dumont DEP in plain terms. No configuration files, no code — just the mental model you need before diving into the technical documentation.

Connectors

A **Connector** is a component that knows how to extract content from a specific type of source. Each connector:

- **Connects** to a content source (a website, a database, a file folder, an AEM instance)
- **Extracts** documents according to its configuration
- **Emits** each document as a **Job Item** into the processing pipeline

Dumont DEP ships with five connectors:

CONNECTOR	SOURCE	HOW IT WORKS
Web Crawler	Websites	Recursively follows links from a starting URL, extracts page content via HTML parsing
Database	JDBC databases	Executes SQL queries and maps each result row to a document
FileSystem	Local/network directories	Walks directory trees, extracts text from files via Apache Tika
AEM	Adobe Experience Manager	Reads content from AEM author/publish instances via the JCR API
WordPress	WordPress sites	Pulls posts, pages, and custom content types from WordPress installations

Each connector implements the `DumConnectorPlugin` interface and provides three operations: **crawl** (full extraction), **indexAll** (re-index a source), and **indexById** (index specific documents by ID).

Job Items

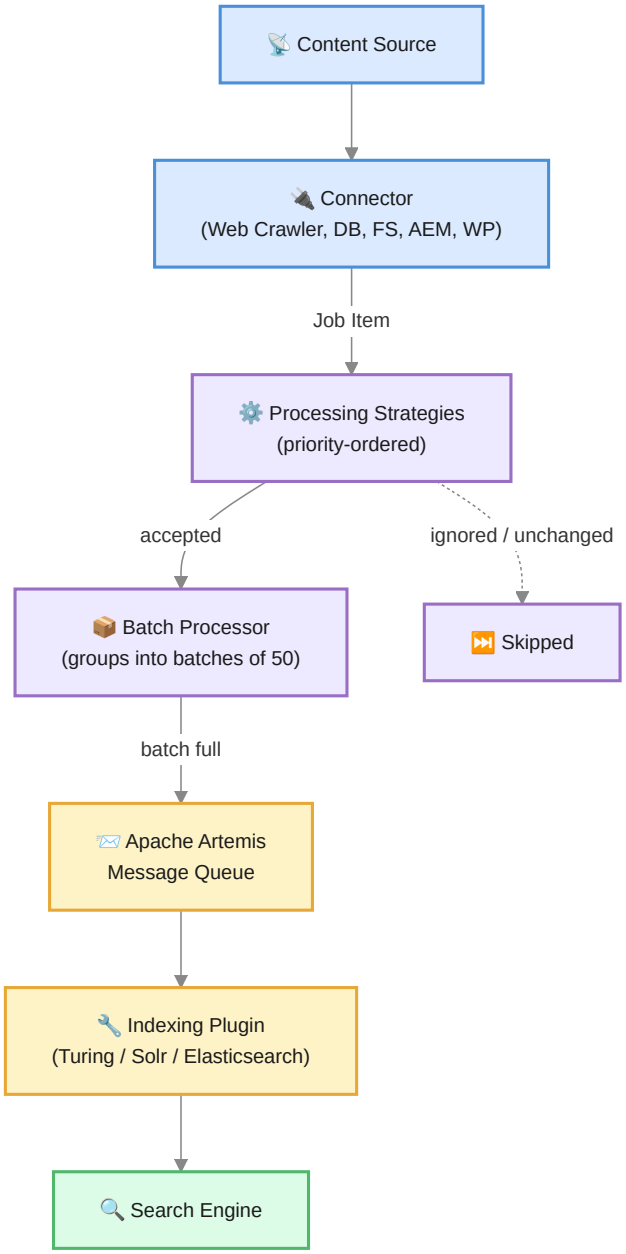
A **Job Item** is a single document moving through the pipeline. It contains:

- **Fields** — key-value pairs representing the document's content (title, text, URL, date, author, and any custom fields)
- **Action** — what to do with this document: `INDEX` (add or update) or `DELETE` (remove from the index)
- **Provider** — which connector produced this item
- **Locale** — the language/country of the content (e.g., `en_US`, `pt_BR`)

Job Items are the universal data format inside Dumont DEP. Every connector produces them, every strategy evaluates them, and every indexing plugin consumes them.

The Processing Pipeline

When a connector extracts a document, it does not send it directly to the search engine. Instead, the document passes through a multi-stage pipeline designed for reliability, efficiency, and flexibility.



Stage 1 — Extraction

The connector reads content from the source and produces Job Items. Each item includes all the fields needed for indexing.

Stage 2 — Strategy Evaluation

Every Job Item passes through a chain of **Processing Strategies**, evaluated in priority order. Each strategy decides whether the item should be indexed, re-indexed, de-indexed, ignored, or skipped:

PRIORITY	STRATEGY	WHAT IT DOES
10	**De-index**	Removes documents marked for deletion
20	**Ignore (Indexing Rules)**	Skips documents matching regex-based ignore rules
30	**Index**	Indexes new documents (not seen before)
40	**Re-index**	Updates documents whose content has changed (checksum comparison)
50	**Unchanged**	Skips documents that haven't changed since the last run

The strategy chain uses **checksum-based change detection** — each document's content is hashed, and the hash is compared against the last indexed version. Only documents that have actually changed are re-indexed.

Stage 3 — Batching

Accepted Job Items are collected by the **Batch Processor** into groups (default: 50 items per batch). This reduces the number of messages sent to the queue and improves indexing throughput.

When a batch reaches its configured size, or when the connector signals it has finished, the batch is flushed to the message queue.

Stage 4 — Queue

The batch is sent to **Apache Artemis** (embedded JMS message queue). The queue decouples extraction from delivery — if the search engine is temporarily unavailable, messages wait in the queue and are delivered when the connection is restored.

Queue messages are persisted to disk (`store/queue/`) and survive application restarts.

Stage 5 — Delivery

The **Indexing Plugin** consumes messages from the queue and delivers them to the configured search engine. Dumont DEP supports three output targets:

PLUGIN	TARGET	DESCRIPTION
Turing (default)	Viglet Turing ES	Uses the Turing Java SDK to deliver documents via REST API
Solr	Apache Solr	Uses SolrJ to add documents directly to a Solr collection
Elasticsearch	Elasticsearch	Uses the Elasticsearch Java Client for bulk indexing

Indexing Rules

Indexing Rules allow you to filter content during extraction — before it enters the pipeline. A rule defines:

- An **attribute** (a field name in the document)
- A **rule type** (currently `IGNORE`)
- One or more **values** (regex patterns)

When a document's attribute matches any of the values, the document is skipped entirely. For example, a rule with `attribute = template` and `values = [error-page, redirect]` will prevent any document with those templates from being indexed.

Indexing Rules are configured per source in the admin console and are evaluated by the **IgnoreIndexingRuleStrategy** at priority 20 — before the index/re-index/unchanged strategies run.

Change Detection

Dumont DEP tracks every document it has processed using a persistent indexing database. For each document, it stores:

- **Object ID** — the unique identifier from the source
- **Checksum** — a CRC32 hash of the document's content
- **Timestamp** — when the document was last indexed
- **Status** — the current state (preparing, indexed, de-indexed, etc.)

On subsequent runs, the connector compares the new checksum against the stored one. If they match, the document is **unchanged** and skipped. If they differ, the document is **re-indexed**. If a previously indexed document is no longer present in the source, it is **de-indexed**.

This mechanism enables efficient **incremental indexing** — only changed content is sent to the search engine, regardless of how large the source is.

Indexing Status Values

Every document's journey through the pipeline is tracked with a status code:

STATUS	MEANING
`PREPARE_INDEX`	Preparing to index the document
`PREPARE_UNCHANGED`	No changes detected since last indexing
`PREPARE_REINDEX`	Preparing a re-indexation (content changed)
`PREPARE_FORCED_REINDEX`	Forced re-indexation triggered
`RECEIVED_AND_SENT_TO_TURING`	Document received and forwarded to the search engine
`SENT_TO_QUEUE`	Document placed in the Artemis processing queue
`RECEIVED_FROM_QUEUE`	Document consumed from the queue by the indexing plugin
`INDEXED`	Document successfully indexed
`FINISHED`	Operation finished
`DEINDEXED`	Document removed from the index
`NOT_PROCESSED` / `IGNORED`	Document skipped due to an Indexing Rule or strategy decision

Ready to go deeper?

I WANT TO...	GO TO
Understand the full system architecture	Architecture
Install Dumont DEP	Installation Guide
Configure a Web Crawler	Web Crawler Connector
Index a database	Database Connector
Understand the indexing plugins	Indexing Plugins
See the full configuration reference	Configuration Reference

Viglet Dumont DEP: Installation Guide

Viglet Dumont DEP is an open-source data extraction platform that connects content sources to search engines. The connector application (`dumont-connector.jar`) provides the pipeline infrastructure — message queue, batch processing, strategies, and indexing plugins — while **connector plugins** (AEM, Web Crawler) provide the actual content extraction capabilities.

> ⚠ **THE CONNECTOR JAR ALONE DOES NOT CRAWL CONTENT**
>
> `dumont-connector.jar` is the pipeline engine. To actually extract content from sources, you must add a **connector plugin JAR** (AEM or Web Crawler) to the classpath via `-Dloader.path`. Without a plugin, the connector has no data source to crawl.

How It Works

```
dumont-connector.jar          ← Pipeline engine (queue, strategies, indexing)
  └── libs/                    ← Plugin directory (added via -Dloader.path)
      ├── aem-plugin.jar       ← AEM connector plugin
      ├── web-crawler-plugin.jar  ← Web Crawler connector plugin
      └── my-custom-plugin.jar    ← Your custom extensions (optional)
```

The connector JAR is built with Spring Boot's **ZIP layout** (PropertiesLauncher), which allows loading additional JARs from external directories at runtime. When you pass `-Dloader.path=libs`, Spring Boot scans the `libs/` directory and adds all JARs to the application classpath — making the connector plugins available to the pipeline.

Installing Java

Dumont DEP requires Java.

> ⓘ **NOTE**
>
> Dumont DEP only supports **Java 21** or later.

Set the `JAVA_HOME` environment variable and add `$JAVA_HOME/bin` to your `PATH`.

Hardware Requirements

MINIMUM	RECOMMENDED
Single 1.5 GHz processor, 1 GB available memory	Dual 2 GHz processors, 2 GB available memory

In addition:

- 500 MB of disk space for software
- Additional disk space proportional to the content being crawled (Web Crawler caches visited URLs)

Installing Dumont DEP

Option 1 — Docker (fastest)

```
docker pull openviglet/dumont:2026.1
docker run -p 30130:30130 openviglet/dumont:2026.1
```

Option 2 — Download JARs

Download `dumont-connector.jar` and the connector plugin JARs from the releases page:

```
mkdir -p /appl/viglet/dumont/server/libs

# Main connector engine
cp dumont-connector.jar /appl/viglet/dumont/server/

# Connector plugins (choose one or both)
cp aem-plugin.jar /appl/viglet/dumont/server/libs/
cp web-crawler-plugin.jar /appl/viglet/dumont/server/libs/
```

Option 3 — Build from source

```
git clone https://github.com/openviglet/dumont.git
cd dumont
mvn clean install

# Copy the connector engine
cp connector/connector-app/target/dumont-connector.jar
/appl/viglet/dumont/server/

# Copy connector plugins
mkdir -p /appl/viglet/dumont/server/libs
cp aem/aem-plugin/target/aem-plugin.jar /appl/viglet/dumont/server/libs/
cp web-crawler/wc-plugin/target/web-crawler-plugin.jar
/appl/viglet/dumont/server/libs/
```

Starting with a Connector Plugin

The key to running Dumont DEP is the `-Dloader.path` JVM property. It tells Spring Boot's PropertiesLauncher where to find the connector plugin JARs.

With AEM Plugin

```
java -Dloader.path=libs \
     -Dturing.url=http://localhost:2700 \
     -Dturing.apiKey=<YOUR_TURING_API_KEY> \
     -jar dumont-connector.jar
```

With Web Crawler Plugin

```
java -Dloader.path=libs \
     -Dturing.url=http://localhost:2700 \
     -Dturing.apiKey=<YOUR_TURING_API_KEY> \
     -jar dumont-connector.jar
```

ONE CONNECTOR PLUGIN PER JVM

Currently, only **one connector plugin** can be active per JVM instance. To run both AEM and Web Crawler, start two separate instances of `dumont-connector.jar` — each with its own `libs/` directory containing the appropriate plugin, and each on a different port.

With Custom AEM Extensions

If you have a custom AEM extension (like the `aem-plugin-sample`), add its JAR to the same `libs/` directory:

```
ls libs/
aem-plugin.jar
aem-plugin-sample.jar    # Your custom extensions

java -Dloader.path=libs \
     -Dturing.url=http://localhost:2700 \
     -Dturing.apiKey=<YOUR_TURING_API_KEY> \
     -jar dumont-connector.jar
```

The configuration JSON references extension classes by fully-qualified name (e.g., `com.viglet.dumont.connector.aem.sample.ext.DumAemExtSampleModelJson`). These classes are found on the classpath because the extension JAR is in the `libs/` directory.

> **LOADER.PATH ACCEPTS DIRECTORIES OR INDIVIDUAL JARS**
>
> You can point to a directory or a specific JAR: `-Dloader.path=libs` or `-Dloader.path=/path/to/aem-plugin.jar`

Database Configuration

By default, Dumont DEP uses **H2** as its embedded database for tracking indexing state. For production, replace with PostgreSQL.

H2 (default — development only)

No configuration needed. The database is created automatically at `./store/db/dumontDB`.

PostgreSQL (recommended for production)

Create a database and user:

```
CREATE USER dumont PASSWORD '<password>' WITH CREATEUSER;
CREATE DATABASE dumont_db;
ALTER DATABASE dumont_db OWNER TO dumont;
```

Configure via JVM properties:

```
java -Dloader.path=libs \
     -Dspring.datasource.url=jdbc:postgresql://localhost:5432/dumont_db \
     -Dspring.datasource.username=dumont \
     -Dspring.datasource.password=<password> \
     -Dspring.datasource.driver-class-name=org.postgresql.Driver \
     -Dturing.url=http://localhost:2700 \
     -Dturing.apiKey=<YOUR_TURING_API_KEY> \
     -jar dumont-connector.jar
```

Connecting to Turing ES

Dumont DEP sends content to Turing ES by default. Configure the connection:

```
turing:
  url: http://localhost:2700
  apiKey: <YOUR_TURING_API_KEY>
```

Create an API Token in **Turing ES → Administration → API Tokens** and use it as the `apiKey` value.

API TOKEN REQUIRED

Dumont DEP cannot send content to Turing ES without a valid API Token. Create one in the Turing ES admin console before starting Dumont DEP.

Creating a Linux Service

Each connector plugin runs as a separate service. Create a dedicated directory per connector with its own configuration.

Directory Layout (AEM example)

```
/appl/viglet/dumont/aem/
├── dumont-connector.jar
├── dumont-connector.properties    # Spring Boot external config
└── libs/
    └── aem-plugin.jar             # Connector plugin
```

External Properties File

Create `/appl/viglet/dumont/aem/dumont-connector.properties`:

```
turing.url=http://localhost:2700
turing.apiKey=<YOUR_TURING_API_KEY>
server.port=30130
```

Systemd Service File

As root, create `/etc/systemd/system/dumont-aem.service`:

```
[Unit]
Description=Viglet Dumont DEP (AEM)
After=syslog.target

[Service]
User=turing
Group=turing
WorkingDirectory=/appl/viglet/dumont/aem
ExecStart=/appl/java/jdk21/bin/java -Xmx512m -Xms512m \
  -Dloader.path=/appl/viglet/dumont/aem/libs/ \
  -jar /appl/viglet/dumont/aem/dumont-connector.jar \
  --spring.config.additional-location=file:/appl/viglet/dumont/aem/dumont-
connector.properties
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

Enable and start the service:

```
systemctl daemon-reload
systemctl enable dumont-aem.service
systemctl start dumont-aem.service
```

Running Multiple Connectors

Since only one connector plugin is supported per JVM, run separate services for each connector — each with its own directory, config, and port:

```
/appl/viglet/dumont/
├── aem/                          # AEM connector (port 30130)
│   ├── dumont-connector.jar
│   ├── dumont-connector.properties
│   └── libs/
│       └── aem-plugin.jar
└── webcrawler/                   # Web Crawler connector (port 30131)
    ├── dumont-connector.jar
    ├── dumont-connector.properties
    └── libs/
        └── web-crawler-plugin.jar
```

Create a separate systemd service for each (e.g., `dumont-aem.service`, `dumont-webcrawler.service`), each pointing to its own `WorkingDirectory` and properties file with a different `server.port`.

Standalone Connectors (Database & FileSystem)

The **Database** and **FileSystem** connectors are **standalone CLI tools** — they do not run as plugins inside the connector application. Instead, they connect to a running Dumont DEP instance via its REST API.

Database Connector

```
java -cp dumont-db-indexer.jar com.viglet.dumont.connector.db.DumDbImportTool
\
  --server http://localhost:30130 \
  --api-key <API_KEY> \
  --driver org.mariadb.jdbc.Driver \
  --connect "jdbc:mariadb://localhost:3306/products" \
  --query "SELECT id, name, description FROM products" \
  --site ProductCatalog \
  --locale en_US
```

FileSystem Connector

```
java -cp dumont-filesystem-indexer.jar
com.viglet.dumont.filesystem.DumFSImportTool \
  --source-dir /mnt/shared/documents \
  --server http://localhost:30130 \
  --api-key <API_KEY> \
  --site InternalDocs \
  --locale en_US
```

These tools run independently and can be scheduled via cron jobs or CI/CD pipelines.

Docker Compose

A full stack with Dumont DEP, Turing ES, Solr, and MariaDB:

```
services:
  dumont:
    image: openviglet/dumont:2026.1
    ports:
      - "30130:30130"
    environment:
      TURING_URL: http://turing:2700
      TURING_APIKEY: <YOUR_API_KEY>

  turing:
    image: openviglet/turing:2026.1
    ports:
      - "2700:2700"

  solr:
    image: solr:9
    ports:
      - "8983:8983"

  mariadb:
    image: mariadb:11
    environment:
      MARIADB_ROOT_PASSWORD: root
      MARIADB_DATABASE: turing
      MARIADB_USER: turing
      MARIADB_PASSWORD: turing
```

Verifying the Installation

After starting Dumont DEP, verify it is running:

```
curl http://localhost:30130/api/v2/connector/status
```

A successful response confirms the service is up and ready to accept connector configurations.

Directory Structure Summary

A complete production deployment with two connectors:

```
/appl/viglet/dumont/
├── aem/                                  # AEM connector instance
│   ├── dumont-connector.jar              # Pipeline engine
│   ├── dumont-connector.properties       # Turing URL, API key, port
│   ├── libs/
│   │   └── aem-plugin.jar               # AEM connector plugin
│   └── store/                            # Auto-created at runtime
│       ├── db/                           # H2 indexing database
│       ├── queue/                        # Artemis persistent queue
│       └── logs/                         # Application logs
└── webcrawler/                           # Web Crawler connector instance
    ├── dumont-connector.jar
    ├── dumont-connector.properties
    ├── libs/
    │   └── web-crawler-plugin.jar
    └── store/
```

Related Pages

PAGE	DESCRIPTION
Configuration Reference	All application.yaml properties
Architecture	System components and data flow
Connectors Overview	Available connectors and how to configure them

Configuration Reference

This page documents every significant property in the Dumont DEP `application.yaml` file. Any property can be overridden via environment variables, JVM system properties (`-Dkey=value`), or a separate `application-production.yaml` file.

OVERRIDE PATTERN

To override a property at runtime, use the environment variable convention: replace `.` and `-` with `_` and uppercase everything. For example, `dumont.indexing.provider` becomes `DUMONT_INDEXING_PROVIDER=solr`.

Full Default Configuration

```yaml
spring:
  datasource:
    url:
jdbc:h2:file:./store/db/dumontDB;DATABASE_TO_UPPER=false;CASE_INSENSITIVE_IDEN
TIFIERS=true
    username: sa
    password: ""
    driver-class-name: org.h2.Driver
  h2:
    console:
      enabled: false
      path: /h2
      settings:
        web-allow-others: true
  jpa:
    properties:
      hibernate:
        "[globally_quoted_identifiers]": true
        "[enable_lazy_load_no_trans]": true
    hibernate:
      ddl-auto: update
    show-sql: false
  artemis:
    mode: embedded
    broker-url: localhost:61616
    embedded:
      enabled: true
      persistent: true
      data-directory: store/queue
      queues: connector-indexing.queue
  jms:
    template:
      default-destination: connector-indexing.queue

server:
  port: ${PORT:30130}

dumont:
  indexing:
    provider: turing
    job:
      batch-size: 50
    solr:
      url: http://localhost:8983/solr
      collection: dumont
    elasticsearch:
```

```
        url: http://localhost:9200
        index: dumont
        username: ~
        password: ~

  turing:
    url: http://localhost:2700
    apiKey: ""

    aem.querybuilder: false
    aem.querybuilder.parallelism: 10

  logging:
    file:
      name: store/logs/dum-connector.log
    level:
      com:
        viglet: INFO
```

Property Reference

Server

PROPERTY	DEFAULT	DESCRIPTION
`server.port`	`30130`	HTTP port for the Dumont DEP application

Database

PROPERTY	DEFAULT	DESCRIPTION
`spring.datasource.url`	`jdbc:h2:file:./store/db/dumontDB`	JDBC URL for the indexing state database
`spring.datasource.username`	`sa`	Database username
`spring.datasource.password`	`""`	Database password
`spring.datasource.driver-class-name`	`org.h2.Driver`	JDBC driver class
`spring.h2.console.enabled`	`false`	Enable H2 web console — **do not enable in production**

Message Queue (Apache Artemis)

PROPERTY	DEFAULT	DESCRIPTION
`spring.artemis.mode`	`embedded`	`embedded` (default) or `native` (external broker)
`spring.artemis.broker-url`	`localhost:61616`	Broker URL when using `native` mode
`spring.artemis.embedded.enabled`	`true`	Enable the embedded broker
`spring.artemis.embedded.persistent`	`true`	Persist queue messages to disk (survives restarts)
`spring.artemis.embedded.data-directory`	`store/queue`	Directory for persisted queue data
`spring.artemis.embedded.queues`	`connector-indexing.queue`	Queue name created on startup

Indexing Configuration

PROPERTY	DEFAULT	DESCRIPTION
`dumont.indexing.provider`	`turing`	Output target: `turing`, `solr`, or `elasticsearch`
`dumont.indexing.job.batch-size`	`50`	Number of Job Items per batch before queue delivery

Turing ES Connection

PROPERTY	DEFAULT	DESCRIPTION
`turing.url`	`http://localhost:2700`	Base URL of the Turing ES instance
`turing.apiKey`	`""`	API Token for authenticating with Turing ES

Solr Direct Connection

PROPERTY	DEFAULT	DESCRIPTION
`dumont.indexing.solr.url`	`http://localhost:8983/solr`	Apache Solr base URL
`dumont.indexing.solr.collection`	`dumont`	Solr collection name

Elasticsearch Direct Connection

PROPERTY	DEFAULT	DESCRIPTION
`dumont.indexing.elasticsearch.url`	`http://localhost:9200`	Elasticsearch base URL
`dumont.indexing.elasticsearch.index`	`dumont`	Elasticsearch index name
`dumont.indexing.elasticsearch.username`	*(none)*	Optional authentication username
`dumont.indexing.elasticsearch.password`	*(none)*	Optional authentication password

AEM QueryBuilder

PROPERTY	DEFAULT	DESCRIPTION
`dumont.aem.querybuilder`	`false`	Enable QueryBuilder-based content discovery instead of tree traversal during full indexing
`dumont.aem.querybuilder.parallelism`	`10`	Number of parallel threads for processing discovered paths

Logging

PROPERTY	DEFAULT	DESCRIPTION
`logging.file.name`	`store/logs/dum-connector.log`	Log file path
`logging.level.com.viglet`	`INFO`	Log level for Dumont DEP application code

Common Production Overrides

A minimal production override:

```yaml
spring:
  datasource:
    url: jdbc:postgresql://db-host:5432/dumont_db
    username: dumont
    password: strong_password
    driver-class-name: org.postgresql.Driver
  h2:
    console:
      enabled: false

dumont:
  indexing:
    provider: turing

turing:
  url: https://search.yourcompany.com
  apiKey: your-turing-api-token

server:
  port: 30130
```

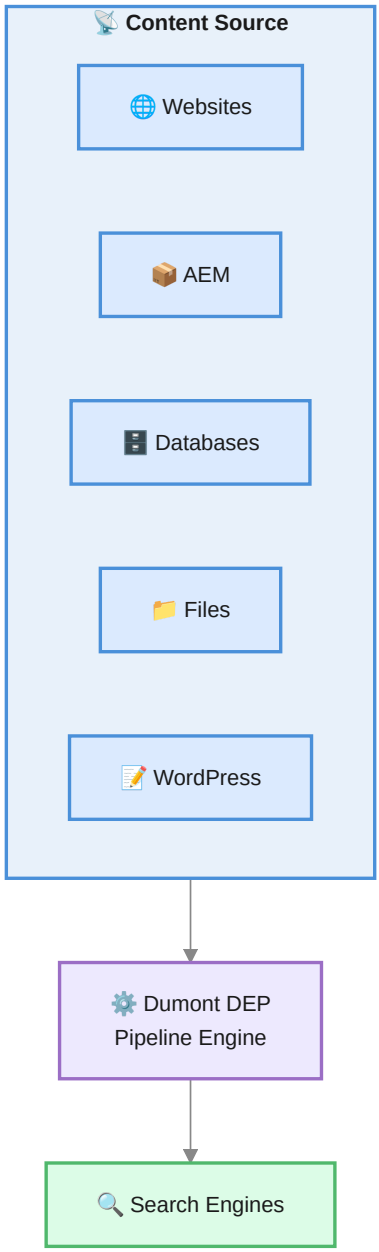
Dumont DEP — Architecture Overview

Introduction

Viglet Dumont DEP is a modular data extraction platform that runs connectors independently and delivers indexed documents to a search engine via an asynchronous message queue. It is designed as a companion application to Viglet Turing ES, but can also deliver content directly to Apache Solr or Elasticsearch.

System Overview

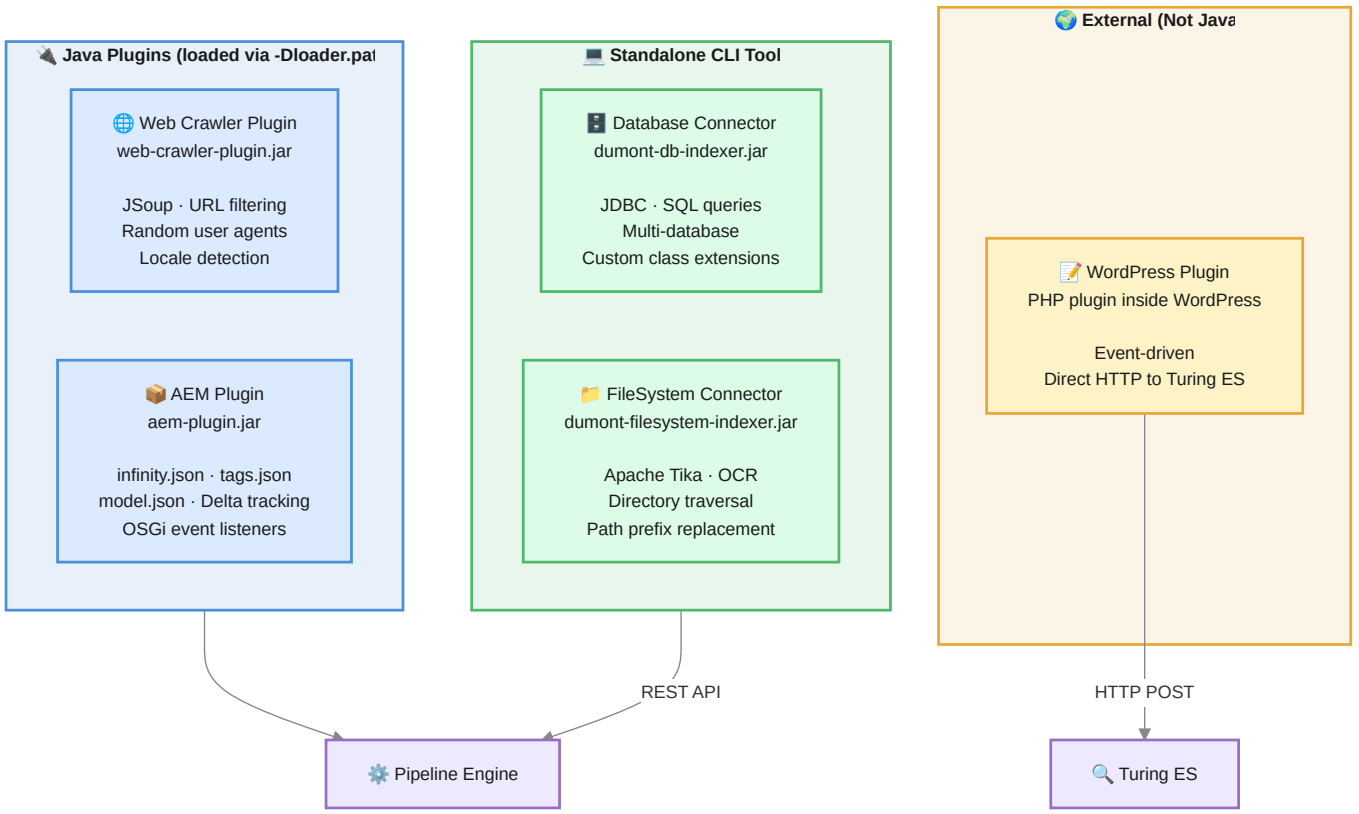
The system has three layers: content sources on the left, the Dumont DEP pipeline in the center, and search engines on the right.



Each numbered block is detailed in its own diagram below.

① Connectors — How Content Enters the Pipeline

Connectors extract content and feed it into the pipeline. They come in three forms: Java plugins, standalone CLI tools, and the WordPress PHP plugin.



TYPE	CONNECTORS	HOW THEY CONNECT
Java Plugins	Web Crawler, AEM	Loaded into `dumont-connector.jar` via `-Dloader.path` — one plugin per JVM
Standalone CLI	Database, FileSystem	Separate JARs that connect to a running Dumont DEP instance via REST API
PHP Plugin	WordPress	Installed inside WordPress — sends content directly to Turing ES, bypasses Dumont

For details on each connector, see Connectors Overview.

② Pipeline Engine — How Content Is Processed

Once a connector produces a Job Item, it passes through a multi-stage pipeline before reaching the search engine.

STAGE	COMPONENT	WHAT IT DOES
①	**Job Item**	A single document with fields, an action (INDEX / DELETE), and a locale
②	**Processing Strategies**	Priority chain: deindex (P10) → ignore rules (P20) → index new (P30) → reindex changed (P40) → skip unchanged (P50)
③	**Batch Processor + Queue**	Groups items into batches of 50, sends to Apache Artemis persistent queue
④	**Indexing Plugin**	Delivers to Turing ES (default), Apache Solr, or Elasticsearch

The **Indexing DB** stores checksums and status for every processed document — enabling incremental indexing on subsequent runs.

For the conceptual explanation of each stage, see Core Concepts — The Processing Pipeline.

③ Data Flow — Indexing Sequence

The complete sequence from content source to search engine:

Internal Module Structure

MODULE	PACKAGE	RESPONSIBILITY
Connector Core	`connector`	Plugin interface, session management, and REST API controllers
Processing Strategies	`connector/strategy`	Priority-based chain — index, re-index, de-index, ignore, or skip
Batch Processor	`connector/batch`	Thread-safe buffer that groups Job Items before queue delivery
Message Queue	`connector/queue`	JMS listener on Apache Artemis — delegates to indexing plugins
Indexing Plugins	`connector/indexing`	Output adapters: Turing ES, Apache Solr, Elasticsearch
Web Crawler	`web-crawler`	JSoup, URL filtering, authentication, locale detection
Database	`db`	JDBC queries, batch chunking, multi-database support
FileSystem	`filesystem`	Apache Tika text extraction, OCR, metadata mapping
AEM	`aem`	infinity.json, tags, model.json, delta tracking, custom extensions
WordPress	`wordpress`	PHP plugin — event-driven indexing inside WordPress
Commons	`commons`	Shared models, interfaces, utilities
AEM Commons	`aem-commons`	AEM extension interfaces (published to Maven Central)
DB Commons	`db-commons`	DB extension interface (published to Maven Central)

Technology Stack

LAYER	TECHNOLOGY	NOTES
Runtime	Java 21	Minimum supported version
Framework	Spring Boot 4.0.3	JMS, caching, async, scheduling
Message Broker	Apache Artemis	Embedded, persistent queues
Database	H2 (dev) / PostgreSQL (prod)	Indexing state, checksums, config
HTML Parsing	JSoup 1.22.1	Web Crawler
Text Extraction	Apache Tika 3.2.3	FileSystem — PDF, DOCX, images (OCR)
Search Clients	Turing Java SDK, SolrJ 10.0.0, ES Client 9.3.2	One active per deployment
Build	Apache Maven	Multi-module project

Deployment Topologies

Development

```
Dumont DEP (H2 embedded + Artemis embedded)
    → Turing ES (http://localhost:2700)
```

Production

```
Dumont DEP + PostgreSQL
    → Turing ES + Apache Solr
```

Direct Indexing (without Turing ES)

```
Dumont DEP + PostgreSQL
    → Apache Solr (direct via SolrJ)
    → Elasticsearch (direct via ES Client)
```

Related Pages

PAGE	DESCRIPTION
Core Concepts	Pipeline stages, strategies, and change detection
Connectors Overview	All connectors and deployment types
Indexing Plugins	Turing ES, Solr, Elasticsearch output targets
Installation Guide	Setup with `-Dloader.path` and systemd
Turing ES — Architecture	Search-side architecture — indexing reception, search flow, and deployment topologies

Indexing Plugins

An **Indexing Plugin** is the output adapter that delivers processed documents from the Dumont DEP pipeline to a search engine. Dumont DEP supports three targets — you choose one per deployment.

Available Plugins

PLUGIN	TARGET	CLIENT LIBRARY	USE CASE
Turing (default)	Viglet Turing ES	Turing Java SDK 2026.1.17	Full enterprise search with GenAI, facets, spotlights
Solr	Apache Solr	SolrJ 10.0.0	Direct Solr indexing without Turing ES
Elasticsearch	Elasticsearch	ES Java Client 9.3.2	Direct Elasticsearch indexing without Turing ES

Turing ES Plugin (default)

The default plugin delivers documents to Turing ES via its REST API, using the official Turing Java SDK.

Configuration

```yaml
dumont:
  indexing:
    provider: turing

turing:
  url: http://localhost:2700
  apiKey: your-turing-api-token
```

PROPERTY	DESCRIPTION
`turing.url`	Base URL of the Turing ES instance
`turing.apiKey`	API Token created in Turing ES → Administration → API Tokens

How It Works

Receives a batch of Job Items from the message queue

Creates a `TurSNServer` instance using the Turing Java SDK

Calls `TurSNJobUtils.importItems()` to submit the batch

Turing ES validates each document against the target SN Site configuration

Documents are queued internally in Turing ES for Solr indexing

API TOKEN REQUIRED

The Turing plugin cannot deliver content without a valid API Token. Create one in **Turing ES → Administration → API Tokens** before starting Dumont DEP.

Apache Solr Plugin

The Solr plugin delivers documents directly to an Apache Solr collection, bypassing Turing ES entirely. Use this when you want Dumont DEP as a pure data extraction tool without Turing ES features.

Configuration

```
dumont:
  indexing:
    provider: solr
    solr:
      url: http://localhost:8983/solr
      collection: my-collection
```

PROPERTY	DESCRIPTION
`dumont.indexing.solr.url`	Apache Solr base URL
`dumont.indexing.solr.collection`	Target Solr collection name

How It Works

1. Receives a batch of Job Items from the message queue
2. Converts each Job Item into a `SolrInputDocument`
3. Adds all documents to the Solr collection via SolrJ
4. Commits the changes

The Solr client is cleaned up automatically when the application shuts down (`@PreDestroy`).

Elasticsearch Plugin

The Elasticsearch plugin delivers documents directly to an Elasticsearch index using bulk requests.

Configuration

```yaml
dumont:
  indexing:
    provider: elasticsearch
    elasticsearch:
      url: http://localhost:9200
      index: my-index
      username: ~
      password: ~
```

PROPERTY	DESCRIPTION
`dumont.indexing.elasticsearch.url`	Elasticsearch base URL
`dumont.indexing.elasticsearch.index`	Target index name
`dumont.indexing.elasticsearch.username`	Optional authentication username
`dumont.indexing.elasticsearch.password`	Optional authentication password

How It Works

1. Receives a batch of Job Items from the message queue
2. Builds a bulk request containing all documents
3. Submits the bulk request to Elasticsearch
4. Logs any per-document errors from the bulk response

Authentication is optional — leave `username` and `password` empty for unauthenticated clusters.

Why Use Turing ES Instead of Solr or Elasticsearch Directly?

The Solr and Elasticsearch plugins deliver raw documents to the search engine — your application is responsible for everything else: building queries, rendering facets, managing spotlights, handling locales, and building the search UI.

Turing ES adds an entire enterprise search platform on top of the search engine. Here's what you get by choosing the Turing plugin over direct Solr or Elasticsearch indexing:

Search Experience Layer

CAPABILITY	WITH TURING ES	DIRECT SOLR / ELASTICSEARCH
Faceted navigation	Configured per site — facet types, sort, AND/OR operators, secondary facets, custom facets with ranges	Build manually with query params
Spotlights	Curated results pinned to search terms, injected at configured positions	Not available — build from scratch
Targeting Rules	Filter results by user profile (department, role, country) at query time	Not available
Merge Providers	Combine documents from two connectors into one enriched result using a join key	Not available
Spell check	Built-in with auto-correction mode	Configure Solr/ES suggester manually
Autocomplete	Ready-to-use endpoint	Configure Solr/ES suggester manually
More Like This	One toggle per site	Configure MLT handler manually
Result ranking	Boost rules with conditions and weights via admin UI	Write boost queries manually
Highlighting	Configurable HTML tags per site	Configure highlight params manually
Self-describing JSON	Response includes pre-built links for pagination, facet filters, locale switching — the front-end is a pure rendering layer	Build all query logic client-side

Generative AI & RAG

CAPABILITY	WITH TURING ES	DIRECT SOLR / ELASTICSEARCH
RAG (Retrieval-Augmented Generation)	Documents are automatically embedded as vectors during indexing — users ask questions in natural language and get grounded answers	Not available
AI Agents	Compose assistants that combine LLM + search + web browsing + code execution + MCP tools	Not available
Chat interface	Built-in UI with direct LLM, Semantic Navigation, and AI Agent tabs	Not available
Knowledge Base	Upload files to MinIO — automatically indexed as vector embeddings for RAG	Not available
LLM providers	Anthropic Claude, OpenAI, Azure OpenAI, Google Gemini, Ollama — configured via admin UI	Not available
Tool calling	27 native tools across 7 categories available to AI Agents	Not available
Token usage monitoring	Dashboard showing LLM consumption by model, day, and month	Not available

Administration & Operations

CAPABILITY	WITH TURING ES	DIRECT SOLR / ELASTICSEARCH
Admin console	Browser-based React UI for all configuration	Solr Admin UI (limited) / Kibana (separate)
Multi-site	Multiple SN Sites on one instance, each with independent fields, facets, and AI settings	Manage collections/indices manually
Multi-language	Locale-aware indexing and search with per-locale Solr cores	Configure manually per core/index
Connector management	Integration page with monitoring, stats, double-check, and indexing manager	Not available
Search metrics	Top search terms by day/week/month/all-time per site	Not available out of the box
Application logs	MongoDB-backed log viewer in the admin console	External log tools required
Security (SSO)	Keycloak OAuth2/OIDC with SAML, LDAP, social login, and MFA	Configure separately per product

Integration

CAPABILITY	WITH TURING ES	DIRECT SOLR / ELASTICSEARCH
REST API	Self-describing search response with pre-built navigation links	Raw query/response
GraphQL	Built-in endpoint	Not available
Java SDK	Official typed client on Maven Central	Use SolrJ / ES Client directly
JavaScript SDK	Official `@viglet/turing-sdk` on npm	No official SDK

When to Use Direct Solr or Elasticsearch

The Solr and Elasticsearch plugins are appropriate when:

- You already have a search infrastructure and only need Dumont DEP as a **data extraction tool**
- You have your own search UI and query layer built on top of Solr/Elasticsearch
- You don't need the GenAI, faceted navigation, or admin console features
- You're integrating with a third-party system that requires direct Solr/Elasticsearch access

For all other cases — especially when building a new search experience — **Turing ES is the recommended target** because it provides a complete, ready-to-use enterprise search platform with GenAI capabilities on top of the same Apache Solr engine.

Switching Plugins

Change the active plugin by setting `dumont.indexing.provider`:

```
# Via JVM property
java -Ddumont.indexing.provider=solr -jar viglet-dumont.jar

# Via environment variable
DUMONT_INDEXING_PROVIDER=elasticsearch
java -jar viglet-dumont.jar
```

Only one plugin is active per deployment. All connectors share the same output target.

Related Pages

PAGE	DESCRIPTION
Configuration Reference	All application.yaml properties
Architecture	Where indexing plugins fit in the pipeline
Turing ES — REST API	Turing ES indexing API reference
Turing ES — Semantic Navigation	SN Sites, facets, spotlights, and search configuration
Turing ES — RAG	Retrieval-Augmented Generation and vector embeddings

Connectors Overview

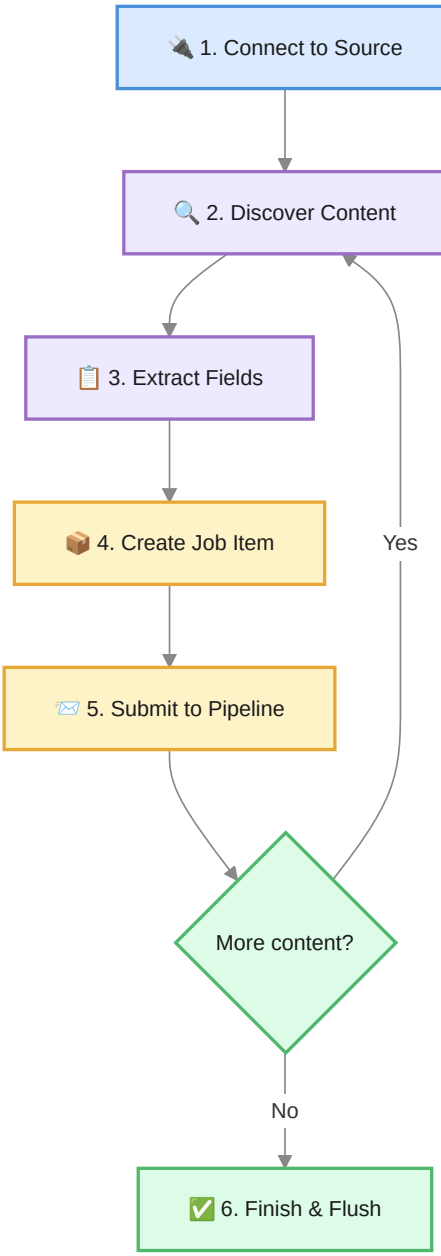
Connectors are the components that extract content from external sources and feed it into the Dumont DEP processing pipeline. Each connector specializes in a specific type of content source and knows how to navigate, extract, and map content into Job Items that the pipeline can process.

Available Connectors

CONNECTOR	SOURCE TYPE	DEPLOYMENT	ARTIFACT
Web Crawler	Websites	**Java Plugin** — loaded via `-Dloader.path`	`web-crawler-plugin.jar`
AEM	Adobe Experience Manager	**Java Plugin** — loaded via `-Dloader.path`	`aem-plugin.jar`
Database	JDBC databases	**Standalone Java CLI** — runs independently	`dumont-db-indexer.jar`
FileSystem	Local/network directories	**Standalone Java CLI** — runs independently	`dumont-filesystem-indexer.jar`
WordPress	WordPress sites	**PHP Plugin** — installed inside WordPress	`viglet-turing-for-wordpress/`

How Connectors Work

Every connector follows the same lifecycle:



Connect — Establish a connection to the content source (HTTP, JDBC, file handle, JCR session)

Discover — Find content to process (follow links, execute query, list files, traverse nodes)

Extract — Pull field values from each content item (title, text, URL, date, custom fields)

Create — Build a Job Item with the extracted fields, an action (INDEX/DELETE), and metadata

Submit — Pass the Job Item into the processing pipeline (strategies → batch → queue)

Finish — Flush any remaining items in the batch processor and signal completion

Connector Interface

All connectors implement the `DumConnectorPlugin` interface:

METHOD	DESCRIPTION
`crawl()`	Full extraction — discover and process all content from the source
`indexAll(source)`	Re-index all content from a specific source
`indexById(source, contentIds)`	Index specific documents by their IDs
`getProviderName()`	Returns the connector's identifier (e.g., `web-crawler`, `database`)

Connector Plugins vs. Standalone Tools

Dumont DEP connectors are distributed in two forms:

Connector Plugins (AEM, Web Crawler)

The **AEM** and **Web Crawler** connectors are **plugin JARs** that run inside the `dumont-connector.jar` pipeline. They must be placed in a `libs/` directory and loaded via Spring Boot's `-Dloader.path`:

```
# Directory layout
dumont-connector.jar
libs/
  ├── aem-plugin.jar
  └── web-crawler-plugin.jar

# Launch with plugins on the classpath
java -Dloader.path=libs -jar dumont-connector.jar
```

> **DUMONT-CONNECTOR.JAR ALONE DOES NOT CRAWL**
>
> The connector JAR provides only the pipeline infrastructure (queue, strategies, indexing). Without a plugin JAR on the classpath, there is no data source to extract content from. You must add exactly **one** connector plugin via `-Dloader.path`.

> **ONE PLUGIN PER JVM INSTANCE**
>
> Only **one connector plugin** can be loaded per JVM instance. To run multiple connectors (e.g., AEM and Web Crawler), start separate `dumont-connector.jar` instances — each with its own plugin and port.

Standalone CLI Tools (Database, FileSystem)

The **Database** and **FileSystem** connectors are **standalone command-line tools** — separate JARs that run independently and connect to a running Dumont DEP instance via REST API:

```
# Database import (standalone JAR)
java -cp dumont-db-indexer.jar com.viglet.dumont.connector.db.DumDbImportTool
\
  --server http://localhost:30130 \
  --api-key <API_KEY> \
  --driver org.mariadb.jdbc.Driver \
  --connect "jdbc:mariadb://localhost:3306/products" \
  --query "SELECT id, name, description, price FROM products" \
  --site ProductCatalog \
  --locale en_US

# FileSystem import (standalone JAR)
java -cp dumont-filesystem-indexer.jar
com.viglet.dumont.filesystem.DumFSImportTool \
  --source-dir /mnt/shared/documents \
  --server http://localhost:30130 \
  --api-key <API_KEY> \
  --site InternalDocs
```

These tools can be scheduled via cron jobs or CI/CD pipelines.

Managing Connectors via the Turing ES Console

The AEM and Web Crawler connector plugins can be managed through the **Turing ES Admin Console**. To connect a running `dumont-connector.jar` instance to the Turing ES UI:

1. Open the Turing ES Admin Console
2. Navigate to **Enterprise Search → Integration**
3. Click **New** to create a new integration instance
4. Set the **Integration Type** (AEM or Web Crawler)
5. Set the **Endpoint** to the URL of your Dumont DEP connector instance (e.g., `http://localhost:30130`)
6. Enable the integration

Once connected, the Turing ES console provides a graphical interface for:

- Configuring sources, content types, and field mappings
- Triggering full indexing and re-indexing operations
- Monitoring indexing progress in real time

Viewing indexing statistics and status

Running double-check consistency validation

For full details on the Integration UI — including monitoring, indexing stats, and double-check — see the Turing ES Integration documentation.

For AEM-specific configuration (sources, content types, author/publish, delta tracking, locales, indexing rules) see Turing ES AEM Connector documentation.

Common Configuration Pattern

Every connector needs at least these pieces of information:

SETTING	DESCRIPTION
Source	Where to read content (URL, connection string, directory path, AEM endpoint)
Credentials	Authentication (username/password, API key, or none)
Target SN Site	The Turing ES Semantic Navigation Site that will receive the content
Locale	The language/country code for the content (e.g., `en_US`)
Field Mapping	How source fields map to search index fields

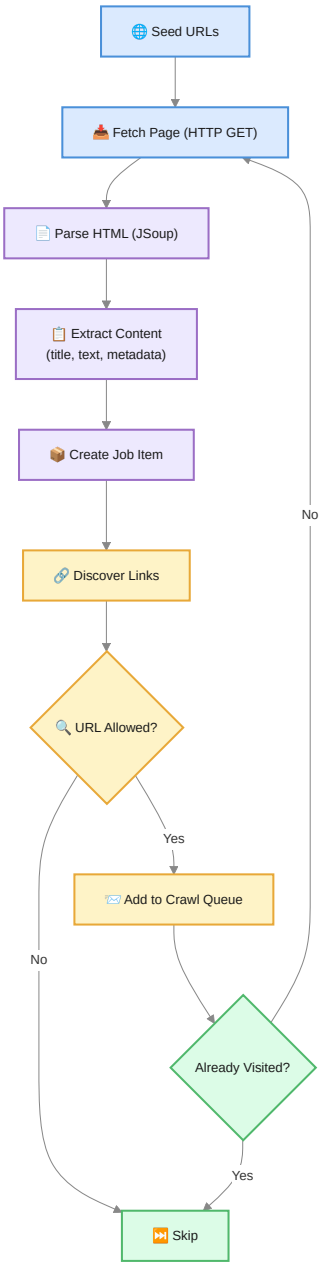
How Each Connector Is Configured

CONNECTOR	CONFIGURATION METHOD	WHERE
Web Crawler	Turing ES Admin Console	Turing ES → Integration
AEM	JSON file + Turing ES Admin Console	`export/` directory + Turing ES → Integration
Database	CLI parameters	Command-line — see Database Connector
FileSystem	CLI parameters	Command-line — see FileSystem Connector
WordPress	WordPress Admin UI	WordPress → Settings → Viglet Dumont

Web Crawler Connector

The Web Crawler connector recursively discovers and extracts content from websites. Starting from one or more seed URLs, it follows links, extracts page content using JSoup HTML parsing, and delivers each page as a searchable document.

How It Works



Start with one or more **seed URLs** (starting points for the crawl)

Fetch each page via HTTP GET with a randomized user agent

Parse the HTML response with JSoup

Extract content fields based on configured **attribute mappings**

Create a Job Item and submit it to the pipeline

Discover all links on the page

Filter links against **allow/deny patterns** and file extension rules

Add new, unvisited URLs to the crawl queue

Repeat until no more URLs remain

Key Features

FEATURE	DESCRIPTION
Recursive crawling	Follows links from seed URLs to discover all reachable pages
URL filtering	Allow and deny patterns (regex) control which URLs are crawled
File extension filtering	Configurable list of file extensions to include or exclude
Authentication	Basic HTTP authentication for protected sites
Random user agents	Generates random browser user-agent strings to avoid blocking
Visited URL tracking	CRC32 checksums prevent re-visiting the same URL in a single crawl
Locale detection	Extensible interface (`DumWCExtLocaleInterface`) for detecting content language
URL normalization	Normalizes URLs to avoid duplicate crawling of the same page
Incremental indexing	Checksum-based change detection — only re-indexes pages that changed

Configuration

Source Settings

FIELD	DESCRIPTION
Name	Identifier for this crawl source
Starting URLs	One or more seed URLs where the crawl begins
Endpoint	Base URL of the website
Username / Password	Optional HTTP Basic authentication credentials

URL Filtering

FIELD	DESCRIPTION
Allow URL Patterns	Regex patterns — only URLs matching these patterns are crawled
Deny URL Patterns	Regex patterns — URLs matching these patterns are skipped
Allowed File Extensions	List of file extensions to include (e.g., `.html`, `.php`, `.aspx`)

Attribute Mapping

Define which parts of the HTML page map to which fields in the search index:

FIELD	DESCRIPTION
Title	CSS selector or extraction rule for the page title
Text	CSS selector for the main content body
URL	The page URL (extracted automatically)
Date	CSS selector or meta tag for the publication date
Custom fields	Additional attribute mappings for any HTML element or meta tag

Example: Crawling a Documentation Site

Crawl a documentation site starting from the homepage, only following links within the `/docs/` path:

SETTING	VALUE
Starting URL	`https://docs.example.com/`
Allow Pattern	`https://docs\.example\.com/docs/.*`
Deny Pattern	`.*\.(css|js|png|jpg|gif|svg)$`
Target SN Site	`Documentation`
Locale	`en_US`

The crawler will:

- Start at the homepage
- Follow all links matching `/docs/` paths
- Skip links to stylesheets, scripts, and images
- Extract each page's title, body text, and URL
- Send each page to the `Documentation` SN Site in Turing ES

Locale Detection

The Web Crawler supports automatic locale detection via the `DumWCExtLocaleInterface` extension point. Implement this interface to determine the locale of each page based on:

- URL path patterns (e.g., `/en/`, `/pt-br/`)
- HTML `lang` attribute
- Content-Language HTTP header
- Custom logic specific to your site structure

If no locale extension is configured, the default locale from the source configuration is used for all pages.

Limitations

The crawler follows **HTML links only** — it does not execute JavaScript. Single-page applications (SPAs) that render content via JavaScript are not supported without a pre-rendering solution.

robots.txt — The crawler does not currently enforce robots.txt directives. Ensure you have permission to crawl the target site.

Rate limiting — There is no built-in rate limiter. For large sites, monitor server load and consider adding delays between requests.

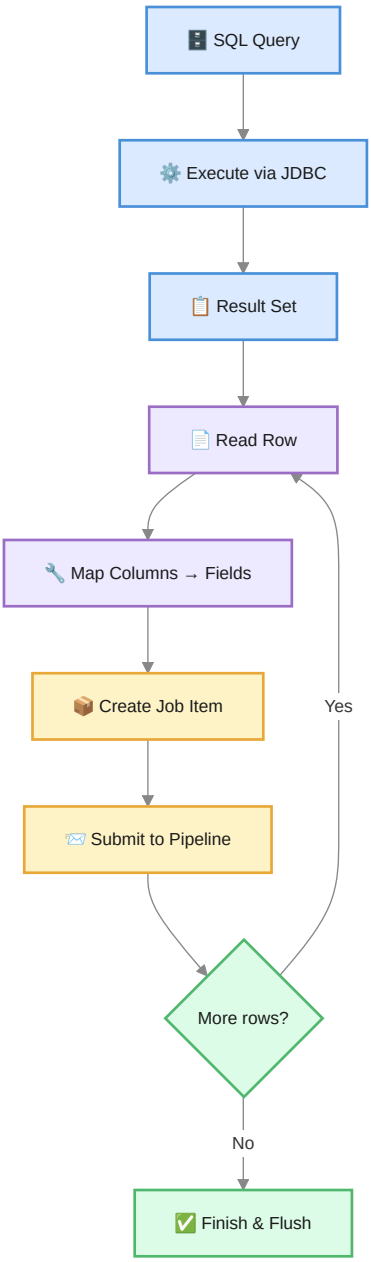
Related Pages

PAGE	DESCRIPTION
Connectors Overview	All available connectors
Core Concepts	Pipeline, strategies, and change detection
Turing ES — Integration	How Turing ES receives content from connectors

Database Connector

The Database Connector extracts content from any JDBC-compatible database by executing SQL queries and mapping each result row to a searchable document. It supports Oracle, PostgreSQL, MariaDB, MySQL, and any database with a JDBC driver.

How It Works



Execute the configured **SQL query** against the database

Iterate through the result set

Map each **column** to a search index **field** based on the attribute mapping

Create a Job Item with the mapped fields

Submit to the pipeline in configurable **chunks**

Repeat until all rows are processed

Key Features

FEATURE	DESCRIPTION
Any JDBC database	Oracle, PostgreSQL, MariaDB, MySQL, and any JDBC-compatible source
SQL flexibility	Use any SELECT query — joins, aggregations, subqueries, functions
External SQL files	Load queries from files using the `file://` protocol
Batch processing	Configurable chunk size for memory-efficient processing
Max content size	Limits per-document content size (default: 5 MB) to prevent oversized documents
Standalone CLI	Run imports from the command line without the full Dumont DEP application
Custom extensions	`DumDbExtCustomImpl` interface for custom row processing logic
Locale support	Configurable default locale for all extracted documents

CLI Parameters

The standalone Database Connector accepts the following command-line parameters:

PARAMETER	REQUIRED	DEFAULT	DESCRIPTION
`--driver`	Yes	—	JDBC driver class name
`--connect`	Yes	—	JDBC connection string
`--query` / `-q`	Yes	—	SQL query or `file://path/to/query.sql`
`--site`	Yes	—	Target Semantic Navigation Site name
`--server` / `-s`	Yes	—	Dumont DEP server URL
`--api-key` / `-a`	Yes	—	API key for authentication
`--locale`	No	`en_US`	Default locale for documents
`--chunk` / `-z`	No	*(varies)*	Batch size for processing
`--max-content-size`	No	`5`	Maximum content size in MB
`--deindex-before-importing`	No	`false`	Remove all existing documents before import

Example: Indexing a Product Catalog

SQL Query

```
SELECT
    p.id,
    p.name AS title,
    p.description AS text,
    p.price,
    c.name AS category,
    p.updated_at AS date,
    CONCAT('https://shop.example.com/products/', p.slug) AS url
FROM products p
JOIN categories c ON p.category_id = c.id
WHERE p.active = 1
```

Standalone Import

```
java -cp dumont-db.jar com.viglet.dumont.db.DumDbImportTool \
  --server http://localhost:30130 \
  --api-key your-api-key \
  --driver org.mariadb.jdbc.Driver \
  --connect "jdbc:mariadb://localhost:3306/shop?user=reader&password=secret" \
  --query "file:///opt/queries/products.sql" \
  --site ProductCatalog \
  --locale en_US \
  --chunk 100
```

This will:

- Connect to the MariaDB database
- Execute the SQL query from the external file
- Map each row to a document (columns become fields)
- Send documents in batches of 100 to the `ProductCatalog` SN Site

Supported Databases

DATABASE	DRIVER	CONNECTION STRING EXAMPLE
MariaDB	`org.mariadb.jdbc.Driver`	`jdbc:mariadb://host:3306/dbname`
MySQL	`com.mysql.cj.jdbc.Driver`	`jdbc:mysql://host:3306/dbname`
PostgreSQL	`org.postgresql.Driver`	`jdbc:postgresql://host:5432/dbname`
Oracle	`oracle.jdbc.OracleDriver`	`jdbc:oracle:thin:@host:1521/service`

Tips

Use aliases in your SQL query to match the field names expected by Turing ES (e.g., `SELECT name AS title`).

External SQL files (`file://` protocol) keep complex queries maintainable and version-controlled.

De-index before importing (`--deindex-before-importing`) is useful for full refreshes — it removes stale documents that no longer exist in the database.

Chunk size affects memory usage: larger chunks are faster but use more memory. Start with 100 and adjust based on document size.

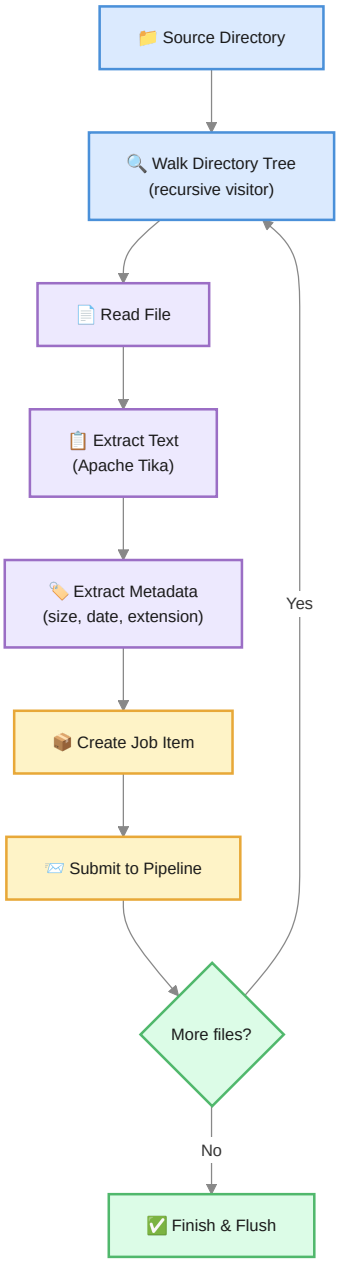
CUSTOMIZING THE DATABASE CONNECTOR

Need to transform or enrich rows during import? See Extending the Database Connector for the `DumDbExtCustomImpl` interface and step-by-step guide.

FileSystem Connector

The FileSystem Connector walks a directory tree, extracts text and metadata from files using Apache Tika, and indexes everything as searchable documents. It supports PDFs, Word documents, spreadsheets, presentations, plain text, HTML, and images (with OCR).

How It Works



Start at the configured **source directory**

Recursively traverse all subdirectories

For each file, extract text content using **Apache Tika**

Collect file metadata (size, modification date, extension, MIME type)

Create a Job Item with the extracted content and metadata

Submit to the pipeline in configurable batches

Repeat until all files are processed

Key Features

FEATURE	DESCRIPTION
Recursive traversal	Walks the full directory tree using Java's file visitor pattern
Apache Tika integration	Text extraction from 1000+ file formats
OCR support	Extract text from images and scanned PDFs (requires Tesseract)
File metadata	Captures file size, extension, modification date, and MIME type
Prefix replacement	Replace file path prefixes with custom URLs (e.g., replace local path with a web URL)
Standalone CLI	Run imports from the command line independently
Configurable batch size	Control memory usage with chunk-based processing

Supported File Formats

Apache Tika supports text extraction from a broad range of formats:

CATEGORY	FORMATS
Documents	PDF, DOCX, DOC, ODT, RTF, EPUB
Spreadsheets	XLSX, XLS, ODS, CSV
Presentations	PPTX, PPT, ODP
Web / Markup	HTML, XHTML, XML
Plain text	TXT, LOG, Markdown
Email	EML, MSG, MBOX
Images (with OCR)	PNG, JPEG, TIFF, BMP, GIF

Files that Tika cannot extract text from are silently skipped.

CLI Parameters

PARAMETER	REQUIRED	DEFAULT	DESCRIPTION
`--source-dir` / `-d`	Yes	—	Root directory to scan
`--server` / `-s`	Yes	—	Dumont DEP server URL
`--api-key` / `-a`	Yes	—	API key for authentication
`--site`	Yes	—	Target Semantic Navigation Site name
`--type` / `-t`	No	`Static File`	Content type label for all documents
`--locale`	No	`en_US`	Default locale
`--chunk` / `-z`	No	`100`	Batch size
`--file-size-field`	No	—	Field name to store file size
`--file-extension-field`	No	—	Field name to store file extension
`--prefix-from-replace`	No	—	Path prefix to replace (e.g., `/mnt/docs`)
`--prefix-to-replace`	No	—	Replacement prefix (e.g., `https://docs.example.com`)

Example: Indexing a Document Repository

```
java -cp dumont-fs.jar com.viglet.dumont.filesystem.DumFSImportTool \
  --source-dir /mnt/shared/documents \
  --server http://localhost:30130 \
  --api-key your-api-key \
  --site InternalDocs \
  --locale en_US \
  --chunk 50 \
  --file-size-field fileSize \
  --file-extension-field fileExtension \
  --prefix-from-replace /mnt/shared/documents \
  --prefix-to-replace https://intranet.example.com/docs
```

This will:

- Scan all files under `/mnt/shared/documents` recursively
- Extract text from each file using Apache Tika
- Replace the local path with a web URL for each document
- Index everything into the `InternalDocs` SN Site
- Store file size and extension as searchable/facetable fields

Path Prefix Replacement

The `--prefix-from-replace` and `--prefix-to-replace` parameters transform file paths into web-accessible URLs. This is essential when the files are served by a web server:

LOCAL PATH	AFTER REPLACEMENT
`/mnt/shared/documents/reports/q1-2026.pdf`	`https://intranet.example.com/docs/reports/q1-2026.pdf`
`/mnt/shared/documents/policies/security.docx`	`https://intranet.example.com/docs/policies/security.docx`

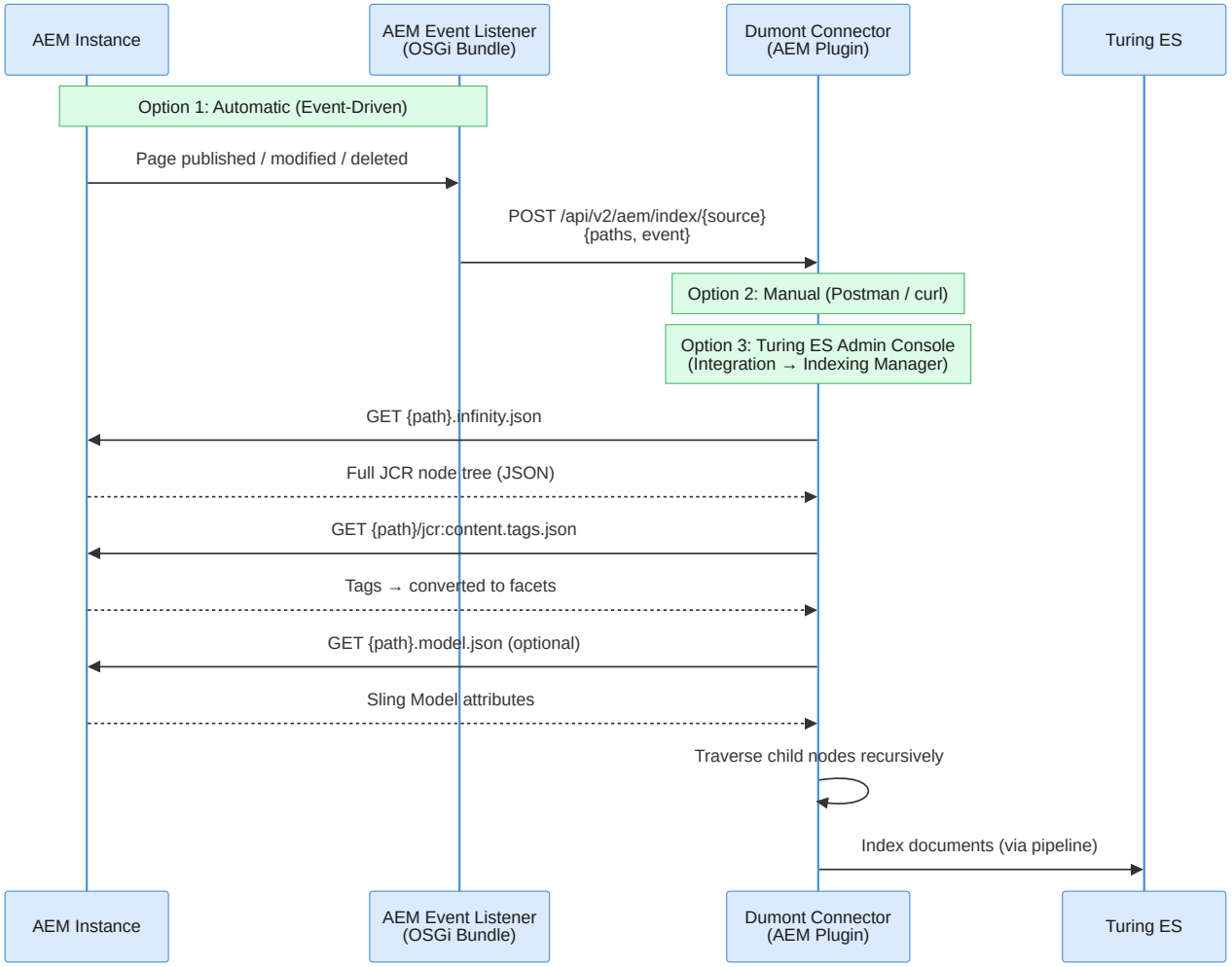
The transformed path becomes the document's **URL** field in the search index, allowing users to click through from search results to the actual file.

AEM Connector

The AEM Connector indexes content from Adobe Experience Manager (AEM) author and publish instances. It consists of two components: an **AEM server-side bundle** (OSGi event listeners installed inside AEM) and a **connector plugin** (Java JAR loaded into `dumont-connector.jar`).

How It Works

The AEM connector receives indexing requests, then accesses AEM to traverse the content tree, fetch page data, extract tags as facets, and optionally call `.model.json` for custom attributes.



Three Ways to Trigger Indexing

METHOD	HOW	WHEN TO USE
AEM Event Listeners	Install the `aem-server` OSGi bundle inside AEM — it automatically sends indexing requests when content is published, modified, or deleted	Production — real-time content sync
Manual API Call	Send a POST request to `/api/v2/aem/index/{source}` with a JSON payload containing paths and event type	Development, testing, one-off re-indexing
Turing ES Admin Console	Use **Enterprise Search → Integration → Indexing Manager** to select paths and trigger indexing/deindexing/publishing operations	Operations — selective re-indexing via UI

Content Discovery Strategies

The AEM connector supports two strategies for discovering content during a full **Index All** operation:

STRATEGY	PROPERTY	HOW IT DISCOVERS CONTENT
Tree Traversal *(default)*	`dumont.aem.querybuilder=false`	Recursively walks the content tree from the root path using `infinity.json` — follows parent→child relationships
QueryBuilder	`dumont.aem.querybuilder=true`	Uses AEM's native QueryBuilder API (`/bin/querybuilder.json`) to find all content matching the configured content type in paginated batches

QueryBuilder Discovery

When enabled, the connector queries the QueryBuilder endpoint instead of walking the tree:

```
GET http://localhost:4502/bin/querybuilder.json?
path=/content/wknd&type=cq:Page&p.hits=slim&p.limit=500&p.offset=0
```

Each page of discovered paths is immediately processed in parallel using a configurable thread pool — the full path list is never held in memory.

Enable it by adding the following properties to `application.yaml` or as JVM arguments:

```
dumont:
  aem.querybuilder: true
  aem.querybuilder.parallelism: 10   # number of parallel threads (default)
```

Or via command-line:

```
java -Ddumont.aem.querybuilder=true -Ddumont.aem.querybuilder.parallelism=10 -
jar dumont-connector.jar
```

CONTENT TYPE REQUIRED

QueryBuilder discovery requires a **Content Type** (e.g., `cq:Page`) configured in the AEM source. Without it, the command logs a warning and skips processing.

WHEN TO USE QUERYBUILDER

QueryBuilder is recommended for **large content trees** where recursive traversal is slow. It reduces the number of HTTP round-trips to AEM by discovering all paths in bulk, then fetching content in parallel. For small sites (< 1 000 pages), the default tree traversal is typically sufficient.

The Indexing Flow (Step by Step)

When the connector receives an indexing request (from any of the three triggers), it processes each path as follows:

1. Fetch the Content Node

The connector calls AEM's `infinity.json` endpoint to get the full JCR node tree:

```
GET http://localhost:4502/content/wknd/us/en/my-page.infinity.json
```

This returns the complete node hierarchy as JSON — all properties, child nodes, and metadata. The connector filters out internal nodes (prefixed with `jcr:`, `rep:`, `cq:`).

2. Extract Tags as Facets

For each page, the connector fetches tags:

```
GET http://localhost:4502/content/wknd/us/en/my-page/jcr:content.tags.json
```

Tags are **automatically converted to facets** in the search index — no manual configuration needed. Each tag becomes a filterable value in the Turing ES facet panel.

3. Fetch Model JSON (Optional — Requires Configuration)

The connector does **not** call `.model.json` by default. To enable it, you must configure a `DumAemExtContentInterface` implementation in the `models` section of the export JSON:

```
"models": [
  {
    "type": "cq:Page",
    "className":
"com.viglet.dumont.connector.aem.sample.ext.DumAemExtSampleModelJson",
    "targetAttrs": [ ... ]
  }
]
```

When this is configured, the extension class fetches the Sling Model exporter:

```
GET http://localhost:4502/content/wknd/us/en/my-page.model.json
```

This returns structured content from AEM's Sling Models — useful for extracting custom attributes like content fragment paths, component data, or experience fragment references. See Extending the AEM Connector for how to implement `DumAemExtContentInterface` and the full JSON configuration reference.

4. Traverse the Content Tree

Starting from the configured **root path** (e.g., `/content/wknd`), the connector recursively traverses all child nodes that match the configured **content type** (e.g., `cq:Page`). Each matching node goes through steps 1–3 above.

5. Map Attributes and Index

For each page, the connector:

- Applies **attribute mappings** from the configuration (global attributes + model-specific source→target mappings)
- Runs **custom extension classes** (if configured via `className`)
- Creates a **Job Item** for both author and publish environments (if enabled)
- Sends the Job Item through the Dumont DEP pipeline to Turing ES

AEM Server-Side Bundle (Event Listeners)

The `aem-server` module is an **OSGi bundle installed inside AEM**. It provides event listeners that automatically notify the Dumont connector when content changes.

Events Captured

EVENT LISTENER	AEM EVENT	DUMONT ACTION
`DumAemPageEventHandler`	Page created / modified	`INDEXING`
`DumAemPageReplicationEventHandler`	Page activated (published)	`PUBLISHING`
`DumAemPageReplicationEventHandler`	Page deactivated (unpublished)	`UNPUBLISHING`
`DumAemResourceEventHandler`	DAM asset created / modified	`INDEXING`

OSGi Configuration

The event listeners are configured in AEM via **OSGi Configuration** (AEM → Web Console → Configuration):

SETTING	DESCRIPTION
Enabled	Toggle to enable/disable automatic indexing
Host	Dumont connector URL (e.g., `http://dumont-server:30130`)
Config Name	Source name configured in the Dumont connector

HTTP Payload

When an event fires, the bundle sends:

```
POST http://dumont-server:30130/api/v2/aem/index/{configName}
Content-Type: application/json

{
  "paths": ["/content/wknd/us/en/my-page"],
  "event": "INDEXING"
}
```

Event types: `INDEXING`, `DEINDEXING`, `PUBLISHING`, `UNPUBLISHING`.

Manual API Triggering

You can trigger indexing manually via HTTP (Postman, curl, etc.):

Index Specific Paths

```
curl -X POST http://localhost:30130/api/v2/aem/index/WKND \
  -H "Content-Type: application/json" \
  -d '{
    "paths": ["/content/wknd/us/en/about"],
    "event": "INDEXING",
    "recursive": true
  }'
```

Deindex Specific Paths

```
curl -X POST http://localhost:30130/api/v2/aem/index/WKND \
  -H "Content-Type: application/json" \
  -d '{
    "paths": ["/content/wknd/us/en/old-page"],
    "event": "DEINDEXING"
  }'
```

Request Body Fields

FIELD	TYPE	DEFAULT	DESCRIPTION
`paths`	string[]	*(required)*	AEM content paths to process
`event`	string	`INDEXING`	`INDEXING`, `DEINDEXING`, `PUBLISHING`, or `UNPUBLISHING`
`recursive`	boolean	`false`	Traverse child nodes recursively
`attribute`	string	`ID`	`ID` (path-based) or `URL` (URL-based)

Source Configuration

Each AEM source defines connection details, content scope, author/publish environments, locale mappings, and delta tracking. Sources are configured in the **Turing ES Admin Console** under **Enterprise Search → Integration → [your AEM instance] → Sources**.

For the JSON configuration file used by custom extensions (attributes, models, locale paths), see Extending the AEM Connector.

General

FIELD	DESCRIPTION
Name	Source identifier
Endpoint	URL of the AEM instance (e.g., `http://localhost:4502`)
Username / Password	Credentials for authenticated access to the AEM instance

Root Path

Defines the root content path within the AEM repository from which content is traversed (e.g., `/content/wknd`). All child nodes matching the configured content type are indexed recursively from this path.

Content Types

FIELD	DESCRIPTION
Content Type	Primary content type to be indexed (e.g., `cq:Page`)
Sub Type	Optional sub-type filter within the content type

Delta Tracking

Controls incremental indexing — how the connector detects which content has changed since the last run.

FIELD	DESCRIPTION
Once Pattern	Pattern used to identify content that should only be indexed once
Delta Class	Fully-qualified Java class name responsible for detecting changed content since the last run (see Extending AEM for custom implementations)

Author / Publish

Configures which AEM environments are indexed and how they map to Turing ES Semantic Navigation Sites.

FIELD	DESCRIPTION
Author	Enable indexing from the AEM author environment
Publish	Enable indexing from the AEM publish environment
SN Site (Author)	Semantic Navigation Site that receives author content
SN Site (Publish)	Semantic Navigation Site that receives publish content
URL Prefix (Author)	URL prefix prepended to document paths in the author index
URL Prefix (Publish)	URL prefix prepended to document paths in the publish index

Locales

Maps content language codes to repository paths.

FIELD	DESCRIPTION
Default Locale	Locale used when no language-specific path is matched
Locale Class	Fully-qualified Java class name responsible for resolving document locale (see Extending AEM for custom implementations)
Locale → Path	Dynamic list mapping each locale code (e.g., `en_US`) to its root path in the repository

Source Actions

Each source has two action buttons available in the Turing ES admin console:

Index All — triggers a full indexing run for all content in this source

Reindex All — forces a full reindexation, replacing all previously indexed content

Indexing Rules

Indexing Rules allow you to filter content during indexing — for example, to exclude error pages or draft content before it reaches the search index. Rules are configured in the **Turing ES Admin Console** under **Enterprise Search → Integration → [your AEM instance] → Indexing Rules**.

FIELD	DESCRIPTION
Name	Rule identifier (required)
Description	Purpose of this rule
Source	The source this rule applies to
Attribute	Document field to evaluate (e.g., `template`)
Rule Type	How the rule is applied — currently supports **IGNORE** (skip documents that match)
Values	Dynamic list of values that trigger the rule (add or remove entries)

Example: A rule with `Attribute = template`, `Rule Type = IGNORE`, and `Values = [error-page]` will prevent any document with `template:error-page` from being indexed.

Indexing Manager

The Indexing Manager provides a stepper form in the **Turing ES Admin Console** for targeting specific documents for manual operations.

OPERATION	DESCRIPTION	COLOUR
INDEXING	Index specific content	Blue
DEINDEXING	Remove specific content from the index	Red
PUBLISHING	Publish content	Green
UNPUBLISHING	Unpublish content	Orange

Each operation step allows you to:

- Select the **Source** to operate on
- Choose the **attribute** to identify documents: **ID** or **URL**
- Enter one or more specific values (IDs or URLs)
- Expand **Advanced Settings** to toggle **Recursive** mode, which traverses child content in hierarchical repositories

Concurrency

The connector supports two execution modes:

MODE	WHEN	BEHAVIOR
Exclusive	Full crawl (`indexAll`)	Only one full crawl per source at a time
Standalone	Specific paths (event-driven / manual)	Multiple concurrent updates allowed

Reactive (parallel) processing can be enabled for large sites:

```
dumont.reactive.indexing=true
dumont.reactive.parallelism=10
```

When using **QueryBuilder discovery**, parallelism is controlled separately via `dumont.aem.querybuilder.parallelism` (default `10`). See Content Discovery Strategies above.

CUSTOMIZING THE AEM CONNECTOR

Need custom attribute extractors, delta date logic, or content processors? See Extending the AEM Connector for the full extension system, configuration JSON reference, and step-by-step guide.

Related Pages

PAGE	DESCRIPTION
AEM Event Listener	Install the OSGi event listener bundle inside AEM for real-time indexing
Extending the AEM Connector	Custom attribute extractors, content processors, and configuration JSON reference
Turing ES — Integration	General integration management — monitoring, indexing stats, and system information
Turing ES — AEM Connector	AEM integration overview from the Turing ES perspective
Turing ES — Semantic Navigation	Configure the SN Sites that receive indexed content

AEM Event Listener Setup

This guide explains how to install the Dumont AEM event listener bundle inside your AEM as a Cloud Service instance. Once installed, AEM automatically notifies the Dumont connector whenever content is published, modified, or deleted — enabling real-time search index synchronization.

What Gets Installed

The Dumont AEM Server module (`aem-server`) deploys three artifacts into AEM:

ARTIFACT	TYPE	PURPOSE
`aem-dumont.structure`	Content package	Creates the `/apps/turing` repository structure
`aem-dumont.config`	Content package	Deploys the OSGi configuration (host, configName, enabled)
`aem-dumont.core`	OSGi bundle	Event handlers that listen for page/asset changes and send HTTP requests to Dumont

Prerequisites

An AEM as a Cloud Service environment (dev, stage, or prod)

A running Dumont connector instance with the AEM plugin (accessible from AEM)

Access to your AEM project's Git repository (for Cloud Manager pipeline deployment)

Java 11+ and Maven 3.6+ for local builds

> **AEM AS A CLOUD SERVICE DEPLOYMENT**
>
> AEM as a Cloud Service does not support direct bundle installation via the Felix Console. All code must be deployed through a **Cloud Manager pipeline** from a Git repository. See Adobe's deployment documentation for details.

Option 1: Add to an Existing AEM Project

If you already have an AEM as a Cloud Service project (created from the AEM Project Archetype), you can integrate the Dumont event listeners into it.

Step 1 — Copy the Source Files

From the Dumont repository, copy these directories into your AEM project:

```
From: dumont/aem/aem-server/core/src/main/java/com/viglet/turing/aem/server/
  ├── config/
  │   └── DumAemIndexerConfig.java
  └── core/
      ├── events/
      │   ├── DumAemPageEventHandler.java
      │   ├── DumAemPageReplicationEventHandler.java
      │   ├── DumAemResourceEventHandler.java
      │   ├── beans/
      │   │   ├── DumAemEvent.java
      │   │   └── DumAemPayload.java
      │   └── utils/
      │       └── DumAemEventUtils.java
      └── services/
          ├── DumAemIndexerService.java
          └── DumAemIndexerServiceImpl.java

To: your-aem-project/core/src/main/java/com/viglet/turing/aem/server/
```

Step 2 — Add Dependencies to Your Core Module

Add these dependencies to your project's `core/pom.xml`:

```
<!-- Lombok (if not already present) -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.34</version>
    <scope>provided</scope>
</dependency>

<!-- Apache Commons Collections -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-collections4</artifactId>
    <version>4.5.0-M3</version>
    <scope>provided</scope>
</dependency>
```

The following dependencies are already provided by AEM SDK and don't need to be added: Jackson, Apache HTTP Client, Sling API, OSGi annotations, AEM Replication API, AEM WCM API.

Step 3 — Add the OSGi Configuration

Create the configuration file for your AEM environment:

```
your-aem-project/ui.config/src/main/content/jcr_root/apps/<your-
project>/osgiconfig/config/
  └──
com.viglet.turing.aem.server.core.services.TurAemIndexerServiceImpl.cfg.json
```

Contents:

```
{
  "enabled": true,
  "host": "$[env:DUMONT_CONNECTOR_HOST;default=http://localhost:30130]",
  "configName": "$[env:DUMONT_CONFIG_NAME;default=WKND]"
}
```

PROPERTY	DESCRIPTION
`enabled`	`true` to activate event-driven indexing, `false` to disable
`host`	URL of the Dumont connector instance (with AEM plugin)
`configName`	AEM source name configured in the Dumont connector

CLOUD MANAGER ENVIRONMENT VARIABLES

Use `$[env:VAR_NAME;default=value]` syntax to configure different values per environment (dev, stage, prod) without changing code. Set the variables in **Cloud Manager → Environments → Environment Variables**.

See Adobe's environment variables documentation.

Step 4 — Configure the Service User

The replication event handler requires a service user mapping. Add this to your project's `ui.config`:

```
your-aem-project/ui.config/src/main/content/jcr_root/apps/<your-
project>/osgiconfig/config/
  └── org.apache.sling.serviceusermapping.impl.ServiceUserMapperImpl.amended-
dumont-indexer.cfg.json
```

```
{
  "user.mapping": [
    "aem-dumont.core:dumont-indexer=[content-reader-service]"
  ]
}
```

Step 5 — Deploy via Cloud Manager

Commit and push your changes to the Git repository

In **Cloud Manager**, run the deployment pipeline for your target environment

After deployment, verify the bundle is active in AEM → **Tools → Operations → Web Console → Bundles** (search for "dumont")

Option 2: Create a New AEM Project from Archetype

If you don't have an existing AEM project, create one from the AEM Project Archetype:

```
mvn -B org.apache.maven.plugins:maven-archetype-plugin:3.2.1:generate \
  -D archetypeGroupId=com.adobe.aem \
  -D archetypeArtifactId=aem-project-archetype \
  -D archetypeVersion=49 \
  -D appTitle="My Site" \
  -D appId="mysite" \
  -D groupId="com.mycompany" \
  -D aemVersion="cloud"
```

Then follow Steps 1–5 from Option 1 above.

For the full archetype reference, see Adobe's archetype documentation.

Option 3: Use the Standalone Dumont AEM Server Module

You can also build and deploy the `aem-server` module directly from the Dumont repository:

```
git clone https://github.com/openviglet/dumont.git
cd dumont/aem/aem-server
mvn clean install -PautoInstallPackage -Daem.host=localhost -Daem.port=4502
```

> **⚠ LOCAL SDK ONLY**
>
> The `-PautoInstallPackage` profile installs directly to a local AEM SDK instance. For AEM as a Cloud Service, you must deploy through a Cloud Manager pipeline (Option 1 or 2).

OSGi Configuration Properties

After deployment, the configuration can be managed in **AEM → Tools → Operations → Web Console → Configuration** (search for "Dumont"):

PROPERTY	DEFAULT	DESCRIPTION
Enabled	`false`	Toggle event-driven indexing on/off
Host	`http://localhost:30130`	Dumont connector URL
Config Name	`WKND`	Source name in the Dumont connector

Cloud Manager Environment Variables

For AEM as a Cloud Service, set these variables in Cloud Manager:

VARIABLE	ENVIRONMENT	VALUE
`DUMONT_CONNECTOR_HOST`	Dev	`http://dev-dumont:30130`
`DUMONT_CONNECTOR_HOST`	Stage	`http://stage-dumont:30130`
`DUMONT_CONNECTOR_HOST`	Prod	`http://prod-dumont:30130`
`DUMONT_CONFIG_NAME`	All	`WKND` (or your source name)

Verifying the Installation

Check bundle status: AEM → Tools → Operations → Web Console → Bundles → search "dumont" — should be **Active**

Check configuration: Web Console → Configuration → search "TurAemIndexer" — verify host and configName

Test: Publish a page in AEM and check the Dumont connector logs for incoming requests

Monitor: In Turing ES → Integration → Monitoring, filter by your AEM source to see indexing events

Related Pages

PAGE	DESCRIPTION
AEM Connector	How the connector processes AEM content (infinity.json, tags, model.json)
Extending the AEM Connector	Custom extensions, configuration JSON, and Maven dependencies
Installation Guide	Deploying Dumont connector with the AEM plugin
Turing ES — AEM Integration	Managing AEM indexing via the Turing ES admin console

WordPress Connector

The WordPress Connector is a **PHP plugin installed directly inside WordPress** — not a Java connector plugin. It hooks into WordPress publish, update, and delete events to keep the search index synchronized with your content in real time.

> **THIS IS A WORDPRESS PLUGIN, NOT A JAVA PLUGIN**
>
> Unlike the AEM and Web Crawler connectors (which are Java JARs loaded via `-Dloader.path`), the WordPress Connector is a **PHP plugin** that you install in your WordPress `wp-content/plugins/` directory. It communicates directly with Turing ES via HTTP — no `dumont-connector.jar` needed.

Unlike the AEM and Web Crawler connectors, the WordPress plugin is **not part of the Dumont connector application** — it runs entirely inside WordPress and sends content directly to Turing ES via HTTP. No `dumont-connector.jar` is needed.

How It Works

The plugin integrates with WordPress hooks to automatically push content to the search index:

- **On publish** — When a post or page is published, the plugin extracts its content and sends it to Turing ES
- **On update** — When published content is modified, the index is updated automatically
- **On delete** — When content is trashed or deleted, it is removed from the index
- **On status change** — When a published post is changed to draft or private, it is removed from the index
- **Manual bulk indexing** — The admin panel provides buttons to index all posts or all pages in batches of 250

There is no scheduled or cron-based indexing — everything is **event-driven** or manually triggered.

Installation

Copy the `viglet-turing-for-wordpress` plugin folder to your WordPress installation:

```
wp-content/plugins/viglet-turing-for-wordpress/
```

Activate the plugin in **WordPress Admin → Plugins**

Configure the connection in **Settings → Viglet Dumont**

Configuration

Server Settings

FIELD	DEFAULT	DESCRIPTION
Host	`localhost`	Turing ES server hostname
Port	`2700`	Turing ES server port
Path	`/dumont`	Base path for the indexing endpoint
Site Name	*(required)*	Semantic Navigation Site name in Turing ES

Indexing Options

FIELD	DESCRIPTION
Post types	Which content types to index (posts, pages, custom post types)
Remove on delete	Automatically remove from index when content is deleted
Remove on status change	Remove from index when status changes from published to draft/private
Index comments	Include post comments in the indexed content
Custom fields	Additional WordPress custom fields to include in the index
Excluded IDs	Post IDs to exclude from indexing

Search Display Options

FIELD	DESCRIPTION
Results per page	Number of search results to display
Show facets	Enable faceted navigation on the search results page
Facet fields	Which fields to show as facets (categories, tags, author, type)
Spellcheck	Enable spelling suggestions for search queries
Max tags	Maximum number of tags to display in facets

Indexed Fields

The plugin extracts and sends these fields to Turing ES:

FIELD	SOURCE
id	WordPress post ID
title	Post/page title
content	Full post content
contentnoshortcodes	Content with WordPress shortcodes removed
permalink	Canonical URL
author	Author display name
type	Content type (post, page, custom)
date	Publication date
modified	Last modification date
categories	Category names (multi-valued)
tags	Tag names (multi-valued)
numcomments	Comment count
comments	Comment text (if comment indexing is enabled)
Custom fields	Any configured custom fields

Multisite Support

The plugin supports WordPress multisite installations. When activated network-wide, it can index content from all blogs in the network, including `blogid`, `blogdomain`, and `blogpath` fields for each document.

Admin Panel Actions

The plugin adds action buttons to the WordPress admin settings page:

ACTION	DESCRIPTION
Ping	Tests the connection to Turing ES
Index all Posts	Bulk indexes all published posts (batches of 250)
Index all Pages	Bulk indexes all published pages (batches of 250)
Delete all	Removes all WordPress content from the Turing ES index
Optimize	Triggers index optimization on the search engine

Architecture

The plugin includes its own PHP HTTP client library (`DumontPhpClient`) for communicating with Turing ES:

```
viglet-turing-for-wordpress/
├── viglet-turing-for-wordpress.php    # Main plugin (hooks, indexing, search)
├── viglet-turing-options-page.php     # Admin settings page
├── DumontPhpClient/
│   └── Viglet/Dumont/
│       ├── Service.php                # HTTP client for Turing ES
│       ├── Document.php               # Document model
│       ├── Response.php               # Response parser
│       └── HttpTransport/
│           ├── Curl.php               # cURL transport
│           ├── CurlNoReuse.php        # cURL without connection reuse
│           └── FileGetContents.php    # file_get_contents fallback
└── template/
    ├── turing4wp_search.php           # Search results template
    ├── search.css                     # Search results styling
    └── autocomplete.css               # Autocomplete styling
```

Developer Guide

Whether you're building custom extensions, contributing to the project, or integrating Dumont DEP into your CI/CD pipeline, this guide has everything you need.

Dumont DEP is fully open-source at github.com/openviglet/dumont. All contributions are welcome.

Tech Stack

LAYER	TECHNOLOGY
Backend	Java 21 · Spring Boot 4.0.3
Message Queue	Apache Artemis (embedded)
Database	H2 (dev) · PostgreSQL (prod)
HTML Parsing	JSoup 1.22.1
Text Extraction	Apache Tika 3.2.3
Search Clients	Turing Java SDK · SolrJ · ES Client
Build	Apache Maven
CI/CD	GitHub Actions

Setting Up Your Dev Environment

Prerequisites

- Java 21 (Temurin recommended)
- Maven 3.9+
- Git
- Turing ES running at `http://localhost:2700` (for end-to-end testing)

Clone, Build, and Run

```
git clone https://github.com/openviglet/dumont.git
cd dumont
mvn clean install

cd connector/connector-app
mvn spring-boot:run
```

The application starts at `http://localhost:30130`.

Project Structure

```
dumont/
├── commons/                    # Shared interfaces, models, utilities
├── aem-commons/                # AEM extension interfaces (published to Maven
Central)
├── spring/                     # Spring Boot configuration and JPA
persistence
├── connector/
│   └── connector-app/          # Main pipeline — strategies, batch, queue,
indexing plugins, API
├── web-crawler/
│   └── wc-plugin/              # Web Crawler connector plugin
├── db/
│   ├── db-commons/             # DB extension interface (published to Maven
Central)
│   ├── db-app/                 # Database connector (standalone CLI)
│   └── db-sample/              # Example custom DB extension
├── filesystem/
│   └── fs-connector/           # FileSystem connector (standalone CLI)
├── aem/
│   ├── aem-plugin/             # AEM connector plugin (loaded via -
Dloader.path)
│   ├── aem-server/             # AEM server-side integration
│   └── aem-plugin-sample/      # Example custom AEM extensions (WKND site)
└── wordpress/                  # WordPress PHP plugin
```

Extension Points

Dumont DEP provides extension points at multiple levels:

Connector-Level Extensions

EXTENSION	GUIDE	MAVEN ARTIFACT
AEM extensions — custom attribute extractors, content processors, delta date logic	Extending the AEM Connector	`com.viglet.dumont:aem-commons:2026.1.19`
Database extensions — custom row transformations during SQL import	Extending the Database Connector	`com.viglet.dumont:db-commons:2026.1.19`

Platform-Level Extensions

Creating a Custom Connector

Implement the `DumConnectorPlugin` interface:

```
public interface DumConnectorPlugin {
    void crawl();
    String getProviderName();
    void indexAll(String source);
    void indexById(String source, List<String> contentId);
}
```

Creating a Custom Indexing Plugin

Implement the `DumIndexingPlugin` interface:

```
public interface DumIndexingPlugin {
    void index(TurSNJobItems turSNJobItems);
    String getProviderName();
}
```

Register your plugin as a Spring `@Component` with `@ConditionalOnProperty(name = "dumont.indexing.provider", havingValue = "your-provider")`.

Processing Strategy Architecture

Strategies are evaluated in priority order for each Job Item:

To add a custom strategy, implement the strategy interface and assign a priority between the existing ones.

REST API

ENDPOINT	DESCRIPTION
`GET /api/v2/connector/status`	Health check
`POST /api/v2/connector/indexing/`	Submit indexing jobs
`GET /api/v2/connector/monitoring/index/{source}`	Monitor indexing progress
`GET /api/v2/connector/validate/{source}`	Validate a content source

For the full API surface, start the application and visit the Swagger UI.

Contributing

Fork the openviglet/dumont repository

Create a branch for your feature or fix: `git checkout -b feature/my-improvement`

Commit with clear, descriptive messages

Open a Pull Request — describe what you changed and why

REST API Reference

Dumont DEP exposes a REST API for triggering indexing operations, monitoring progress, managing sources, and configuring indexing rules. All endpoints use JSON and support CORS.

Connector API

Base path: `/api/v2/connector`

Health Check

```
GET /api/v2/connector/status
```

Returns `{"status": "ok"}` if the service is running.

Index All Content

```
GET /api/v2/connector/index/{name}/all
```

Triggers a full indexing operation for all content in the specified source.

PARAMETER	LOCATION	DESCRIPTION
`name`	Path	Source name

Response: `{"status": "sent"}`

Index Specific Content

```
POST /api/v2/connector/index/{name}
```

Indexes specific documents by their IDs.

PARAMETER	LOCATION	DESCRIPTION
`name`	Path	Source name
(body)	JSON	Array of content IDs: `["id-1", "id-2"]`

Response: `{"status": "sent"}`

Reindex All Content

```
GET /api/v2/connector/reindex/{name}/all
```

Deletes the existing index and reindexes all content from the source.

PARAMETER	LOCATION	DESCRIPTION
`name`	Path	Source name

Response: `{"status": "sent"}`

Reindex Specific Content

```
GET /api/v2/connector/reindex/{name}
```

Deletes and reindexes specific documents.

PARAMETER	LOCATION	DESCRIPTION
`name`	Path	Source name
(body)	JSON	Array of content IDs

Response: `{"status": "sent"}`

Validate Source

```
GET /api/v2/connector/validate/{source}
```

Validates content differences between the source and the search index. Used by the Turing ES Double Check feature.

PARAMETER	LOCATION	DESCRIPTION
`source`	Path	Source name

Response: `DumConnectorValidateDifference` — lists missing and extra documents.

Monitoring API

Base path: `/api/v2/connector/monitoring`

Get Indexing Status

```
GET /api/v2/connector/monitoring/index/{source}
```

Returns indexing status records for a specific source. Returns `404` if no records found.

Get All Monitoring Data

```
GET /api/v2/connector/monitoring/indexing
```

Returns all indexing monitoring data across all sources.

Get Monitoring by Source

```
GET /api/v2/connector/monitoring/indexing/{source}
```

Returns monitoring data filtered by source name.

Search Monitoring Records

```
POST /api/v2/connector/monitoring/indexing/_search
```

Searches indexing monitoring records with pagination and filters.

Request body:

```
{
  "source": "WKND",
  "status": "INDEXED",
  "page": 0,
  "size": 20
}
```

Get Monitoring by Content IDs

```
POST /api/v2/connector/monitoring/indexing
```

Request body: Array of content IDs.

Indexing Statistics

```
GET /api/v2/connector/monitoring/indexing/stats
GET /api/v2/connector/monitoring/indexing/stats/{source}
```

Returns indexing operation statistics (start time, document count, duration, throughput). Returns `404` if no stats found.

System Information API

```
GET /api/v2/connector/system-info
```

Returns runtime system information consumed by the Turing ES Integration → System Information page:

Application name and version

Java version, vendor, and JVM details

Operating system name and version

Physical memory (total, used, free)

JVM heap memory (total, used, free)

Disk space (total, used, free)

Indexing Rules API

Base path: `/api/v2/connector/indexing-rule`

CRUD operations for managing regex-based indexing rules that filter content during extraction.

List All Rules

```
GET /api/v2/connector/indexing-rule
```

List Rules by Source

```
GET /api/v2/connector/indexing-rule/source/{source}
```

Get Rule by ID

```
GET /api/v2/connector/indexing-rule/{id}
```

Create Rule

```
POST /api/v2/connector/indexing-rule
```

Request body:

```
{
  "name": "Exclude error pages",
  "source": "WKND",
  "attribute": "template",
  "ruleType": "IGNORE",
  "values": ["error-page", "redirect"]
}
```

Update Rule

```
PUT /api/v2/connector/indexing-rule/{id}
```

Request body: Same as create.

Delete Rule

```
DELETE /api/v2/connector/indexing-rule/{id}
```

Response: `true` if deleted, `false` if not found.

Get Empty Structure

```
GET /api/v2/connector/indexing-rule/structure
```

Returns an empty model for reference.

AEM Plugin API

Base path: `/api/v2/aem`

These endpoints are available when the AEM plugin JAR is loaded via `-Dloader.path`.

Health Check

```
GET /api/v2/aem/status
```

Returns `{"status": "ok"}`.

Index AEM Paths

```
POST /api/v2/aem/index/{source}
```

Triggers indexing for specific AEM content paths. Includes built-in deduplication — repeated requests for the same paths within 30 seconds are ignored.

PARAMETER	LOCATION	DESCRIPTION
`source`	Path	AEM source name (e.g., `WKND`)
(body)	JSON	`DumAemPathList` object

Request body:

```
{
  "paths": ["/content/wknd/us/en/about"],
  "event": "INDEXING",
  "recursive": false,
  "attribute": "ID"
}
```

FIELD	TYPE	DEFAULT	DESCRIPTION
`paths`	string[]	*(required)*	AEM content paths
`event`	string	`INDEXING`	`INDEXING`, `DEINDEXING`, `PUBLISHING`, or `UNPUBLISHING`
`recursive`	boolean	`false`	Traverse child nodes
`attribute`	string	`ID`	`ID` (path-based) or `URL` (URL-based)

Response: `{"status": "sent"}`

AEM Source Management

CRUD operations for AEM source configurations.

List All Sources

```
GET /api/v2/aem/source
```

Get Source by ID

```
GET /api/v2/aem/source/{id}
```

Create Source

```
POST /api/v2/aem/source
```

Request body: Full `DumAemSource` JSON (see Extending the AEM Connector — Configuration JSON).

Update Source

```
PUT /api/v2/aem/source/{id}
```

Returns `400` if ID mismatch, `404` if not found.

Delete Source

```
DELETE /api/v2/aem/source/{id}
```

Index All Content for Source

```
GET /api/v2/aem/source/{id}/indexAll
```

Triggers an asynchronous full indexing operation for the specified AEM source.

Get Empty Structure

```
GET /api/v2/aem/source/structure
```

Returns an empty AEM source model for reference.

Related Pages

PAGE	DESCRIPTION
AEM Connector	AEM indexing flow, event listeners, and manual triggering
Connectors Overview	All connectors and how they're managed
Turing ES — Integration	Turing ES admin console for managing connectors

Extending the AEM Connector

The AEM connector provides a powerful extension system that lets you customize how content is extracted, transformed, and indexed from Adobe Experience Manager. Extensions are Java classes that implement interfaces from the `aem-commons` library — published on Maven Central.

Maven Dependency

To create custom AEM extensions, add `aem-commons` to your project:

```
<!-- Source: https://mvnrepository.com/artifact/com.viglet.dumont/aem-commons
-->
<dependency>
    <groupId>com.viglet.dumont</groupId>
    <artifactId>aem-commons</artifactId>
    <version>2026.1.19</version>
    <scope>compile</scope>
</dependency>
```

Extension Interfaces

The AEM connector provides four extension interfaces:

INTERFACE	PURPOSE	CONFIG FIELD
`DumAemExtAttributeInterface`	Custom logic for extracting or transforming individual field values	`attributes[].className` or `sourceAttrs[].className`
`DumAemExtContentInterface`	Extract additional content from AEM pages (e.g., `.model.json`)	`models[].className`
`DumAemExtDeltaDateInterface`	Custom delta date resolution for incremental indexing	`sources[].deltaClass`
`DumAemExtUrlAttributeInterface`	Specialized URL handling with ID extraction (extends `DumAemExtAttributeInterface`)	`attributes[].className`

DumAemExtAttributeInterface

The most commonly used extension. Implement this to transform or extract individual attribute values with custom logic.

```
import
com.viglet.dumont.connector.aem.commons.ext.DumAemExtAttributeInterface;
import com.viglet.dumont.connector.aem.commons.DumAemObject;
import com.viglet.dumont.connector.aem.commons.context.DumAemConfiguration;
import com.viglet.dumont.connector.aem.commons.mappers.*;
import com.viglet.turing.client.sn.TurMultiValue;

public class MyCustomAttribute implements DumAemExtAttributeInterface {

    @Override
    public TurMultiValue consume(
            DumAemTargetAttr dumAemTargetAttr,
            DumAemSourceAttr dumAemSourceAttr,
            DumAemObject aemObject,
            DumAemConfiguration dumAemConfiguration
    ) {
        // Example: extract a custom property and transform it
        String rawValue = aemObject.getAttributes()
            .get("myProperty").toString();
        return TurMultiValue.singleItem(rawValue.toUpperCase());
    }
}
```

Parameters:

PARAMETER	DESCRIPTION
`DumAemTargetAttr`	Target search field being populated (name, type)
`DumAemSourceAttr`	Source AEM property definition (name, className)
`DumAemObject`	AEM content node: `path`, `title`, `template`, `jcrNode`, `jcrContentNode`, `lastModified`, `attributes`
`DumAemConfiguration`	Connection config: `url`, `username`, `rootPath`, `authorSNSite`, `publishSNSite`, `providerName`

Built-in implementations:

CLASS	WHAT IT DOES
`DumAemExtContentId`	Returns the AEM page path as the document ID
`DumAemExtContentUrl`	Builds the full URL from the page path and URL prefix config
`DumAemExtContentTags`	Fetches tags from the `/jcr:content.tags.json` endpoint
`DumAemExtCreationDate`	Returns the `jcr:created` date
`DumAemExtModificationDate`	Returns the `cq:lastModified` or `jcr:lastModified` date
`DumAemExtPublicationDate`	Returns the last replication date
`DumAemExtHtml2Text`	Converts HTML content to plain text
`DumAemExtPageComponents`	Extracts text from responsive grid components
`DumAemExtTypeName`	Returns the content type name
`DumAemExtSourceApps`	Returns the provider name from configuration
`DumAemExtSiteName`	Returns the site name

DumAemExtContentInterface

Implement this to fetch additional content from AEM — for example, calling the `.model.json` Sling Model exporter to get structured data.

```
import com.viglet.dumont.connector.aem.commons.ext.DumAemExtContentInterface;
import com.viglet.dumont.connector.aem.commons.bean.DumAemTargetAttrValueMap;

public class MyModelJsonExtractor implements DumAemExtContentInterface {

    @Override
    public DumAemTargetAttrValueMap consume(
            DumAemObject aemObject,
            DumAemConfiguration dumAemConfiguration
    ) {
        DumAemTargetAttrValueMap result = new DumAemTargetAttrValueMap();

        // Fetch the .model.json endpoint
        String url = dumAemConfiguration.getUrl()
            + aemObject.getPath() + ".model.json";
        // ... HTTP call to get JSON ...

        result.addWithSingleValue("fragmentPath",
            "/content/dam/fragment", false);
        return result;
    }
}
```

Referenced in the `models[].className` field of the configuration JSON.

DumAemExtDeltaDateInterface

Customize how the connector determines the "last modified" date for incremental indexing.

```
import
com.viglet.dumont.connector.aem.commons.ext.DumAemExtDeltaDateInterface;

public class MyDeltaDate implements DumAemExtDeltaDateInterface {

    @Override
    public Date consume(
            DumAemObject aemObject,
            DumAemConfiguration dumAemConfiguration
    ) {
        return Optional.ofNullable(aemObject.getLastModified())
                .map(Calendar::getTime)
                .orElse(null);
    }
}
```

Referenced in the `sources[].deltaClass` field of the configuration JSON.

AEM Configuration JSON

The AEM connector is configured via a JSON file that defines sources, attributes, locale mappings, and content models. This file is placed in an `export/` directory and imported at startup.

Full Example (WKND Site)

```
{
  "sources": [
    {
      "name": "WKND",
      "defaultLocale": "en_US",
      "localeClass":
"com.viglet.dumont.connector.aem.commons.ext.DumAemExtLocale",
      "deltaClass": "com.example.MyDeltaDate",
      "endpoint": "http://localhost:4502",
      "username": "admin",
      "password": "admin",
      "oncePattern": "^/content/wknd/us/en/faqs",
      "rootPath": "/content/wknd",
      "contentType": "cq:Page",
      "author": true,
      "publish": true,
      "authorSNSite": "wknd-author",
      "publishSNSite": "wknd-publish",
      "authorURLPrefix": "http://localhost:4502",
      "publishURLPrefix": "https://wknd.site",
      "localePaths": [
        { "locale": "en_US", "path": "/content/wknd/us/en" },
        { "locale": "es", "path": "/content/wknd/es/es" }
      ],
      "attributes": [
        {
          "name": "id", "type": "STRING", "mandatory": true,
          "className":
"com.viglet.dumont.connector.aem.commons.ext.DumAemExtContentId"
        },
        {
          "name": "title", "type": "TEXT", "mandatory": true,
          "facetName": { "default": "Titles", "pt_BR": "Títulos" }
        },
        {
          "name": "tags", "type": "STRING", "multiValued": true,
          "facet": true, "facetName": { "default": "Tags" },
          "className":
"com.viglet.dumont.connector.aem.commons.ext.DumAemExtContentTags"
        },
        {
          "name": "url", "type": "STRING", "mandatory": true,
          "className":
"com.viglet.dumont.connector.aem.commons.ext.DumAemExtContentUrl"
        }
      ],
```

```
        "models": [
          {
            "type": "cq:Page",
            "className": "com.example.MyModelJsonExtractor",
            "targetAttrs": [
              { "name": "title", "sourceAttrs": [{ "name": "jcr:title" }] },
              { "name": "tags",  "sourceAttrs": [{ "name": "cq:tags" }] },
              {
                "name": "text",
                "sourceAttrs": [{
                  "className":
"com.viglet.dumont.connector.aem.commons.ext.DumAemExtPageComponents"
                }]
              }
            ]
          }
        ]
      }
    ]
}
```

Source Fields

FIELD	TYPE	DESCRIPTION
`name`	string	Source identifier (displayed in the admin console)
`endpoint`	string	AEM instance URL (e.g., `http://localhost:4502`)
`username` / `password`	string	AEM authentication credentials
`rootPath`	string	Content tree root to crawl (e.g., `/content/wknd`)
`contentType`	string	JCR node type to index (e.g., `cq:Page`)
`defaultLocale`	string	Fallback locale code (e.g., `en_US`)
`localeClass`	string	Class for locale resolution
`deltaClass`	string	Class implementing `DumAemExtDeltaDateInterface`
`oncePattern`	string	Regex — matching paths are indexed only once (never re-indexed)
`author` / `publish`	boolean	Enable indexing from author/publish environments
`authorSNSite` / `publishSNSite`	string	Turing ES SN Site names for each environment
`authorURLPrefix` / `publishURLPrefix`	string	Public URL prefixes for documents

Locale Paths

```
"localePaths": [
  { "locale": "en_US", "path": "/content/wknd/us/en" },
  { "locale": "es",    "path": "/content/wknd/es/es" }
]
```

Content found under each path is tagged with the corresponding locale.

Attribute Fields

FIELD	TYPE	DESCRIPTION
`name`	string	Field name in the search index
`type`	string	`STRING`, `TEXT`, or `DATE`
`mandatory`	boolean	Whether this field is required
`multiValued`	boolean	Whether this field holds multiple values
`description`	string	Human-readable description
`facet`	boolean	Expose as a facet filter in search results
`facetName`	object	Localized labels: `{ "default": "Tags", "pt_BR": "Etiquetas" }`
`className`	string	Class implementing `DumAemExtAttributeInterface` — extracts the value instead of reading from JCR

Model Fields

FIELD	TYPE	DESCRIPTION
`type`	string	JCR node type this model applies to
`className`	string	Class implementing `DumAemExtContentInterface`
`targetAttrs[].name`	string	Target field (must match `attributes[]`)
`targetAttrs[].sourceAttrs[].name`	string	JCR property to read (e.g., `jcr:title`)
`targetAttrs[].sourceAttrs[].className`	string	Class implementing `DumAemExtAttributeInterface` for custom extraction

Creating a Custom AEM Extension

Step 1 — Create a Maven project

```
<project>
    <groupId>com.example</groupId>
    <artifactId>my-aem-extensions</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <dependency>
            <groupId>com.viglet.dumont</groupId>
            <artifactId>aem-commons</artifactId>
            <version>2026.1.19</version>
        </dependency>
    </dependencies>
</project>
```

Step 2 — Implement your extension

```
package com.example.ext;

import
com.viglet.dumont.connector.aem.commons.ext.DumAemExtAttributeInterface;
import com.viglet.turing.client.sn.TurMultiValue;

public class MyBreadcrumb implements DumAemExtAttributeInterface {
    @Override
    public TurMultiValue consume(DumAemTargetAttr target,
            DumAemSourceAttr source, DumAemObject aemObject,
            DumAemConfiguration config) {
        String path = aemObject.getPath()
            .replace(config.getRootPath(), "");
        return TurMultiValue.singleItem(
            String.join(" > ", path.split("/")));
    }
}
```

Step 3 — Build and deploy

```
mvn clean package
cp target/my-aem-extensions-1.0.0.jar /appl/viglet/dumont/aem/libs/
```

The `libs/` directory must contain both `aem-plugin.jar` and your extension JAR.

Step 4 — Reference in the JSON

```
{
  "name": "breadcrumb",
  "type": "STRING",
  "className": "com.example.ext.MyBreadcrumb"
}
```

How classes are loaded

Extension classes are loaded via `DumCustomClassCache` using `Class.forName()`. Requirements:

Public no-argument constructor

On the classpath (via `libs/` and `-Dloader.path`)

Thread-safe (one instance is shared across all calls)

Related Pages

PAGE	DESCRIPTION
AEM Connector	AEM connector features, configuration, and locale mapping
Installation Guide	How to deploy plugins with `-Dloader.path`
Developer Guide	Project structure, build, and contribution guide

Extending the Database Connector

The Database Connector supports a custom extension point that lets you transform, enrich, or filter database rows before they are indexed. Extensions are Java classes that implement the `DumDbExtCustomImpl` interface from the `db-commons` library.

Maven Dependency

```
<!-- Source: https://mvnrepository.com/artifact/com.viglet.dumont/db-commons -
->
<dependency>
    <groupId>com.viglet.dumont</groupId>
    <artifactId>db-commons</artifactId>
    <version>2026.1.19</version>
    <scope>compile</scope>
</dependency>
```

The DumDbExtCustomImpl Interface

```
public interface DumDbExtCustomImpl {
    Map<String, Object> run(Connection connection, Map<String, Object>
attributes);
}
```

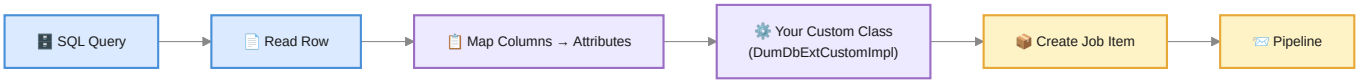
PARAMETER	DESCRIPTION
`Connection connection`	The active JDBC connection — you can execute additional queries to enrich the row
`Map<String, Object> attributes`	Mutable map of column values from the current row (column aliases become keys)

Return: The modified attributes map. The returned map is what gets indexed.

How It Works

For each row in the SQL result set, the connector:

- Reads all columns into an attributes map (`Map<String, Object>`)
- Applies value formatting (HTML tag removal, date formatting)
- **Calls your custom class** via `run(connection, attributes)`
- Processes multi-valued fields
- Creates a Job Item and sends it to the pipeline



Example: Prefixing Titles

A minimal example that prefixes every title with a label:

```
package com.example.ext;

import com.viglet.dumont.connector.db.ext.DumDbExtCustomImpl;
import java.sql.Connection;
import java.util.Map;

public class MyDbCustom implements DumDbExtCustomImpl {

    private static final String TITLE = "title";

    @Override
    public Map<String, Object> run(Connection connection,
                                   Map<String, Object> attributes) {
        if (attributes.containsKey(TITLE)) {
            attributes.replace(TITLE,
                String.format("Product: %s", attributes.get(TITLE)));
        }
        return attributes;
    }
}
```

Example: Enriching with a Second Query

Use the JDBC connection to run additional queries and add data:

```
public class EnrichWithCategory implements DumDbExtCustomImpl {

    @Override
    public Map<String, Object> run(Connection connection,
                                   Map<String, Object> attributes) {
        Object categoryId = attributes.get("category_id");
        if (categoryId != null) {
            try (var stmt = connection.prepareStatement(
                    "SELECT name FROM categories WHERE id = ?")) {
                stmt.setObject(1, categoryId);
                try (var rs = stmt.executeQuery()) {
                    if (rs.next()) {
                        attributes.put("category", rs.getString("name"));
                    }
                }
            } catch (Exception e) {
                // log and continue
            }
        }
        return attributes;
    }
}
```

Using the Extension

Via CLI

Pass the fully-qualified class name with the `--class-name` parameter:

```
java -cp dumont-db-indexer.jar:my-db-extensions.jar \
  com.viglet.dumont.connector.db.DumDbImportTool \
  --class-name com.example.ext.MyDbCustom \
  --server http://localhost:30130 \
  --api-key <API_KEY> \
  --driver org.mariadb.jdbc.Driver \
  --connect "jdbc:mariadb://localhost:3306/shop" \
  --query "SELECT id, name AS title, price FROM products" \
  --site ProductCatalog \
  --locale en_US
```

CLASSPATH

Your extension JAR must be on the classpath (`-cp`). Add it alongside the `dumont-db-indexer.jar` separated by `:` (Linux) or `;` (Windows).

Creating a Custom DB Extension

Step 1 — Create a Maven project

```
<project>
    <groupId>com.example</groupId>
    <artifactId>my-db-extensions</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <dependency>
            <groupId>com.viglet.dumont</groupId>
            <artifactId>db-commons</artifactId>
            <version>2026.1.19</version>
        </dependency>
    </dependencies>
</project>
```

Step 2 — Implement the interface

```
package com.example.ext;

import com.viglet.dumont.connector.db.ext.DumDbExtCustomImpl;
import java.sql.Connection;
import java.util.Map;

public class MyDbCustom implements DumDbExtCustomImpl {
    @Override
    public Map<String, Object> run(Connection connection,
                                   Map<String, Object> attributes) {
        // Transform, enrich, or filter attributes
        return attributes;
    }
}
```

Step 3 — Build

```
mvn clean package
```

Step 4 — Run with the extension

```
java -cp dumont-db-indexer.jar:target/my-db-extensions-1.0.0.jar \
  com.viglet.dumont.connector.db.DumDbImportTool \
  --class-name com.example.ext.MyDbCustom \
  --server http://localhost:30130 \
  --api-key <API_KEY> \
  ...
```

Requirements

Public no-argument constructor

On the classpath (via `-cp`)

Thread-safe — one instance processes all rows sequentially, but should avoid mutable shared state

Related Pages

PAGE	DESCRIPTION
Database Connector	CLI parameters, SQL examples, supported databases
Extending the AEM Connector	AEM extension interfaces and configuration JSON
Developer Guide	Project structure, build, and contribution guide