CONTENTSTACK

New CMS Connector API Guide

This document serves as a guide for extending the CMS Connector API to support new Content Management Systems (CMS) and file types. It outlines the process of creating new CMS packages, updating validator and mapper logic, and handling various file extensions and CMS configurations in the `/validator` route and `handleFileProcessing` function. The goal is to maintain a modular and CMS-agnostic architecture, allowing for seamless integration of different CMS platforms by following a structured approach for validation, processing, and data mapping.

🆕 Creating a Similar CMS Package

Like migration-contentful, migration-sitecore, migration-wordpress

To support a new CMS (e.g., Drupal), create a new package like this:

```
TypeScript
📁 migration-drupal/
```

Inside it, implement the required structure:

```
TypeScript
// migration-drupal/index.js


const validator = (data) => {
  // Validate file format, structure, required fields
};


const ExtractFiles = async (filePath) => {
  // Unzip or parse content
};
```

If you have any questions, please reach out to tso-migration@contentstack.com.

CONTENTSTACK

```
const contentTypes = {
  // Drupal-specific content type definitions
};

const reference = {
  // Drupal-specific field references
};

const extractLocales = (data) => {
  // Return locale array
};

module.exports = {
  validator,
  ExtractFiles,
  contentTypes,
  reference,
  extractLocales
};
```

🧩 Where It's Plugged In

Once your package (e.g., migration-drupal) is ready:

1. Add it to package.json:

```JSON
      "migration-drupal": "file:migration-drupal"
```

2. Update your validator logic:

```TypeScript
const { validator: drupalValidator } = require('migration-drupal');


switch (CMSIdentifier) {
  case 'drupal-zip':
    return drupalValidator({ data });
}
```

3. Update createMapper logic and Add the case in createMapper switch::

```TypeScript
case 'drupal':
  return await createDrupalMapper(...);
```

✅ Summary

To add a new CMS, simply:

- Scaffold a package like migration-drupal
- Follow the existing method structure
- Add it to validator and mapper switch blocks

This keeps the architecture modular and CMS-agnostic, all the explanations provided in subTabs.

If you have any questions, please reach out to tso-migration@contentstack.com.

CONTENTSTACK

📄 /validator Route

This route is used to validate and process a file (local or from AWS S3), based on file type and CMS configuration.

✅ Functionality Overview

- Accepts a GET request to /validator
- Determines if the file source is local or from S3
- Based on the file extension:
    - If XML → reads as string
    - Else (e.g., ZIP) → reads as binary buffer
- Calls handleFileProcessing with file data
- On success, maps the processed file using createMapper

🔍 Where to Add Support for New File Types

In this route, file extensions are determined using:

```
JavaScript
const fileExt = fileName?.split?.('.')?.pop() ?? '';
```

Then based on file extension, logic is handled like this:

```
JavaScript
if (fileExt === 'xml') {
  // Process XML as string
} else {
  // Process as buffer (e.g., zip)
}
```

👉 If you want to add a new file type, you can add it inside this condition:

If you have any questions, please reach out to tso-migration@contentstack.com.

CONTENTSTACK

```javascript
if (fileExt === 'xml') {
  // ...
} else if (fileExt === 'yourNewExtension') {
  // Add logic to read and handle this new file type
} else {
  // Default processing for binary files
}
```

You may also need to extend handleFileProcessing to handle the new file type.

🔄 Where to Add Support for New CMS Types

The cmsType is retrieved from:

```
const cmsType = config?.cmsType?.toLowerCase();
```

Later, this variable is passed to:

```
const data = await handleFileProcessing(fileExt, fileData, cmsType, name);
```

👉 If you want to add a new CMS, you can extend logic inside handleFileProcessing or wherever you handle CMS-specific processing.

Also update:

```
createMapper(filePath, projectId, app_token, affix, config);
```

To support your new CMS type, add logic to createMapper to handle it accordingly.

✅ Summary

- You can add new file extensions by modifying the fileExt condition

- You can add new CMS types by extending handleFileProcessing and createMapper
- The system already separates XML (string) and ZIP (buffer) handling — follow that structure

📄 HandleFileProcessing

This document explains how to integrate a new CMS platform into the existing file validation and processing flow using the functions:

- handleFileProcessing()
- validator()

🧩 Purpose

The backend currently supports multiple CMS types (like Sitecore, Contentful, WordPress, AEM). The goal is to validate uploaded files and transform them into a structured format specific to each CMS.

The system identifies a CMS + file type pair using this format:

📌 type-extension → e.g., sitecore-zip, contentful-json, wordpress-xml

---

🛠️ Where to Add a New CMS

Step 1: Update the validator() Function

Located inside:

```
JavaScript
const validator = ({ data, type, extension }: { data: any; type: string;
extension: string }) => { ... }
```

Find the switch statement on CMSIdentifier. Each case handles one CMS-type and file-extension combination.

👉 To add a new CMS, add a new case in this switch block:

Example:

```javascript
case 'newcms-json': {
  return newCMSValidator(data); // Your new validation logic
}
```

Make sure:

- CMSIdentifier format matches the expected {type}-{extension}
- The validation function (e.g., newCMSValidator) is imported or defined

🧠 You can add multiple variations, like:

```
case 'newcms-zip':
case 'newcms-xml':
```

Step 2: Create the Validator Function

In your project, define the validator logic for the new CMS. Example:

```javascript
const newCMSValidator = (data) => {
  // Your custom validation logic for JSON, ZIP, etc.
  return true; // or false if validation fails
};
```

📥 Input: This could be:

- A JSZip object (for zip)
- Raw XML string
- JSON object/string

📤 Output: Boolean (true = valid, false = rejected)

If you have any questions, please reach out to tso-migration@contentstack.com.

CONTENTSTACK

Step 3: Test with handleFileProcessing()

The validator() function is used inside handleFileProcessing():

```
JavaScript

if (await validator({ data: zipBuffer, type: cmsType, extension: fileExt })) {
  // ...
}
```

Make sure your new CMS validator is covered in the condition by passing the correct:

- cmsType (from config.cmsType)
- fileExt (from the uploaded file)

✅ Summary Checklist

Task	Description
Add new case	Update validator() switch block
Implement logic	Write your own newCMSValidator function
Return true/false	Ensure validator returns a boolean
Use correct key	Follow type-extension format (e.g., newcms-zip)

🧪 Optional: Debugging Tips

If you have any questions, please reach out to tso-migration@contentstack.com.

- Log the CMSIdentifier inside validator() to ensure your case is reached.
- Ensure handleFileProcessing is passing correct fileExt and cmsType.

🗺️ Adding Mapper Support for a New CMS

After validating and processing a file, the backend prepares mapping data using the createMapper function. This step transforms the extracted content into a standardized format that can be used to generate dummy data and locale mappings in your CMS.

🔄 How Mapping Works (High-Level Flow)

1. ✅ File is successfully validated and saved (ZIP, XML, JSON, etc.)
2. 📍 Path to the processed file is determined:

```
JavaScript
const filePath = path.join(__dirname, '..', '..', 'extracted_files', fileName);
```

3. 🧠 The createMapper function is invoked:

```
JavaScript
createMapper(filePath, projectId, app_token, affix, config);
```

4. 🔁 This function routes logic based on the CMS type (e.g., sitecore, contentful, wordpress)

🔧 createMapper Function – Structure

```
JavaScript
const createMapper = async (

  filePath,

  projectId,

  app_token,
```

```
  affix,
  config
) => {
  const CMSIdentifier = config?.cmsType?.toLowerCase();
  switch (CMSIdentifier) {
    case 'sitecore':
      return await createSitecoreMapper(filePath, projectId, app_token, affix,
config);
    case 'contentful':
      return await createContentfulMapper(projectId, app_token, affix, config);
    case 'wordpress':
      return createWordpressMapper(filePath, projectId, app_token, affix);
    default:
      return false;
  }
};
```

📌 Each case corresponds to a CMS and calls its mapper function.

🆕 How to Add a New CMS

Step 1: Add Case in createMapper

Add a new case for your CMS identifier:

JavaScript

```
case 'newcms': {
  return await createNewCMSMapper(filePath, projectId, app_token, affix,
config);
```

```
}
```

Step 2: Create the Mapper Function

Implement a new function like:

```JavaScript
const createNewCMSMapper = async (filePath, projectId, app_token, affix, config) => {
  // 1. Read and transform file contents
  // 2. Generate field mapping object
  // 3. Send to /v2/mapper/createDummyData
  // 4. Generate locale mapping and call /v2/migration/localeMapper
};
```

Step 3: Make Dummy Data API Call

Use axios like the existing implementation:

```JavaScript
const config = {
  method: 'post',
  maxBodyLength: Infinity,
  url: `${process.env.NODE_BACKEND_API}/v2/mapper/createDummyData/${projectId}`,
  headers: {
    app_token,
    'Content-Type': 'application/json'
  },
  data: JSON.stringify(fieldMapping),
```

If you have any questions, please reach out to tso-migration@contentstack.com.

```
};


const { data } = await axios.request(config);


if (data?.data?.content_mapper?.length) {
  deleteFolderSync(infoMap?.path);
  logger.info('Validation success:', {
    status: HTTP_CODES?.OK,
    message: HTTP_TEXTS?.MAPPER_SAVED,
  });
}
```

Step 4: Handle Locale Mapping

If your CMS supports localization, add this or add en-us as default:

```
JavaScript
const mapperConfig = {
  method: 'post',
  maxBodyLength: Infinity,
  url:
`${process.env.NODE_BACKEND_API}/v2/migration/localeMapper/${projectId}`,
  headers: {
    app_token,
    'Content-Type': 'application/json'
  },
  data: {
```

If you have any questions, please reach out to tso-migration@contentstack.com.

```
    locale: Array.from(localeData) ?? []
  }
};

await axios.request(mapperConfig);
```

Running the `upload-api` Project on Any Operating System

The following instructions will guide you in running the `upload-api` folder on any operating system, including Windows and macOS.

Starting the `upload-api` Project

There are two methods to start the `upload-api` project:

Method 1:
Run the following command from the root directory of your project:

```Shell
npm run upload
```

This command will directly start the `upload-api` package.

Method 2:
Navigate to the `upload-api` directory manually and run the development server:

```Shell
cd upload-api

npm run start
```

This approach starts the `upload-api` from within its own directory.

Restarting After Termination

If the project terminates unexpectedly, you can restart it by following the same steps outlined above. Choose either Method 1 or Method 2 to relaunch the service.