CMS Connector API - Wordpress

This document details the functionality of the **`migration-wordpress`** Node.js utility package, which facilitates the migration of WordPress content to Contentstack. It outlines the package's modules, including extractContentTypes, contentTypeMaker, and extractLocales, and explains how these functions process and transform WordPress data into Contentstack-compatible formats. Additionally, the document covers the validator module used to ensure proper XML and JSON structures for various CMS exports, and it highlights limitations of the WordPress migration process.

📦 migration-wordpress

A Node.js utility package designed to support the migration of Wordpress content. It provides functionality to extract content types, configuration, locale files, and reference mappings from a Wordpress structure.

📁 Modules

1. `extractContentTypes(affix)`

Description:

Generates a list of content type schemas by mapping static schema definitions and writing them to a `schema.json`

Parameters:

- `affix` (`string`) – A string used to determine whether a restricted UID prefix should be applied globally.

Returns:

A `Promise<void>`. Resolves when the schema file is successfully created. Logs success or error messages to the console.

---

2. `contentTypeMaker(affix)`

Description:
Generates a Contentstack-compatible schema mapping from content type definitions and writes them to a `schema_mapper.json` file.

🗂️ Project Structure

```
JavaScript
migration-sitecore/
├── libs/
│   ├── content_types.js
│   ├── contenttypemapper.js
│   └── extractLocales.js
├── index.js
```

---

3. `extractLocales(path)`

Description:
Extracts locale-specific content from the Wordpress JSON export.

Parameters:

- `directoryPath` *(string)* – Path to the Wordpress JSON export.

Returns:

An `Array[string]` with the unique locale-codes used.

📄 `extractContentTypes` Function

Overview

This module defines a set of Content Types (such as Authors, Categories, Tags, Terms, and Posts) in a format compatible with **Contentstack**. It includes logic to generate the schema for each content type and ensures valid UID generation based on a global prefix and restricted UID list.

CONTENTSTACK

🔧 Function: `extractContentTypes(filePath, prefix)`

Description:

Generates a JSON schema file for content types by transforming content type definitions using `generateSchema`, and writes the result to the configured path. It optionally applies a UID prefix if the provided `affix` matches any value in a restricted list.

Parameters:

- `affix` (`string`): A UID string used to determine if a prefix (`globalPrefix`) should be applied to the content type processing logic.

Returns:

- `Promise<void>`: Resolves when content type schemas are successfully generated and written to disk.

Behavior:

1. Calls `startingDir()` to ensure the environment is ready for processing.
2. Checks if the `affix` exists in the `restrictedUid` list and sets `globalPrefix` accordingly.
3. Maps each entry in `ContentTypesSchema` through `generateSchema()` to transform the raw content type data.
4. Writes the generated array of schema objects to `content_types/schema.json` using `helper.writeFileAsync`.
5. Logs a success message on completion.
6. Catches and logs any errors that occur during the process.

Throws:

- Catches and logs any errors that occur during the process.

🛠️ Function: `startingDir()`

Description:

Ensures that the content type directory and schema file are created before any schema writing operations.

Parameters:

- None

Returns:

- `void`: This function writes files and does not return anything.

Behavior:

1. Checks if the directory at `contentTypeFolderPath` exists.
2. If it doesn't exist, it creates the directory using `mkdirp.sync`
3. Initializes an empty schema file (`schema.json`) at the specified path using `helper.writeFile`.

🛠️ Function: `generateSchema(title, uid, fields, options)`

Description:

Generates a content type schema object with a standardized structure, including computed title, UID, description, and optional settings.

Parameters:

- `title` (string): The raw title of the content type.
- `uid` (string): The original UID of the content type.
- `fields` (Array): The schema fields to be included in the content type.
- `options` (Object): Optional configuration for the content type.

Returns:

- `Object`: A structured content type schema object containing `title`, `uid`, `schema`, `description`, and `options`.

Behavior:

1. Computes the title using `generateTitle(title)`.
2. Computes the UID using `generateUid(uid)`.
3. Creates a description string based on the computed title.
4. Returns a new object representing the full content type schema.

Output File:

Each content type gets its own `.json` file named after its cleaned-up title (special characters removed, first letter capitalized).

Output Format (per field):

Each saved JSON array contains objects with the following keys:

- `title`: Cleaned and formatted content type title.
- `uid`: Transformed UID used for the content type.

- `schema`: Original field definitions array for the content type.
- `description`: A string describing the content type schema.
- `options`: Additional metadata and settings associated with the content type.

📄 `contentTypeMaker` Function

Overview

This `contentTypeMaker` function reads content type schemas from a configured file (`schema.json`), processes each content type and its fields, and generates a `schema_mapper.json` file containing a transformed schema structure compatible with Contentstack. It ensures necessary directories exist and applies UID corrections and field type conversions.

📦 Function Signature

```
JavaScript
const contentTypeMaker = async (affix) => { ... }
```

🔧 Inputs

- **affix** (string) A string prefix to append to UIDs if they are restricted.

Reads files from:

- Reads `schema.json` (from configured path) using `readFileSync`.

📤 Output

An object in the following structure:

```javascript
{
  "status": 1,
  "otherCmsTitle": "Article",
  "otherCmsUid": "article",
  "isUpdated": false,
  "updateAt": "",
  "contentstackTitle": "Article",
  "contentstackUid": "article",
  "fieldMapping": [
    {
      "id": "title",
      "uid": "title",
      "otherCmsField": "Title",
      "contentstackFieldType": "single_line_text",
      "backupFieldType": "single_line_text",
      "backupFieldUid": "title"
    }
  ],
  "type": "content_type"
}
```

Each `contentType` object contains:

Field	Type	Description
`status`	`number`	Indicates active state (1 = active)
`isUpdated`	`boolean`	Whether the mapping was updated
`updateAt`	`string`	Timestamp (left blank initially)
`otherCmsTitle`	`string`	Content type title from Wordpress
`otherCmsUid`	`string`	UID from Wordpress
`contentstackTitle`	`string`	Transformed title for Contentstack (Capitalized)
`contentstackUid`	`string`	UID for Contentstack, corrected with `uidCorrector()`
`type`	`string`	Always `'content_type'`
`fieldMapping`	`array`	An array of field definitions, including system fields

---

⚙️ Internal Logic

1. Ensure Directory Setup

```
JavaScript
startingDir();
```

- Initializes the necessary folder structure if it doesn't already exist.
- Ensures `schema_mapper.json` has a valid base before proceeding.

---

2. Read File Content

```
JavaScript
const fileContent = readFileSync(contentTypesFile);

const contentTypes = JSON.parse(fileContent);
```

- Reads the source JSON file (`schema.json`) containing content type definitions.
- Parses it into a JavaScript array of content type objects from another CMS (e.g., Sitecore, Drupal, etc.).

---

3. Map Content Types

```
JavaScript
const mainSchema = contentTypes.map((element) => ({ ... }));
```

For each content type object:

- `Copies over the original title and UID.`
- `Sets migration metadata (status, isUpdated, etc.).`
- `Constructs fieldMapping by transforming each field using ContentTypeSchema.`

---

4. Transform Fields

```
JavaScript
fieldMapping: element.schema.map(({ display_name, uid, data_type,
field_metadata, reference_to }) => ContentTypeSchema({...}))
```

For each field:

- Corrects UID if needed using `uidCorrector`.
- Extracts properties such as `default_value`, `multiline`, and `reference_to`.
- Maps each field to a Contentstack compatible format via `ContentTypeSchema`.

---

5. Write transformed schema to disk

```
JavaScript
await writeFileAsync(
  path.join(process.cwd(), config.data, contentTypesConfig.dirName,
'schema_mapper.json'),
  mainSchema,
  4
);
```

Writes the complete transformed array (`mainSchema`) to `schema_mapper.json` under the configured `data` directory.

Uses pretty formatting with 4-space indentation.

---

6. Return Result

```
JavaScript
return mainSchema;
```

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

Example Return

JavaScript

```
{
  "contentTypes": [
    {
      "status": 1,
      "isUpdated": false,
      "updateAt": "",
      "otherCmsTitle": "blogPost",
      "otherCmsUid": "blogPostID",
      "contentstackTitle": "BlogPost",
      "contentstackUid": "blog_post",
      "type": "content_type",
      "fieldMapping": [
        {
          "uid": "title",
          "contentstackFieldType": "text",
          ...
        },
        ...
      ]
    }
  ]
}
```

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

CONTENTSTACK

🔗 Related Functions

- `contentTypeSchema`: Create Schema with contentstack format.
- `uidCorrector`: Normalizes field UIDs.
- startingDir: Checks the starting directory as schema_mapper.json.

📄 `extractLocale` Function

📝 Description

The extractLocale function processes a WordPress-exported XML file parsed into JSON to identify and return all unique language codes defined globally or within individual post metadata.

📦 Dependencies

- **`path`**: File path of the wordpress JSON export.
- **`fs`** (Node.js built-in): Used to read the file contents.

🔧 Function: `extractLocale`

Purpose

To extract and deduplicate language or locale identifiers from WordPress export data (typically in XML-to-JSON converted format), assisting in locale mapping in content migrations.

Parameters

- `jsonFilePath` (`string`) – A valid file path where the exported data is parsed onto JSON format.

Returns

CONTENTSTACK

`Array<string>` – A list of unique locale codes.

Example:

```
JavaScript
['en-us', 'fr-fr']
```

- Returns an empty array `[]` if:
    - No locales are found.
    - The file is missing or invalid.

Behavior

- Validates the file path and JSON structure.
- Validates if the data is in correct JSON format
- Collects unique locale codes (`locale.code`).
- Handles and logs errors gracefully.

📌 Example Usage

```
JavaScript
const extractLocale = require('./libs/extractLocale');

const pathToFile = './wordpress-export.json';

try {

  const locales = extractLocale(pathToFile);

  console.log('Locales found:', locales);
```

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

CONTENTSTACK

```
} catch (err) {

console.error(err.message);

}();
```

⚙️ Validator Wordpress

🧾 Overview

A **WordPress XML Validator** that checks whether all required tags defined in a configuration file (`wordpress.json`) are present in a given WordPress XML export. It is used when validating content structure before processing or importing.

🔧 Function: `wordpressValidator(data: string)`

Description:

Validates a WordPress XML string by comparing it against a predefined schema defined in `wordpress.json`.

Parameters:

- `data` (`string`): The raw XML string exported from WordPress

Returns:

✅ **`true`** if:

- All required tags defined in the config are present in the XML.

❌ **`false`** if:

- Any required tag is missing.
- The input XML is malformed (i.e., cannot be parsed).

🔧 Internal Logic

CONTENTSTACK

1. Load XML using Cheerio

```javascript
const $ = Cheerio.load(data, { xmlMode: true });
```

1. Uses Cheerio in XML mode to parse the raw XML string for DOM-like querying.

2. Iterate over config properties

```javascript
Object.entries(Config).every(([item, itemValue]) => { ... })
```

1. Loops through each key defined in wordpress.json
2. Extracts the expected tag name and required flag

3. Iterate over config properties

```javascript
const xmlItem = $(`${val?.name}`);

if (xmlItem?.length === 0 && val?.required === 'true') {

  return false;

}
```

1. Checks whether the tag exists in the loaded XML.
2. If the tag is **required** and **missing**, validation fails.

CONTENTSTACK

✅ Example Usage

```javascript
import wordpressValidator from './validators/wordpress-validator';


const xmlData = fs.readFileSync('export.xml', 'utf-8');


if (wordpressValidator(xmlData)) {
  console.log('✅ XML is valid!');
} else {
  console.error('❌ XML is invalid or missing required fields.');
};
```

📦 Validator

A lightweight utility module used to **detect and validate** CMS content exports (WordPress, Contentful, Sitecore, AEM) based on the combination of content type and file extension. It dynamically delegates to the appropriate validator module to ensure required structure, fields, and folders exist before processing begins.

📁 Modules

1. `wordpressValidator(data: string)`

Description:

Validates a WordPress XML string by comparing it against a predefined schema defined in `wordpress.json`.

Parameters:

- `data` (`string`): The raw XML string exported from WordPress

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

Returns:

A `Boolean`. Based on the validation if passed - true, else false.

---

2. `sitecoreValidator({ data }: props)`

Description:

Validates that all required Sitecore folders exist within the `files` object provided, including templates, content, blobs, and media library files.

Parameters:

- `data` (`object`): A `props` object containing a `files` map representing file paths extracted from Sitecore export

Returns:

A `Boolean`. Based on the validation if passed - true, else false

---

3. `contentfulValidator(data: string)`

Description:

Validates a raw JSON string exported from Contentful against a required schema definition provided in `contentful.json`.

Parameters:

- `data` (`string`): Raw JSON content model as a string (from Contentful)

Returns:

A `Boolean`. Based on the validation if passed - true, else false

📁 Project Structure

CONTENTSTACK

```
validator/
├── sitecore/
│   └── sitecore.js        🔁 Validates Sitecore ZIP structure
├── wordpress/
│   └── wordpress.js       🔁 Validates WordPress XML tags
├── contentful/
│   └── contentful.js      🔁 Validates Contentful JSON schema
├── aem/
│   └── aem.js             🔁 Validates AEM ZIP structure
└── index.ts               ✅ Main entry point to route validation by CMS type
```

🔧 Internal Logic

1. Build CMS Identifier

```javascript
const CMSIdentifier = `${type}-${extension}`;
```

1. Constructs a unique key to match the CMS and file type (e.g., `wordpress-xml`).

2. Validator Routing

```javascript
switch (CMSIdentifier) {

  case 'sitecore-zip': return sitecoreValidator({ data });

  case 'contentful-json': return contentfulValidator(data);

  case 'wordpress-xml': return wordpressValidator(data);
```

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

```
  case 'aem-zip': return aemValidator({ data });
  default: return false;
}
```

1. Routes to the appropriate validator module.
2. If no match is found, returns `false` by default.

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.

Limitations Of Wordpress

1. Not supporting Wordpress dynamic content models
2. Supports basic data migration process
3. Not handle the use case of deletion of existing destination stack in runtime
4. **Content Type Migration Limitations in Test Stacks**

   When migrating content types in a test stack, the handling of attached references depends on your organization's reference limit:

   - **Organizations with a reference limit of 50:** Full data migration is supported if a content type has more than 10 references.
   - **Organizations with a reference limit of 10:** If a content type has more than 10 references, only the 'title' and 'URL' fields will be migrated.