CMS Connector API - Sitecore

The document describes **`migration-sitecore`**, a Node.js utility package designed to facilitate the migration of Sitecore content to another system. It details various modules within the package, including those for extracting locales, configuration, content types, and reference mappings. The document also outlines the project structure, explains the functionality of key functions like "extractLocales" and "ContentTypeSchema," and discusses the "ExtractRef" function for handling content references. Finally, it covers the "Sitecore Validator," which ensures that required folder structures are present before migration, and highlights limitations regarding content type migration in test stacks.

📦 migration-sitecore

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

📁 Modules

1.extractLocales`(directoryPath)`

Description:
Extracts locale-specific content from the Sitecore content tree.

Parameters:

- `directoryPath` *(string)* – Path to the Sitecore content folder (usually ends in `/content`).

Returns:
A `Promise` that resolves when locale extraction is complete.

---

2. `ExtractConfiguration(directoryPath)`

Description:
Extracts configuration files or values from the specified Sitecore project directory.

Parameters:

- `directoryPath` *(string)* – Base directory where Sitecore configuration is located.

Returns:
A `Promise`.

---

3. `contentTypes(directoryPath, affix, config)`

Description:
Parses content types from the Sitecore project and transforms them based on provided configuration.

Parameters:

- `directoryPath` *(string)* – Path where content types are located.
- `affix` *(string)* – A suffix/prefix used for naming or filtering.
- `config` *(object)* – Configuration object for processing.

Returns:
A `Promise`.

---

4. `reference()`

Description:
Generates a reference map used for linking related items during migration.

Returns:
A `Promise` that resolves with a reference map object.

---

5. `ExtractFiles` *(utility/module)*

Description:
Utility module for converting XML to JSON.

🗂️ Project Structure

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

CONTENTSTACK

```
JavaScript
migration-sitecore/

├── libs/

│   ├── contenttypes.js

│   ├── configuration.js

│   ├── reference.js

│   ├── convert.js

│   └── extractLocales.js

├── index.js
```

📄 `extractLocales` – Locale Extraction Utility

Function: `extractLocales(dir)`

The `extractLocales` function scans a directory (and its subdirectories) for `data.json` files, extracts locale codes from them, and returns a set of all unique locale/language codes found. These locales are typically used to map language content from a legacy CMS to Contentstack.

📦 Dependencies

- **`fs`** (Node.js built-in): For reading files.

- **`path`** (Node.js built-in): For resolving file paths across different operating systems.

🛠️ Parameters

- `dir` *(string)*: Path to the root directory that contains the JSON files generated from an XML export. This should be the starting point for recursive traversal.

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

🔁 Behavior

- Recursively traverses the directory to find `data.json` files.
- Validates paths and file formats.
- Extracts locale codes from valid JSON files.
- Accumulates and returns a `Set` of unique locale codes (e.g., `'en-us'`, `'fr-fr'`).
- Returns an empty set `[]` if no locales are found.
- Handles and logs any file or parsing errors.

✅ Returns

- `Set<string>` – A set of unique locale codes.
- Returns `new Set()` if none are found or if an error occurs.

📄 `ContentTypeSchema` – Field Type Mapper

Function: `ContentTypeSchema({ type, name, uid, default_value, id, choices, sourLet, sitecoreKey, affix })`

The `ContentTypeSchema` function maps legacy CMS field definitions (primarily from Sitecore) to Contentstack-compatible schema objects. It handles field type normalization, UID adjustments, and schema formatting for a wide range of input types.

🧩 Core Purpose

Transforms an input field (defined by its type, name, and other metadata) into a structured object that matches Contentstack's content type schema format. Used during automated schema migration.

🛠️ Parameters

- `type` *(string)* – Field type from the source system (e.g., `'Single-Line Text'`, `'Droplist'`, etc.).
- `name` *(string)* – Display name or label of the field.
- `uid` *(string)* – Unique identifier, optionally prefixed.
- `default_value` *(string)* – Default value assigned to the field.
- `id` *(string | number)* – Internal ID or identifier.
- `choices` *(array)* – List of options, used for dropdown-type fields.
- `sourLet` *(object)* – Optional metadata related to source mappings.
- `sitecoreKey` *(string)* – Original field identifier in Sitecore.
- `affix` *(string)* – Optional string to prepend to UIDs that match restricted names.

🔁 Behavior

- Adds an `affix` to UIDs if the UID is in a restricted list.
- Uses a large `switch-case` to match field types and output the appropriate Contentstack schema definition.
- Handles common field types such as:

    - `single_line_text`
    - `multi_line_text`
    - `boolean`
    - `html`
    - `number`
    - `file`
    - `dropdown`
    - `link`
    - `isodate`
    - `extension`
- Fields not mapped or unsupported are skipped or logged.

✅ Returns

An object formatted for Contentstack schema import, typically including:

- `id`, `uid`, `otherCmsField`, `otherCmsType`
- `contentstackField`, `contentstackFieldUid`, `contentstackFieldType`

- `backupFieldType`, `backupFieldUid`
- `advanced`: Includes `default_value` or `options` where relevant

If the field type is unsupported or intentionally skipped, it returns `undefined`.

Key	Type	Description	Example
`id`	`string`	Internal ID of the field, typically used for internal tracking or reference.	`"123"`
`uid`	`string`	UID from Sitecore (`sitecoreKey`), used to maintain link to source field.	`"sc_title"`
`otherCmsField`	`string`	Original field name from the legacy CMS.	`"Title"`
`otherCmsType`	`string`	Type of the field in the source CMS (e.g., Sitecore field type).	`"Single-Line Text"`
`contentstackField`	`string`	Field name as it will appear in Contentstack. Usually the same name.	`"Title"`
`contentstackFieldUid`	`string`	UID used by Contentstack to uniquely identify the field.	`"title"`

`contentstackFieldType`	`string`	Mapped Contentstack field type. For text input fields, it is `'single_line_text'`.	`"single_line_text"`
`backupFieldType`	`string`	Backup type for internal validation or fallbacks. Mirrors the main type.	`"single_line_text"`
`backupFieldUid`	`string`	Backup UID for traceability, usually the same as `contentstackFieldUid`.	`"title"`
`advanced.default_value`	`string | null`	Default value for the field. If not provided, it's `null`.	`"Default Title"` or `null`

📄 `ExtractRef`

Overview:

The `ExtractRef` function is a critical component in a data migration process, specifically for migrating content between two systems (e.g., from a legacy Sitecore CMS to Contentstack). The primary responsibility of this function is to handle the extraction, transformation, and mapping of content references between different content types and global fields, ensuring the migration data is properly formatted for use in Contentstack.

This function does the following:

1. **Reads Configuration Files**: It loads configuration files like `base.json`, `contentTypeKey.json`, and `treeListRef.json` to establish the relationships and references between content types and fields.

2. **Processes Content Types**: Iterates through the content types to identify references and global fields and applies necessary transformations (e.g., UID corrections, reference mappings).

3. **Updates the Data**: Writes the transformed data back to files, ensuring the proper structure and format for later use.

4. **Deletes Processed Files**: Cleans up by deleting the original content type files after processing.

5. **Returns Path and UIDs**: Provides a summary of the processed data, including paths and lists of content type and global field UIDs.

The function is primarily used to prepare content for migration by resolving field references, correcting UIDs, and making sure the data aligns with Contentstack's structure.

Detailed Breakdown:

1. Clearing Old Data (`emptyGlobalFiled`):

- The first operation is clearing the existing global field data, which ensures that the function starts with a clean slate.

- `emptyGlobalFiled();` clears the existing global field mappings before processing new data.

2. Reading Configuration Files:

The function then loads the following files using `helper.readFile`:

- **`base.json`**: Contains data on base pages. This is important because some content types may have references to base pages that need to be resolved.

- **`contentTypeKey.json`**: This file maps content type keys to their corresponding UIDs. These mappings are essential for establishing relationships between content types.

- **`treeListRef.json`**: Contains references for tree structures. It helps identify which content types have references to others, allowing the function to map them properly.

3. Initializing Key Variables:

- **`globalFieldUids`**: An empty array is initialized to collect unique global field UIDs.

- **`contentTypesPaths`**: This variable holds an array of content type file paths, which the function will iterate over to process the content types.

4. Processing Each Content Type:

- The function checks if there are valid content types and references (`base.json`, `contentTypeKey.json`, or `treeListRef.json`). If so, it proceeds to loop through each content type and process it.

Inside the loop:

- **Reading Content Type Files**: For each content type, the corresponding JSON file is read.

- **Handling References in `treeListRef.json`**:

  - If the content type has references defined in `treeListRef.json` (specifically in `refTree.unique`), the function maps these references using the keys in `contentTypeKey.json`.

  - If valid UIDs are found, the function corrects the UID if necessary and creates a reference mapping (`schemaObject`). This is then added to the content type's field mapping.

- **Handling Base Pages**:

  - The function checks if the content type has associated base pages (from `base.json`).

  - If base page references exist, the function splits them, looks up their corresponding UIDs, and creates global field mappings for them.

- **Writing the Updated Content Type Data**: After processing each content type, the function writes the updated data back to disk using `helper.writeFile`.

5. Handling Global Field UIDs:

After processing all the content types, the function proceeds to manage global field references:

- **Unique Global Field UIDs**: It collects the unique global field UIDs by removing duplicates (`new Set(globalFieldUids)`).

- **Reading Existing Global Fields**: It reads existing global fields from `GLOBAL_FIELDS_FILE`.

- **Adding New Global Fields**: It adds new global fields for the references identified in the base page references.

- **Cleaning Field Mappings**: For each global field, it removes unwanted fields such as `title` and `url` to ensure only valid fields remain in the global field mappings.

- **Writing Updated Global Fields**: The function writes the cleaned-up global field data back to the global fields directory.

6. Deleting Processed Content Type Files:

After processing each content type and writing the updated data, the function deletes the original content type files to avoid redundant files in the working directory.

7. Returning Processed Data:

Finally, the function returns an object containing:

- **Path**: The base path where the migration data is stored.

- **Content Type UIDs**: A list of content type UIDs that have been processed.

- **Global Field UIDs**: A list of global field UIDs that have been referenced during the migration.

Key Concepts:

1. **UID Correction**: UIDs are unique identifiers for content types and fields. The function ensures that if a UID is already in use, it will append `"_changed"` to create a new unique identifier. This prevents collisions during the migration process.

2. **Field Mappings**: The function builds mappings between legacy CMS content (in the form of JSON data) and Contentstack fields. This is essential for maintaining relationships and references when migrating content.

3. **Data Integrity**: It ensures that references between content types, fields, and global fields are correctly mapped, maintaining the integrity of the migrated content.

4. **File Management**: By reading, writing, and deleting files as needed, the function keeps the migration process clean and efficient.

⚙️ `Validator Sitecore`

🧾 Overview

The **Sitecore Validator** checks whether essential folder structures (`/templates`, `/content`, `/blob`, `/media library`, etc.) are present in a provided Sitecore project ZIP or file structure. This function ensures that all required directories are available before starting a Sitecore content migration or transformation pipeline.

🔧 Function: `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:

✅ `true` if:

1. All of the following directories have at least one file:
    - `/templates`
    - `/content`
    - `/blob`
    - `/media library`

❌ `false` if:

- One or more required directories are missing.
- An exception occurs during execution.

🔧 Internal Logic

1. Extract filenames from file map

```javascript
for (const filename of Object.keys(data?.files)) {
```

1. Iterates through all filenames in the provided `data.files` object.

2. Iterate over config properties

```javascript
if (filename.includes('/templates') &&
!filename.includes('properties/items/master/sitecore/templates')) { ... }
```

1. Collects paths into arrays based on folder type:
    - `templates`
    - `content`
    - `Configuration` (optional)
    - `blob`
    - `media library`

3. Await Promises (fallback for any async I/O)

```javascript
templates = await Promise.all(templates);
```

1. Resolves all path arrays (in case they contain promises or async resolution is needed).

4. Validate Required Sections

```javascript
if (templates?.length > 0 && content?.length > 0 && blob?.length > 0 &&
mediaLibrary?.length > 0) {
  return true;
}
return false;
```

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

1. Returns `true` only if all required folder categories have files.

5. ❗ Error Handling

```javascript
try { ... } catch (err) {

  console.info('🚀 ~ sitecoreValidator ~ err:', err);

  return false;

}
```

1. Any unexpected error is caught and logged.
2. Validator fails gracefully by returning `false`.

✅ Example Usage

```javascript
import sitecoreValidator from './validators/sitecore-validator';


const zipData = {

  files: {

    '/templates/article.yml': {},

    '/content/articles.json': {},

    '/blob/media1.png': {},

    '/media library/image1.jpg': {}

  }
```

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

```
};


if (await sitecoreValidator({ data: zipData })) {
  console.log('✅ Sitecore structure is valid!');
} else {
  console.error('❌ Sitecore structure is invalid or incomplete.');
}
```

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

1. Not handle the use case of deletion of existing destination stack in runtime
2. **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.