CMS Connector API - Contentful

This document outlines the functionalities and usage of the `migration-contentful` package, which is designed to simplify the process of migrating data from Contentful to Contentstack CMS. It provides key utility functions for extracting content types, mapping fields, and handling locales, ensuring a smooth and efficient migration process. The following sections detail the installation, available functions, and their specific behaviors.

`migration-contentful` Package

Overview

The `migration-contentful` package is used to simplify the process of migrating data from Contentful to Contentstack CMS. It helps with extracting content types, mapping fields, and handling locales. This package contains key utility functions to facilitate these tasks.

Installation

In your `package.json`, you reference this package from your local file system like this:

```
JavaScript
"dependencies": {

  "migration-contentful": "file:migration-contentful"

}
```

This makes the `migration-contentful` package available locally without the need to fetch it from an external registry like npm.

Key Functions

You can import the following functions from the package to use in your project:

1. **`extractContentTypes`**:
    - This function extracts content types from the exported Contentful data and saves them locally.
    - It processes the data to create a structured format that can be used for migration to another CMS.
      `const { extractContentTypes } = require('migration-contentful');`

2. **`createInitialMapper`**:
    - Once the content types are extracted, this function transforms them into a mapping schema. This schema helps in migrating the data by ensuring fields are mapped correctly between systems.
      `const { createInitialMapper } = require('migration-contentful');`
3. **`extractLocale`**:
    - This function extracts locale information (languages and region data) from Contentful exports. This is crucial for migrating content with language variations to the new CMS.
      `const { extractLocale } = require('migration-contentful');`

How to Use

To use the functions in your code, simply import them like this:

```javascript
const { extractContentTypes, createInitialMapper, extractLocale } =
require('migration-contentful');
```

These functions will allow you to:

- **Extract content types** from a Contentful export.
- **Create a field mapping** based on the extracted data.
- **Extract and manage locales** for migrating language data

`extractContentTypes` Function

Overview

This module provides utilities to **extract and save Contentful content type definitions** along with their associated editor interfaces. The data is processed and saved as structured JSON files for use in CMS migration or synchronization pipelines.

Function: `extractContentTypes(filePath, prefix)`

Description:

Reads a file containing Contentful content types and editor interfaces, processes them, and saves each content type in a structured format.

Parameters:

- `filePath` (`string`): Path to the file containing the exported Contentful content model.
- `prefix` (`string`): Prefix to namespace or label the saved field definitions.

Returns:

- `Promise<void>`: Resolves when content types are successfully processed and saved.

Behavior:

1. Checks if the folder at `contentfulFolderPath` exists. If not, it creates it.
2. Reads the input file and extracts `contentTypes` and `editorInterfaces`.
3. If `contentTypes` exist, it delegates processing to `saveContentType()`. Otherwise, it logs "No content-type found".

Throws:

- Logs and throws an error if reading the file or saving data fails.

Function: `saveContentType(contentTypes, editorInterface, prefix)`

Description:

Processes each content type and its fields by matching them with their respective editor controls and saves the data to JSON files.

Parameters:

- `contentTypes` (`Array<Object>`): Array of content type definitions.
- `editorInterface` (`Array<Object>`): Array of editor configurations for fields.
- `prefix` (`string`): Used to label field data.

Returns:

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

Behavior:

1. Converts the `sys.id` of each content type into snake_case.
2. Finds the matching editor interface by comparing content type IDs.
3. Merges field definitions with their widget/editor info.
4. Saves each content type as a prettified `.json` file in the target folder.

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:

- `prefix`
- `contentUid`
- `contentDescription`
- `contentfulID`
- `id`, `name`, `type`, `localized`, `required`, `validations`, `disabled`, `omitted` (from field definition)
- `widgetId` (from editor interface)

- `contentNames` (list of all content type IDs)

📄 `createInitialMapper` Function

Overview

The `createInitialMapper` function reads content model data exported from Contentful, transforms each model into a Contentstack-compatible schema format, and returns a complete mapping of content types.

It is typically used during CMS migrations or synchronization processes between Contentful and Contentstack.

📦 Function Signature

```JavaScript
const createInitialMapper = async () => { ... }
```

🔧 Inputs

- **None** were directly passed.

Reads files from:

```JavaScript
path.resolve(process.cwd(), `${config.data}/${config.contentful.contentful}`)
```

- where each file contains field definitions for a Contentful content type.

📤 Output

An object in the following structure:

CONTENTSTACK

```javascript
{
  contentTypes: [ ... ] // array of content type mapping objects
}
```

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 Contentful
`otherCmsUid`	`string`	UID from Contentful (`contentfulID`)
`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. Read Files

```javascript
const files = await fs.readdir(...);
```

Gets all JSON files representing Contentful content types.

---

2. Iterate Files

```javascript
for (const file of files) { ... }
```

For each file:

- Reads field definitions
- Derives content type title from filename

---

3. Build Schema Object

Each object includes:

- Metadata about the content type
- Default fields (`title` and `url`) added manually
- Custom fields generated via `contentTypeMapper()`

---

4. Generate Fields

```
JavaScript
const contentstackFields = [...uidTitle,
...contentTypeMapper(data)].filter(Boolean);
```

- `uidTitle`: Hardcoded default fields for `title` and `url`
- `contentTypeMapper(data)`: Dynamically maps Contentful fields to Contentstack schema

---

5. Delete Temporary Folder

```
deleteFolderSync(path.resolve(...));
```

Removes the data folder after processing to avoid re-use or conflicts.

---

6. Return Result

Returns the final list of mapped content types for further use.

🛑 Error Handling

Any error during file reading, transformation, or folder deletion is caught and logged:

js

CopyEdit

```
console.error('Error saving content type:', error);
```

Example Return

```
JavaScript
{

  "contentTypes": [
```

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

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

🔗 Related Functions

- `contentTypeMapper`: Converts each Contentful field to Contentstack schema.
- `uidCorrector`: Normalizes field UIDs.

- `extractAdvancedFields`: Injects metadata (e.g., mandatory, unique).
- `deleteFolderSync`: Cleans up local temporary directories.

📄 `contentTypeMapper` Function

Overview

The `contentTypeMapper` function is a comprehensive mapping utility that transforms a content model structure (typically from a CMS like Contentful) into a schema array suitable for another CMS (like Contentstack). Here's a breakdown of how it works and what each section does — suitable for use in developer documentation:

📌 Function Purpose

The `contentTypeMapper` function takes an array of content field definitions (`data`) and returns a normalized array of schema objects formatted for Contentstack. It supports various field types, widget configurations, and advanced metadata.

🧩 Function Signature

```
JavaScript
const contentTypeMapper = (data) => { ... };
```

- **Input:** `data` – An array of field definition objects, typically exported from a CMS source.
- **Output:** `schemaArray` – An array of transformed field objects structured for Contentstack.

🧠 Core Logic (How it Works)

1. Iterate Through Input Data

```
data.reduce((acc, item) => { ... }, []);
```

Each field (`item`) in the `data` array is processed based on its `type` and `widgetId`.

2. Handle Different Field Types

✅ **Rich Text**

- Calls `arrangeRte` to resolve references.
- Uses `createFieldObject` with `json` type.

✅ **Symbol, Text**

- Handles widgets like `singleLine`, `urlEditor`, `slugEditor`, `multipleLine`, `markdown`, `dropdown`, `radio`, `tagEditor`, and `listInput`.

✅ **Number, Integer**

- Maps editors to `number`, or re-maps to `dropdownNumber`, `radioNumber`, etc.

✅ **Date**

- Mapped directly to `isodate`.

✅ **Array, Link**

- **Assets (Images, Files):** Mapped to `file`, with `.multiple = true` if it's an array.
- **Entries (References):**
  - Parses `linkContentType` from `validations`.
  - Derives references (`referenceFields`) using helper logic.
  - Uses `createFieldObject` with `reference` type.
- **Checkbox, Tag Editor, List Input:** Mapped appropriately.

✅ **Boolean**

- Mapped to `boolean`.

✅ **Object**

- If `widgetId` is an objectEditor, marked as `app` type.
- Otherwise, enriches the name using app metadata from `appDetails`.

✅ **Location**

- Creates a `group` field plus subfields for `lat` and `lon`.

---

3. Helper Function: `createFieldObject`

This standardizes the schema field output:

```
JavaScript
{
  uid,
  otherCmsField,
  otherCmsType,
  contentstackField,
  contentstackFieldUid,
  contentstackFieldType,
  backupFieldType,
  backupFieldUid,
  advanced
}
```

Where `advanced` comes from `extractAdvancedFields`, providing details like:

- `mandatory`
- `unique`
- `nonLocalizable`
- `referenceFields` (when applicable)

🔧 Special Logic

- `uidCorrector()` is used to standardize IDs for compatibility.

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

- Conditional checks on `item.items`, `validations`, and `widgetId` ensure the mapper handles deeply nested or loosely structured fields gracefully.
- References are deduplicated and capped for performance (`slice(0, 9)` or `length < 25`).

🏷️ Console Logging

For widgets like `tagEditor` or `listInput`, a `console.info()` is called — likely for debugging purposes.

✅ Summary

This function is critical in migrating or transforming content model definitions from one CMS to another. It ensures:

- Widget-specific logic is respected
- Reference relationships are preserved
- Fields are enriched with necessary metadata
- Complex widgets like `Location`, `Object`, and `Reference` are handled correctly

📄 `extractLocale` Function

📝 Description

This function extracts **unique locale codes** (e.g., `en-us`, `fr-fr`) from a Contentful JSON export and returns them in an array. These locale codes are later pushed to the database via a backend API to support localization in a new CMS like Contentstack.

📦 Dependencies

- **`jsonFilePath`**: Path to the exported JSON file from Contentful.

- `fs` (Node.js built-in): Used to read the file contents.

🔧 Function: `extractLocale`

Purpose

Extracts and returns a list of unique locale codes used in the legacy CMS.

Parameters

- `jsonFilePath` (`string`) – File path to the exported Contentful data (JSON format).

Returns

`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.
- Parses the JSON and extracts values from the `locales` array.
- Collects unique locale codes (`locale.code`).
- Handles and logs errors gracefully.

📌 Example Usage

CONTENTSTACK

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


(async () => {

  const locales = await extractLocale('./legacy-export.json');

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

})();
```

⚙️ Validator Contentful

🧾 Overview

The **Contentful Validator** verifies that a JSON content model exported from Contentful contains all required properties as defined in a configuration schema (`contentful.json`). It is used as a pre-validation step before transforming or migrating content models into another CMS like Contentstack.

🔧 Function: `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:

✅ **`true`** if:

- All required properties listed in the config are present in the parsed JSON object.

❌ **`false`** if:

- Any required property is missing.

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

- The JSON is malformed or cannot be parsed.

🔧 Internal Logic

1. Parse the JSON input

```javascript
jsonData = JSON.parse(data);
```

1. Attempts to parse the input string into a valid JavaScript object.
2. If parsing fails, the function returns `false`.

2. Iterate Over Config Schema

```javascript
Object.values(jsonConfig).every((prop: any) => { ... });
```

1. Loops through each field defined in the `contentful.json` schema.

3. Validate Each Property

```javascript
if (jsonData?.hasOwnProperty(prop?.name)) {
  return true;
}
else if (prop?.required === 'true') {
  return false;
}
return true;
```

1. Checks if the required properties exist in the parsed `jsonData`.
2. If a required property is missing, the function short-circuits and returns `false`.

4. ❗ Error Handling

```
JavaScript
try { ... } catch (error) {
  return false;
}
```

1. The function handles all parsing or runtime errors.
2. If parsing fails or unexpected data structures are encountered, `false` is returned without throwing.

✅ Example Usage

```
JavaScript
import contentfulValidator from './validators/contentful-validator';
import fs from 'fs';

const rawData = fs.readFileSync('contentful-export.json', 'utf-8');

if (contentfulValidator(rawData)) {
  console.log('✅ Contentful schema is valid!');
} else {
  console.error('❌ Invalid schema: Missing required fields or bad JSON.');
}
```

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.

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

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 mapper module | existing stack | existing content type mapped | Modular blocks, taxonomy these fields can be matched with Single Line Textbox field
3. **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.

4. **Issue:** Data migration may encounter unforeseen problems if the Contentful data contains duplicate UIDs.