Backend API Reference

This document provides a comprehensive overview of various configurations, controllers, and middleware used in a backend application, primarily focused on API interactions, authentication, migrations, organization management, project management (including content mapping), user profiles, and authentication middleware. It details environment configurations, log file paths, how to load environment variables, best practices for handling sensitive data, definitions of constants for various application-wide settings, HTTP request handlers for different entities, and middleware for authenticating users via JWT tokens. The documentation emphasizes maintainability, consistency, and security in application development.

The `devConfig` object centralizes environment-specific API endpoints, application URLs, and log file path configuration for development and testing purposes. This configuration is intended to be imported and used throughout your codebase to ensure consistency and ease of maintenance.

1. `CS_API`

Contains the base URLs for the Contentstack API for different regions and cloud providers. Use these endpoints to make API requests depending on the deployment environment.

Keys:

- `NA`: North America (default/staging)

- `EU`: Europe (default/staging)

- `AZURE_NA`: North America on Azure cloud

- `GCP_NA`: North America on Google Cloud Platform

2. `CS_URL`

Contains the base URLs for the Contentstack web application for different regions and cloud providers. Use these URLs to direct users to the appropriate Contentstack UI.

Keys:

- `NA`: North America (default)

- `EU`: Europe (default)

- `AZURE_NA`: North America on Azure cloud

- `AZURE_EU`: Europe on Azure cloud

- `GCP_NA`: North America on Google Cloud Platform

3. `LOG_FILE_PATH`

Specifies the path to the log file used by the application. The path is determined based on the operating system to ensure compatibility:

- On Windows (`win32`), uses `.\\combine.log`

- On other platforms (Linux, macOS), uses `./combine.log`

- Environment Selection:

Choose the appropriate API and URL keys based on the region and cloud provider your application is targeting.

- Log File Path:

Update the log file path if your application's logging requirements change or if you need to store logs in a different location.

This configuration file is designed to be easily extendable. Add new regions, cloud providers, or configuration keys as your infrastructure evolves.

Here is a documentation-style explanation for your `index.ts` configuration file. This covers the purpose, structure, and usage of the file, as well as best practices and important notes about environment variable management.

Configuration Module (`index.ts`)

This module is responsible for loading environment variables, selecting the appropriate configuration based on the environment, and exporting a strongly-typed configuration object for use throughout your application.

- Environment Variable Loading: Uses the `dotenv` package to load environment variables from a file named according to the current `NODE_ENV` (e.g., `production.env`, `development.env`).

- Type Safety:Defines a `ConfigType` TypeScript type to ensure all required configuration values are present and correctly typed.

- Environment-Specific Config: Dynamically merges either `prodConfig` or `devConfig` into the exported configuration object, depending on the value of `NODE_ENV`.

- The `.env` files should be named as `<NODE_ENV>.env` (e.g., `production.env`, `development.env`).

- These files should **not be committed to version control as they may contain sensitive information.**

Configuration Type

- Defines the shape of the configuration object.

- Ensures that all required configuration values are present and correctly typed.

Exported Configuration Object

- Sets static and environment-based configuration values.

- Uses the spread operator to merge in either `prodConfig` or `devConfig` based on the current environment.

- The `!` (non-null assertion) is used to tell TypeScript that these environment variables are guaranteed to be present.

Best Practices

- **Do not commit `.env` files to version control. Add them to `.gitignore`.**

- **Validate required environment variables at startup to avoid runtime errors.**

- **Use different `.env` files for each environment (development, production, etc.).**

- **Keep secrets and sensitive data out of your codebase and only in environment variables.**

Troubleshooting

- Ensure the `.env` file for the current `NODE_ENV` exists in your project root.

- If a variable is `undefined`, check the spelling and presence in the `.env` file.

- The `dotenv` config should be called **before any code that uses environment variables.**

Constants/index.ts

The `constants/index.ts` file serves as a centralized location for all application-wide constants, configuration values, and enumerations used throughout the project. This approach promotes maintainability, consistency, and ease of updates.

Overview

This file exports a variety of constants, including:

- **Region and environment identifiers**

- API endpoints and URLs
- CMS types and project modules
- HTTP status codes and messages
- Validation error messages
- Project and content type statuses
- Locale mappings
- Configuration for migration data directories and files

Key Sections

Regions and URLs

- `CS_REGIONS`: Lists supported regions for the application.
- `DEVURLS`: Maps each region to its corresponding developer hub API URL.

CMS and Modules

- `CMS`: Enumerates supported CMS types.
- `MODULES` and `MODULES_ACTIONS`: Define available modules and their possible actions.

HTTP Codes and Messages

- `HTTP_CODES`: Maps common HTTP status codes to descriptive names.
- `HTTP_TEXTS`: Provides user-friendly messages for various HTTP responses and error scenarios.
- `HTTP_RESPONSE_HEADERS`: Default headers for API responses.

Validation and Error Handling

- `VALIDATION_ERRORS`: Standardized error messages for input validation.
- `METHODS_TO_INCLUDE_DATA_IN_AXIOS`: HTTP methods that should include a data payload in Axios requests.

Project and Content Type Status

- `PROJECT_STATUS`, `NEW_PROJECT_STATUS`, `PREDEFINED_STATUS`, `PREDEFINED_STEPS`: Enumerate possible project states and workflow steps.
- `CONTENT_TYPE_STATUS`: Enumerates content type mapping statuses.

Locale and Field Mapping

- `LOCALE_MAPPER`: Maps master and other locales between systems.
- `POPULATE_CONTENT_MAPPER`, `POPULATE_FIELD_MAPPING`, `CONTENT_TYPE_POPULATE_FIELDS`: Fields used for populating related data in queries.

Miscellaneous

- `CHUNK_SIZE`: Default chunk size for file operations.
- `LIST_EXTENSION_UID`: Unique identifier for a specific extension.
- `KEYTOREMOVE`: List of keys to be excluded or removed in certain operations.
- `AFFIX_REGEX`: Regular expression for validating project affixes.

Migration Data Configuration

- `MIGRATION_DATA_CONFIG`: Centralizes all directory and file names used for migration data, backups, locales, webhooks, environments, content types, marketplace apps, extensions, references, assets, entries, authors, categories, tags, terms, posts, chunks, global fields, and export info.

Usage

Import any constant as needed in your modules:

```
import { CS_REGIONS, HTTP_CODES, HTTP_TEXTS } from '../constants'
```

This ensures that all parts of the application use the same values, reducing the risk of typos and inconsistencies.

CONTENTSTACK

Maintenance

- Add new constants here as the application grows.
- Update values centrally to propagate changes throughout the codebase.
- Group related constants together and use clear, descriptive names.

Example

To check if a project is in a "Draft" state:

```
if (project.status === PROJECT_STATUS.DRAFT) {
  // handle draft logic
}
```

To get a user-friendly error message for a 401 response:

```
const message = HTTP_TEXTS.UNAUTHORIZED;
```

This file is a single source of truth for all static values and configuration keys, making the codebase easier to manage and understand.

/src/controllers/auth.controller.ts

The `auth.controller.ts` file defines controller functions for handling authentication-related HTTP requests in the application. These controllers act as intermediaries between incoming Express requests and the authentication service logic.

Overview

This module exports the `authController` object, which contains methods for:

- Handling user login requests
- Handling requests to send SMS messages (e.g., for two-factor authentication)

Each controller method receives the Express `Request` and `Response` objects, delegates the main logic to the `authService`, and sends the appropriate HTTP response.

CONTENTSTACK

Exported Controller Methods

1. `login`

Handles user login requests.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `authService.login(req)` to perform authentication.
  - Sends the response with the status and data returned by the service.

Usage Example:
```
POST /api/auth/login
```

2. `RequestSms`

Handles requests to send an SMS, typically for authentication or verification purposes.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `authService.requestSms(req)` to trigger the SMS sending logic.
  - Sends the response with the status and data returned by the service.

Usage Example:
```
POST /api/auth/request-sms
```

Example Usage

Import and use the controller in your route definitions:

```
import { authController } from './controllers/auth.controller';

router.post('/login', authController.login);

router.post('/request-sms',                                    ;
```

src/controllers/migration.controller.ts

The `migration.controller.ts` file defines controller functions for handling migration-related HTTP requests. These controllers serve as the interface between Express routes and the migration service logic, managing the migration workflow for projects.

Overview

This module exports the `migrationController` object, which provides methods for:

- **Creating and deleting test stacks**
- **Starting test and final migrations**
- **Retrieving migration logs**
- **Saving source locales and mapped locales**

Each controller method receives the Express `Request` and `Response` objects, delegates the main logic to the `migrationService`, and sends the appropriate HTTP response.

Exported Controller Methods

1. `createTestStack`

Creates a test stack for migration.

- **Parameters:**
  - **`req`: Express `Request` object**
  - **`res`: Express `Response` object**
- **Process:**

- Calls `migrationService.createTestStack(req)` to create a test stack.
- Responds with the status and data from the service.

---

2. `deleteTestStack`

Deletes an existing test stack.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `migrationService.deleteTestStack(req)` to delete the test stack.
  - Responds with the result from the service.

---

3. `startTestMigration`

Initiates a test migration process.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `migrationService.startTestMigration(req)` to start the test migration.
  - Responds with the result from the service.

---

4. `startMigration`

Starts the final migration process.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `migrationService.startMigration(req)` to start the final migration.
  - Responds with the result from the service.

---

5. `getLogs`

Retrieves migration logs for a project or stack.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `migrationService.getLogs(req)` to fetch logs.
  - Responds with the logs data.

---

6. `saveLocales`

Saves the source locales for a migration.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object

- **Process:**
    - **Calls `migrationService.createSourceLocales(req)` to save locales.**
    - **Responds with the result.**

---

7. `saveMappedLocales`

Saves or updates the mapped locales for a migration.

- **Parameters:**
    - **`req`: Express `Request` object**
    - **`res`: Express `Response` object**
- **Process:**
    - **Calls `migrationService.updateLocaleMapper(req)` to update locale mappings.**
    - **Responds with the result.**

Example Usage

Import and use the controller in your route definitions:

```
import { migrationController } from './controllers/migration.controller';


router.post('/test-stack', migrationController.createTestStack);
router.delete('/test-stack', migrationController.deleteTestStack);
router.post('/test-migration', migrationController.startTestMigration);
router.post('/migration', migrationController.startMigration);
router.get('/logs', migrationController.getLogs);
router.post('/locales', migrationController.saveLocales);
```

```
router.post('/mapped-locales', migrationController.saveMappedLocales);
```

---

src/controllers/org.controller.ts

The `org.controller.ts` file defines controller functions for handling organization and stack-related HTTP requests. These controllers act as the interface between Express routes and the organization service logic, managing operations such as stack management, locale retrieval, and organization details.

Overview

This module exports the `orgController` object, which provides methods for:

- **Retrieving all stacks in an organization**
- **Creating a new stack**
- **Fetching organization and stack locales**
- **Checking stack status**
- **Retrieving organization details**

Each controller method receives the Express `Request` and `Response` objects, delegates the main logic to the `orgService`, and sends the appropriate HTTP response.

Exported Controller Methods

1. `getAllStacks`

Retrieves all stacks associated with the organization.

- **Parameters:**
  - **`req`: Express `Request` object**
  - **`res`: Express `Response` object**
- **Process:**

- Calls `orgService.getAllStacks(req)` to fetch stacks.
- Responds with the status and data from the service.

---

2. `createStack`

Creates a new stack within the organization.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `orgService.createStack(req)` to create the stack.
  - Responds with the status and data from the service.

---

3. `getLocales`

Retrieves the locales configured for the organization.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `orgService.getLocales(req)` to fetch locales.
  - Responds with the status and data from the service.

---

4. `getStackStatus`

Retrieves the status of a specific stack.

- **Parameters:**
    - `req`: Express `Request` object
    - `res`: Express `Response` object
- **Process:**
    - Calls `orgService.getStackStatus(req)` to get the stack status.
    - Responds with the status and data from the service.

---

5. `getStackLocale`

Retrieves the locales for a specific stack.

- **Parameters:**
    - `req`: Express `Request` object
    - `res`: Express `Response` object
- **Process:**
    - Calls `orgService.getStackLocale(req)` to fetch stack locales.
    - Responds with the status and data from the service.

---

6. `getOrgDetails`

Retrieves details about the organization.

- **Parameters:**
    - `req`: Express `Request` object
    - `res`: Express `Response` object
- **Process:**
    - Calls `orgService.getOrgDetails(req)` to fetch organization details.

- Responds with the status and data from the service.

Example Usage

Import and use the controller in your route definitions:

```
import { orgController } from './controllers/org.controller';


router.get('/stacks', orgController.getAllStacks);
router.post('/stacks', orgController.createStack);
router.get('/locales', orgController.getLocales);
router.get('/stack-status', orgController.getStackStatus);
router.get('/stack-locales', orgController.getStackLocale);
router.get('/org-details',                                  ;
```

---

src/controllers/projects.contentMapper.controller.ts

The `projects.contentMapper.controller.ts` file defines controller functions for managing content mapping operations within projects. These controllers serve as the interface between Express routes and the `contentMapperService`, handling requests related to content types, field mappings, global fields, and content mapper updates.

Overview

This module exports the `contentMapperController` object, which provides methods for:

- **Retrieving and updating content types and field mappings**
- **Managing test data and content type fields**
- **Handling global fields**
- **Resetting and removing content mappers**

Each controller method receives the Express `Request` and `Response` objects, delegates the main logic to the `contentMapperService`, and sends the appropriate HTTP response.

Exported Controller Methods

1. `putTestData`

Updates test data for content mapping.

- Parameters:
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- Process:
  - Calls `contentMapperService.putTestData(req)` to update test data.
  - Responds with the status and data from the service.

---

2. `getContentTypes`

Retrieves available content types.

- Parameters:
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- Process:
  - Calls `contentMapperService.getContentTypes(req)` to fetch content types.
  - Responds with the status and data from the service.

---

3. `getFieldMapping`

Retrieves field mapping for a given request.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `contentMapperService.getFieldMapping(req)` to fetch field mapping.
  - Responds with the status and data from the service.

---

4. `getExistingContentTypes`

Retrieves existing content types.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `contentMapperService.getExistingContentTypes(req)` to fetch existing content types.
  - Responds with status 201 and the data from the service.

---

5. `getExistingGlobalFields`

Retrieves existing global fields.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object

- **Process:**
    - **Calls `contentMapperService.getExistingGlobalFields(req)` to fetch global fields.**
    - **Responds with status 201 and the data from the service.**

---

6. `putContentTypeFields`

Updates content type fields.

- **Parameters:**
    - **`req`: Express `Request` object**
    - **`res`: Express `Response` object**
- **Process:**
    - **Calls `contentMapperService.updateContentType(req)` to update fields.**
    - **Responds with the status and data from the service.**

---

7. `resetContentType`

Resets a content type to its initial mapping.

- **Parameters:**
    - **`req`: Express `Request` object**
    - **`res`: Express `Response` object**
- **Process:**
    - **Calls `contentMapperService.resetToInitialMapping(req)` to reset the mapping.**
    - **Responds with the status and data from the service.**

---

8. `removeContentMapper`

Removes a content mapper.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `contentMapperService.removeContentMapper(req)` to remove the mapper.
  - Responds with status 200 and the data from the service.

---

9. `getSingleContentTypes`

Retrieves a single content type.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `contentMapperService.getSingleContentTypes(req)` to fetch a single content type.
  - Responds with status 201 and the data from the service.

---

10. `getSingleGlobalField`

Retrieves a single global field.

- **Parameters:**
  - `req`: Express `Request` object

- `res`: Express `Response` object

- **Process:**
  - Calls `contentMapperService.getSingleGlobalField(req)` to fetch a single global field.
  - Responds with status 201 and the data from the service.

---

11. `updateContentMapper`

Updates content mapping details for a project.

- **Parameters:**
  - `req`: Express `Request` object
  - `res`: Express `Response` object
- **Process:**
  - Calls `contentMapperService.updateContentMapper(req)` to update mapping details.
  - Responds with the status and data from the service.

Example Usage

Import and use the controller in your route definitions:

```
import { contentMapperController } from
'./controllers/projects.contentMapper.controller';


router.put('/test-data', contentMapperController.putTestData);

router.get('/content-types', contentMapperController.getContentTypes);

router.get('/field-mapping', contentMapperController.getFieldMapping);

router.get('/existing-content-types',
contentMapperController.getExistingContentTypes);
```

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

```
router.get('/existing-global-fields',
contentMapperController.getExistingGlobalFields);

router.put('/content-type-fields',
contentMapperController.putContentTypeFields);

router.post('/reset-content-type',
contentMapperController.resetContentType);

router.delete('/content-mapper',
contentMapperController.removeContentMapper);

router.get('/single-content-type',
contentMapperController.getSingleContentTypes);

router.get('/single-global-field',
contentMapperController.getSingleGlobalField);

router.put('/update-content-mapper',
contentMapperController.updateContentMapper);
```

---

src/controllers/projects.controller.ts

The `projects.controller.ts` file defines controller functions for managing project-related operations. These controllers act as the interface between Express routes and the `projectService`, handling requests for creating, updating, retrieving, and deleting projects, as well as managing project-specific settings and workflow steps.

Overview

This module exports the `projectController` object, which provides methods for:

- **Project CRUD operations**
- **Updating project settings (legacy CMS, affix, file format, destination stack)**
- **Handling confirmations for affix and file format**
- **Managing project workflow steps and migration execution**
- **Reverting projects and retrieving migrated stacks**

Each controller method receives the Express `Request` and `Response` objects, delegates the main logic to the `projectService`, and sends the appropriate HTTP response.

CONTENTSTACK

Exported Controller Methods

1. `getAllProjects`

Retrieves all projects.

- **Calls `projectService.getAllProjects(req)` and responds with status 200 and the list of projects.**

---

2. `getProject`

Retrieves a single project based on the request.

- **Calls `projectService.getProject(req)` and responds with status 200 and the project data.**

---

3. `createProject`

Creates a new project.

- **Calls `projectService.createProject(req)` and responds with status 201 and the created project.**

---

4. `updateProject`

Updates an existing project.

- **Calls `projectService.updateProject(req)` and responds with status 200 and the updated project.**

---

5. `updateLegacyCMS`

Updates the legacy CMS configuration for a project.

- Calls `projectService.updateLegacyCMS(req)` and responds with the service's status and data.

---

6. `updateAffix`

Updates the affix for a project.

- Calls `projectService.updateAffix(req)` and responds with the service's status and data.

---

7. `affixConfirmation`

Handles affix confirmation for a project.

- Calls `projectService.affixConfirmation(req)` and responds with the service's status and data.

---

8. `updateFileFormat`

Updates the file format for a project.

- Calls `projectService.updateFileFormat(req)` and responds with the service's status and data.

---

9. `fileformatConfirmation`

Handles file format confirmation for a project.

- Calls `projectService.fileformatConfirmation(req)` and responds with the service's status and data.

---

10. `updateDestinationStack`

Updates the destination stack for a project.

- **Calls `projectService.updateDestinationStack(req)` and responds with the service's status and data.**

---

11. `updateCurrentStep`

Updates the current step of a project workflow.

- **Calls `projectService.updateCurrentStep(req)` and responds with status 200 and the updated project.**

---

12. `deleteProject`

Deletes a project.

- **Calls `projectService.deleteProject(req)` and responds with status 200 and the deleted project data.**

---

13. `revertProject`

Reverts a project to a previous state.

- **Calls `projectService.revertProject(req)` and responds with the service's status and data.**

---

14. `updateStackDetails`

Updates stack details for a project.

- **Calls `projectService.updateStackDetails(req)` and responds with the service's status and data.**

---

15. `updateMigrationExecution`

Updates migration execution details for a project.

- **Calls `projectService.updateMigrationExecution(req)` and responds with the service's status and data.**

---

16. `getMigratedStacks`

Retrieves stacks that have been migrated for a project.

- **Calls `projectService.getMigratedStacks(req)` and responds with the service's status and data.**

Example Usage

Import and use the controller in your route definitions:

```
import { projectController } from './controllers/projects.controller';


router.get('/projects', projectController.getAllProjects);

router.get('/projects/:id', projectController.getProject);

router.post('/projects', projectController.createProject);

router.put('/projects/:id', projectController.updateProject);

router.put('/projects/:id/legacy-cms', projectController.updateLegacyCMS);

router.put('/projects/:id/affix', projectController.updateAffix);

router.post('/projects/:id/affix-confirmation',
projectController.affixConfirmation);

router.put('/projects/:id/file-format',
projectController.updateFileFormat);

router.post('/projects/:id/file-format-confirmation',
projectController.fileformatConfirmation);
```

```
router.put('/projects/:id/destination-stack', projectController.updateDestinationStack);

router.put('/projects/:id/current-step', projectController.updateCurrentStep);

router.delete('/projects/:id', projectController.deleteProject);

router.post('/projects/:id/revert', projectController.revertProject);

router.put('/projects/:id/stack-details', projectController.updateStackDetails);

router.put('/projects/:id/migration-execution', projectController.updateMigrationExecution);

router.get('/projects/:id/migrated-stacks',
                                                    ;
```

---

src/controllers/user.controller.ts

The `user.controller.ts` file defines controller functions for handling user-related operations. This controller acts as the interface between Express routes and the `userService`, focusing on retrieving user profile information.

Overview

This module exports the `userController` object, which currently provides a single method:

- **Retrieving the authenticated user's profile**

The controller method receives the Express `Request` and `Response` objects, delegates the main logic to the `userService`, and sends the appropriate HTTP response.

Exported Controller Methods

1. `getUserProfile`

Retrieves the profile information for the current user.

- **Parameters:**
    - `req`: Express `Request` object
    - `res`: Express `Response` object
- **Process:**
    - Calls `userService.getUserProfile(req)` to fetch the user's profile data.
    - Responds with the status and data provided by the service.

Example Usage

Import and use the controller in your route definitions:

```
import { userController } from './controllers/user.controller';

router.get('/user/profile', userController.getUserProfile);
```

Note :-

- The controller method is asynchronous and uses `await` to handle the service response.
- The business logic is encapsulated in the `authService,migrationService,orgService,contentMapperService,projectService,userService`, keeping the controller focused on request/response handling.
- HTTP status codes and response data are determined by the service layer, allowing for flexible error handling and messaging.

src/middlewares/auth.middleware.ts

This middleware provides authentication for incoming HTTP requests by verifying a JWT token. It ensures that only requests with a valid token can access protected routes. The middleware checks for the presence of an `app_token` header, verifies the token using the application's secret key, and attaches the decoded payload to the request object for downstream use.

Exported Middleware

`authenticateUser`

Purpose:

- **Authenticates requests by validating the JWT token provided in the `app_token` header.**
- **Parameters:**
    1. **`req`: Express Request object**
    2. **`res`: Express Response object**
    3. **`next`: Express NextFunction callback**
- **Process:**
    1. **Retrieves the token from the `app_token` header.**
    2. **If the token is missing, responds with a 401 Unauthorized status and an error message.**
    3. **Verifies the token using the secret key from the configuration.**
    4. **If verification fails, responds with a 401 Unauthorized status and an error message.**
    5. **If verification succeeds, attaches the decoded payload to `req.body.token_payload`.**
    6. **Calls `next()` to pass control to the next middleware or route handler.**

Notes

- **The middleware expects the JWT token to be provided in the `app_token` header.**
- **If the token is missing or invalid, the request is rejected with a 401 Unauthorized response.**

- The decoded token payload is attached to `req.body.token_payload` for use in subsequent handlers.
- The secret key for token verification is sourced from the application configuration (`config.APP_TOKEN_KEY`).
- This middleware should be applied to any route that requires authentication.

Best Practices

- Always use this middleware on routes that require user authentication.
- Never expose sensitive information from the token payload in responses.
- Ensure the secret key is securely managed and not hard-coded in the codebase.

This middleware helps enforce secure access control across your application's protected endpoints.

`src/middlewares/auth.uploadService.middleware.ts`

Overview

This middleware secures routes intended for the upload service by validating a secret key provided in the request headers. Only requests with the correct secret key, as configured in the application settings, are allowed to proceed. This helps prevent unauthorized access to file upload endpoints.

Exported Middleware

`authenticateUploadService`

Purpose:

- Authenticates requests to upload service routes by checking the `secret_key` header against the configured file upload key.
- Parameters:
    1. `req`: Express Request object

2. `res`: Express Response object
3. `next`: Express NextFunction callback

- **Process:**
  1. Retrieves the `secret_key` from the request headers.
  2. Compares the provided key to the configured `FILE_UPLOAD_KEY`.
  3. If the key is invalid or missing, responds with a 401 Unauthorized status and an error message.
  4. If the key is valid, calls `next()` to pass control to the next middleware or route handler.

Example Usage

```
import { authenticateUploadService } from
'./middlewares/auth.uploadService.middleware';


router.post('/upload', authenticateUploadService, (req, res) => {
  // Handle file upload logic here
  res.json({ message: 'Upload authorized and successful!' });
});
```

Notes

- The middleware expects the secret key to be provided in the `secret_key` header.
- If the key does not match the configured value (`config.FILE_UPLOAD_KEY`), the request is rejected with a 401 Unauthorized response.
- Use this middleware on any route that should be restricted to trusted upload services or clients.
- The file upload key should be securely managed and never exposed in client-side code.

CONTENTSTACK

Best Practices

- **Always protect sensitive upload endpoints with this middleware.**
- **Rotate the file upload key periodically and update the configuration accordingly.**
- **Never log or expose the secret key in responses or logs.**

This middleware ensures that only authorized services or clients can access your application's file upload functionality.

`src/models/Authentication.ts`

This module provides a database interface for managing authentication data, specifically user records, using a JSON file as persistent storage. It leverages the lowdb library with Lodash utilities for convenient data manipulation.

Overview

- **Purpose:**
  To define the structure and provide access to the authentication data store, which contains user authentication details.
- **Storage:**
  Data is persisted in a JSON file located at `database/authentication.json` in the project root.
- **Tech Stack:**
  - **lowdb for lightweight JSON database operations.**
  - **Lodash utilities for enhanced querying and data manipulation.**
  - **TypeScript for type safety.**

Interface: `AuthenticationDocument`

Defines the structure of the authentication data stored in the database.

```
interface AuthenticationDocument {
  users: {
    user_id: string;
    email: string;
    region: string;
    authtoken: string;
    created_at: string;
    updated_at: string;
  }[];
}
```

- **users:**
  An array of user objects, each containing:
  - `user_id`: **Unique identifier for the user.**
  - `email`: **User's email address.**
  - `region`: **User's region or location.**
  - `authtoken`: **Authentication token for the user.**
  - `created_at`: **Timestamp of when the user was created.**
  - `updated_at`: **Timestamp of the last update to the user record.**

Default Data

Defines the initial structure of the database if the JSON file does not exist or is empty.

```
const defaultData: AuthenticationDocument = { users: [] };
```

Database Instance

Creates and exports a singleton database instance for authentication data.

```
const db = new LowWithLodash(

  new JSONFile<AuthenticationDocument>(path.join(process.cwd(),
"database", "authentication.json")),

  defaultData

);


export default
```

- **File Path:**
  The database file is located at `<project_root>/database/authentication.json`**.**
- **LowWithLodash:**
  A utility wrapper (imported from `../utils/lowdb-lodash.utils.js`**) that combines lowdb with Lodash methods for easier data manipulation.**
- **Default Data:**
  If the file does not exist, it is initialized with an empty `users` **array.**

Notes

- **Persistence:**
  All changes to the database must be followed by a call to `db.write()` **to persist changes to disk.**
- **Type Safety:**
  The use of TypeScript interfaces ensures that only valid user objects are stored.
- **Extensibility:**
  Additional fields or methods can be added to the `AuthenticationDocument` **or the database instance as needed.**

This module defines the data model and database instance for managing content type mappings between Contentstack and another CMS. It uses lowdb with a custom Lodash wrapper for convenient data access and manipulation. The data is persisted in a JSON file.

Imports

```
import { JSONFile } from "lowdb/node";

import path from 'path';

import LowWithLodash from "../utils/lowdb-lodash.utils.js";
```

- **JSONFile: Adapter for lowdb to read/write JSON files.**
- **path: Node.js module for handling file paths.**
- **LowWithLodash: Custom utility that wraps lowdb with Lodash for enhanced querying.**

Interfaces

`ContentTypesMapper`

Represents a mapping between a Contentstack content type and a content type in another CMS.

```
export interface ContentTypesMapper {

  id: string;                       // Unique identifier for the mapper entry

  projectId: string;                // Associated project ID

  otherCmsTitle: string;            // Title of the content type in the other
CMS

  otherCmsUid: string;              // Unique identifier in the other CMS

  isUpdated: boolean;               // Whether the mapping has been updated

  updateAt: Date;                   // Last update timestamp

  contentstackTitle: string;        // Title in Contentstack

  contentstackUid: string;          // UID in Contentstack

  status: number;                   // Status code (e.g., active, inactive)

  fieldMapping: [];                 // Field mapping between the two content
types

  type: string;                     // Type/category of the content type

}
```

`ContentTypeMapperDocument`

Represents the structure of the JSON document stored in the database.

```
interface ContentTypeMapperDocument {
  ContentTypesMappers: ContentTypesMapper[];
}
```

Default Data

Defines the default structure for the database file if it does not exist.

```
const defaultData: ContentTypeMapperDocument = { ContentTypesMappers: [] };
```

Database Instance

Initializes a lowdb instance with Lodash utilities, using a JSON file for persistence.

```
const db = new LowWithLodash(
  new JSONFile<ContentTypeMapperDocument>(path.join(process.cwd(), "database", 'contentTypesMapper.json')),
  defaultData
);
```

- **db: The main database instance for reading and writing content type mappings.**

Exports

```
export default db;
```

- **Exports the database instance for use in other modules.**

Notes

- The `fieldMapping` property is typed as an empty array (`[]`). For better type safety, consider defining a specific type for field mappings.
- The `updateAt` property is a `Date` object. When persisting to JSON, ensure proper serialization/deserialization.

src/models/FieldMapper.ts

This module defines the data model and database instance for managing field mapping configurations between Contentstack and other CMS platforms. It leverages lowdb for lightweight JSON-based storage, with Lodash utilities for enhanced querying.

Imports

```
import { JSONFile } from "lowdb/node";
import LowWithLodash from "../utils/lowdb-lodash.utils.js";
import path from "path";
```

- **JSONFile: Adapter for lowdb to read/write JSON files.**
- **LowWithLodash: Custom utility extending lowdb with Lodash methods.**
- **path: Node.js module for handling file paths.**

Advanced Interface

```
export interface Advanced {
  validationRegex: string;
  mandatory: boolean;
  multiple: boolean;
  unique: boolean;
  nonLocalizable: boolean;
  embedObject: boolean;
```

```
  embedObjects: any;
  minChars: string;
  maxChars: number;
  default_value: string;
  description: string;
  validationErrorMessage: string;
  options: any[];
}
```

Description:

Defines advanced configuration options for a field mapping, such as validation rules, localization, embedding, and UI options.

- `validationRegex`: **Regular expression for validating field values.**
- `mandatory`: **Whether the field is required.**
- `multiple`: **Whether the field accepts multiple values.**
- `unique`: **Whether the field value must be unique.**
- `nonLocalizable`: **If true, the field is not localizable.**
- `embedObject`: **If true, the field embeds an object.**
- `embedObjects`: **Additional embedded object configuration.**
- `minChars`: **Minimum character length for the field value.**
- `maxChars`: **Maximum character length for the field value.**
- `default_value`: **Default value for the field.**
- `description`: **Description of the field.**
- `validationErrorMessage`: **Custom error message for validation failures.**
- `options`: **List of selectable options (for dropdowns, etc.).**

FieldMapper Interface

```
interface FieldMapper {
  field_mapper: {
    id: string;
    projectId: string;
    contentTypeId: string;
    uid: string;
    otherCmsField: string;
    otherCmsType: string;
    contentstackField: string;
    contentstackFieldUid: string;
    contentstackFieldType: string;
    isDeleted: boolean;
    backupFieldType: string;
    backupFieldUid: string;
    refrenceTo: { uid: string; title: string };
    advanced: Advanced;
  }[];
}
```

Description:

Represents the structure of the field mapping data stored in the database.

- `id`: **Unique identifier for the field mapping.**
- `projectId`: **Associated project ID.**
- `contentTypeId`: **Content type ID in Contentstack.**
- `uid`: **Unique identifier for the mapping.**
- `otherCmsField`: **Field name in the other CMS.**
- `otherCmsType`: **Field type in the other CMS.**

- **`contentstackField`: Field name in Contentstack.**
- **`contentstackFieldUid`: Field UID in Contentstack.**
- **`contentstackFieldType`: Field type in Contentstack.**
- **`isDeleted`: Soft delete flag.**
- **`backupFieldType`: Backup field type.**
- **`backupFieldUid`: Backup field UID.**
- **`refrenceTo`: Reference to another field (with `uid` and `title`).**
- **`advanced`: Advanced configuration options (see above).**

Default Data

```
const defaultData: FieldMapper = { field_mapper: [] };
```

Description:

Initializes the database with an empty array of field mappings if the JSON file does not exist.

Database Instance

```
const db = new LowWithLodash(
  new JSONFile<FieldMapper>(path.join(process.cwd(), "database", "field-mapper.json")),
  defaultData
);
```

Description:

Creates a lowdb database instance for field mappings, stored at `database/field-mapper.json` in the project root.

- **Uses the `LowWithLodash` utility for Lodash-powered queries.**

- **Initializes with `defaultData` if the file is missing.**

Export

```
export default db;
```

Description:

Exports the database instance for use in other modules.

/src/models/project-lowdb.ts

This module defines the data model and database instance for managing project-related data using lowdb with Lodash utilities. It provides TypeScript interfaces for the project structure and initializes a persistent JSON-based database for storing project information.

Imports

- **path: Node.js module for handling file and directory paths.**
- **JSONFile: Adapter from `lowdb/node` for reading and writing JSON files.**
- **LowWithLodash: Custom utility that extends lowdb with Lodash methods for easier data manipulation.**

Interfaces

`LegacyCMS`

Represents metadata and configuration for a legacy CMS file and its storage details.

```
interface LegacyCMS {
  cms: string;
  affix: string;
  affix_confirmation: boolean;
  file_format: string;
```

```
  file_format_confirmation: boolean;
  file: {
    id: string;
    name: string;
    size: number;
    type: string;
    path: string;
  };
  awsDetails: {
    awsRegion: string;
    bucketName: string;
    buketKey: string;
  };
  file_path: string;
  is_fileValid: boolean;
  is_localPath: boolean;
}
```

StackDetails

Describes a stack's metadata, such as its unique identifier, label, and creation details.

```
interface StackDetails {
  uid: string;
  label: string;
  master_locale: string;
  created_at: string;
  isNewStack: boolean;
}
```

ExecutionLog

Represents a log entry for a project execution, including a URL and timestamp.

```
interface ExecutionLog {
  log_url: string;
  date: Date;
}
```

Project

Defines the structure of a project, including metadata, status, related stacks, migration state, and logs.

```
interface Project {
  id: string;
  region: string;
  org_id: string;
  owner: string;
  created_by: string;
  updated_by: string;
  former_owner_ids: [];
  name: string;
  description: string;
  status: number;
  current_step: number;
  destination_stack_id: string;
  test_stacks: [];
  current_test_stack_id: string;
  legacy_cms: LegacyCMS;
  content_mapper: any[];
  execution_log: [ExecutionLog];
  created_at: string;
```

```
  updated_at: string;
  isDeleted: boolean;
  isNewStack: boolean;
  newStackId: string;
  stackDetails: [];
  mapperKeys: {};
  extract_path: string;
  isMigrationStarted: boolean;
  isMigrationCompleted: boolean;
  migration_execution: boolean;
}
```

`ProjectDocument`

The root object for the database, containing an array of projects.

```
interface ProjectDocument {
  projects: Project[];
}
```

Default Data

Defines the initial structure for the database if the file does not exist.

```
const defaultData: ProjectDocument = { projects: [] };
```

Database Instance

Initializes a lowdb instance with Lodash utilities, using a JSON file at `database/project.json` in the project root.

```
const db = new LowWithLodash(
```

```
  new JSONFile<ProjectDocument>(path.join(process.cwd(), "database", "project.json")),
  defaultData
);
```

- **db: The exported database instance. Use this to read, write, and manipulate project data.**

src/models/types.ts

These interfaces are used throughout the application to ensure type safety and consistency when handling user data, authentication payloads, service responses, migration queries, and locale information.

Interfaces

1. `User`

Represents a user in the system.

Property	Type	Description
email	string	The email address of the user.
password	string	The password of the user.

```
export interface User {
  email: string;
  password: string;
}
```

---

2. AppTokenPayload

Represents the payload contained within an application token, typically used for authentication and authorization.

Property	Type	Description
region	string	The region associated with the user or token.
user_id	string	The unique identifier of the user.

```
export interface AppTokenPayload {
  region: string;
  user_id: string;
```

---

3. LoginServiceType

Represents the structure of a response from a login service.

Property	Type	Description
data	any	The data returned by the login service.
status	number	The HTTP status code of the response.

```
export interface LoginServiceType {
```

```
  data: any;
  status: number;
}
```

---

4. `MigrationQueryType`

Represents the structure of a migration query, typically used for database or organizational migrations.

Property	Type	Description
id	string	The unique identifier for the migration query.
org_id	string	The organization ID associated with the migration.
region	string	The region where the migration is taking place.
owner	string	The owner of the migration query.

```
export interface MigrationQueryType {
  id: string;
  org_id: string;
  region: string;
  owner: string;
}
```

---

5. `Locale`

Represents locale information for internationalization and localization purposes.

Property	Type	Description
code	string	The locale code (e.g., 'en-US').
name	string	The display name of the locale.
fallback_locale	string	The fallback locale code.
uid	string	The unique identifier for the locale.

```
export interface Locale {
  code: string;
  name: string;
  fallback_locale: string;
  uid: string;
}
```

---

src/routes/auth.routes.ts

Authentication Routes (`auth.routes.ts`)

CONTENTSTACK

This module defines the Express router for handling authentication-related endpoints in the application. It provides routes for user login and requesting an SMS token, with request validation and asynchronous error handling.

Overview

- **File:** `src/routes/auth.routes.ts`
- **Purpose: Exposes authentication endpoints for user login and SMS token requests.**
- **Dependencies:**
    - `express`**: For routing.**
    - `authController`**: Contains authentication logic.**
    - `asyncRouter`**: Utility for async error handling.**
    - `validator`**: Middleware for request validation.**

Endpoints

1. User Login

- **Route:** `POST /user-session`
- **Description: Authenticates a user and creates a session.**
- **Request Body:**
    - **Expects user credentials (e.g., username/email and password).**
- **Middleware:**
    - `validator("auth")`**: Validates the request body for required fields and format.**
    - `asyncRouter(authController.login)`**: Handles the login logic asynchronously.**
- **Responses:**
    - **200 OK: Returns user session information (e.g., tokens, user details).**

- **400 Bad Request: If validation fails.**
- **500 Internal Server Error: If an unexpected error occurs.**

Example Request

```
{
  "email": "user@example.com",
  "password": "yourPassword"
}
```

Example Response

```
{
  "token": "jwt-token-string",
  "user": {
    "id": "123",
    "email": "user@example.com"
  }
}
```

---

2. Request SMS Token

- **Route:** `POST /request-token-sms`
- **Description: Requests an SMS token for user authentication (e.g., for 2FA or passwordless login).**
- **Request Body:**
  - **Expects user information (e.g., phone number).**
- **Middleware:**
  - `validator("auth")`**: Validates the request body for required fields and format.**

- `asyncRouter(authController.RequestSms)`: **Handles the SMS token request logic asynchronously.**

- **Responses:**
  - **200 OK: Returns the SMS token or confirmation of SMS sent.**
  - **400 Bad Request: If validation fails.**
  - **500 Internal Server Error: If an unexpected error occurs.**

Example Request

```
POST /request-token-sms
{
  "phone": "+1234567890"
}
```

Example Response

```
{
  "message": "SMS token sent successfully"
}
```

Error Handling

- **ValidationError: Returned if the request body does not meet the required schema.**
- **InternalServerError: Returned if an error occurs during processing.**

contentMapper.routes.ts

This file defines the Express routes for the Content Mapper API. These endpoints allow for managing content types, field mappings, and global fields within a project. The routes are handled by the `contentMapperController` and use the `asyncRouter` utility for error handling.

CONTENTSTACK

The Content Mapper API provides endpoints for:

- **Creating dummy data for development/testing**
- **Listing and managing content types and field mappings**
- **Retrieving and updating global fields**
- **Resetting or updating content type mappings**

All endpoints are prefixed by the router mount path (e.g., `/api/content-mapper`).

Routes

POST `/createDummyData/:projectId`

Description:

Developer endpoint to create dummy data for a given project.

Parameters:

- `projectId` **(string, path): The ID of the project.**

Response:

Creates and returns dummy data for the specified project.

GET `/contentTypes/:projectId/:skip/:limit/:searchText?`

Description:

Get a paginated list of content types for a project, optionally filtered by search text.

Parameters:

- `projectId` **(string, path): The ID of the project.**

- `skip` (number, path): Number of items to skip (for pagination).
- `limit` (number, path): Maximum number of items to return.
- `searchText` (string, path, optional): Text to filter content types.

Response:

Returns a list of content types.

GET `/fieldMapping/:projectId/:contentTypeId/:skip/:limit/:searchText?`

Description:

Get a paginated list of field mappings for a specific content type in a project.

Parameters:

- `projectId` (string, path): The ID of the project.
- `contentTypeId` (string, path): The ID of the content type.
- `skip` (number, path): Number of items to skip.
- `limit` (number, path): Maximum number of items to return.
- `searchText` (string, path, optional): Text to filter field mappings.

Response:

Returns a list of field mappings.

GET `/:projectId/contentTypes/:contentTypeUid?`

Description:

Get a list of existing content types for a project, or a specific content type if `contentTypeUid` is provided.

Parameters:

- `projectId` **(string, path): The ID of the project.**
- `contentTypeUid` **(string, path, optional): The UID of the content type.**

Response:

Returns content type(s) information.

GET `/:projectId/globalFields/:globalFieldUid?`

Description:

Get a list of existing global fields for a project, or a specific global field if `globalFieldUid` is provided.

Parameters:

- `projectId` **(string, path): The ID of the project.**
- `globalFieldUid` **(string, path, optional): The UID of the global field.**

Response:

Returns global field(s) information.

PUT `/contentTypes/:orgId/:projectId/:contentTypeId`

Description:

Update field mapping or content type for a given organization, project, and content type.

Parameters:

- `orgId` **(string, path): The ID of the organization.**

- `projectId` **(string, path): The ID of the project.**
- `contentTypeId` **(string, path): The ID of the content type.**

Request Body:

Field mapping or content type data to update.

Response:

Returns the updated content type or field mapping.

PUT `/resetFields/:orgId/:projectId/:contentTypeId`

Description:

Reset field mapping or content type for a given organization, project, and content type.

Parameters:

- `orgId` **(string, path): The ID of the organization.**
- `projectId` **(string, path): The ID of the project.**
- `contentTypeId` **(string, path): The ID of the content type.**

Response:

Resets and returns the content type or field mapping.

GET `/:orgId/:projectId/content-mapper`

Description:

Remove the content mapper for a given organization and project.

Parameters:

- `orgId` **(string, path): The ID of the organization.**
- `projectId` **(string, path): The ID of the project.**

Response:

Removes and returns the status of the content mapper.

PATCH `/:orgId/:projectId/mapper_keys`

Description:

Update the content mapper keys for a given organization and project.

Parameters:

- `orgId` **(string, path): The ID of the organization.**
- `projectId` **(string, path): The ID of the project.**

Request Body:

Mapper keys data to update.

Response:

Returns the updated mapper keys.

/src/routes/migration.routes.ts

migration.routes.ts

This file defines the Express router for handling migration-related API endpoints. It provides routes for starting and deleting test stacks, creating test stacks, starting migrations, fetching

migration logs, and updating locale mappings. All routes are grouped under the "Migration" category.

Imports

- **express: Web framework for Node.js.**
- **asyncRouter: Utility to wrap route handlers for async error handling.**
- **migrationController: Controller containing migration logic.**

Router Initialization

```
const router = express.Router({ mergeParams: true });
```

Initializes an Express router with merged parameters from parent routers.

Route Definitions

1. Start Test Migration

```
router.post(
  "/test-stack/:orgId/:projectId",
  asyncRouter(migrationController.startTestMigration)
);
```

- **Method: POST**
- **Path: `/test-stack/:orgId/:projectId`**
- **Description: Initiates a test migration for a given organization and project.**
- **Parameters:**
    - `orgId` **(string): Organization ID (URL param)**
    - `projectId` **(string): Project ID (URL param)**

- **Returns: Promise resolving when the test migration is started.**

---

2. Delete Test Stack

```
router.post(
  "/test-stack/:projectId",
  asyncRouter(migrationController.deleteTestStack)
);
```

- **Method: POST**
- **Path:** `/test-stack/:projectId`
- **Description: Deletes a test stack for the specified project.**
- **Parameters:**
    - `projectId` **(string): Project ID (URL param)**
- **Returns: Promise resolving when the test stack is deleted.**

---

3. Create Test Stack

```
router.post(
  "/create-test-stack/:orgId/:projectId",
  asyncRouter(migrationController.createTestStack)
);
```

- **Method: POST**
- **Path:** `/create-test-stack/:orgId/:projectId`
- **Description: Creates a new test stack for the specified organization and project.**

- **Parameters:**
    - `orgId` **(string): Organization ID (URL param)**
    - `projectId` **(string): Project ID (URL param)**
- **Returns: Promise resolving when the test stack is created.**

---

4. Start Final Migration

```
router.post(
  "/start/:orgId/:projectId",
  asyncRouter(migrationController.startMigration)
)
```

- **Method: POST**
- **Path:** `/start/:orgId/:projectId`
- **Description: Starts the final migration for the specified organization and project.**
- **Parameters:**
    - `orgId` **(string): Organization ID (URL param)**
    - `projectId` **(string): Project ID (URL param)**
- **Returns: Promise resolving when the migration is started.**

---

5. Get Migration Logs

```
router.get(
  "/get_migration_logs/:orgId/:projectId/:stackId",
  asyncRouter(migrationController.getLogs)
);
```

- **Method: GET**

- **Path:** `/get_migration_logs/:orgId/:projectId/:stackId`
- **Description: Retrieves migration logs for a specific stack within a project and organization.**
- **Parameters:**
    - `orgId` **(string): Organization ID (URL param)**
    - `projectId` **(string): Project ID (URL param)**
    - `stackId` **(string): Stack ID (URL param)**
- **Returns: Promise resolving with the migration logs.**

---

6. Save Source Locales

```
router.post(
  "/localeMapper/:projectId",
  asyncRouter(migrationController.saveLocales)
);
```

- **Method: POST**
- **Path:** `/localeMapper/:projectId`
- **Description: Updates the source locales fetched from the legacy CMS for the specified project.**
- **Parameters:**
    - `projectId` **(string): Project ID (URL param)**
    - **Body:** `{ locales: Object }` **- Locales to be saved**
- **Returns: Promise resolving when locales are updated in the database.**

---

7. Save Mapped Locales

```
router.post(
```

```
  "/updateLocales/:projectId",
  asyncRouter(migrationController.saveMappedLocales)
);
```

- **Method: POST**
- **Path:** `/updateLocales/:projectId`
- **Description: Updates the mapped locales as provided by the user for the specified project.**
- **Parameters:**
    - `projectId` **(string): Project ID (URL param)**
    - **Body:** `{ locales: Object }` **- Mapped locales to be saved**
- **Returns: Promise resolving when mapped locales are updated in the database.**

Export

```
export default router;
```

Exports the configured router for use in the main application.

`src/routes/org.routes.ts`

org.routes.ts Documentation

This file defines the Express router for handling organization-related routes in the application. It connects HTTP endpoints to controller logic, applies validation, and ensures asynchronous error handling.

Overview

- **File:** `src/routes/org.routes.ts`
- **Purpose: To define and export routes related to organization stacks, locales, and details.**

- **Dependencies:**
    - `express`: **For routing.**
    - `orgController`: **Contains the business logic for each route.**
    - `asyncRouter`: **Utility to handle async errors in route handlers.**
    - `validator`: **Middleware for validating request bodies.**

Route Definitions

1. Get All Stacks

- Endpoint: `GET /stacks/:searchText?`
- **Description: Retrieves all stacks for the organization. Optionally filters stacks by `searchText`.**
- **Parameters:**
    - `searchText` **(optional): String to filter stacks.**
- **Controller:** `orgController.getAllStacks`
- **Example:**
  `GET /stacks`
  `GET /stacks/marketing`

---

2. Create a New Stack

- Endpoint: `POST /stacks`
- **Description: Creates a new stack in the organization.**
- **Validation: Uses `validator("stack")` to validate the request body.**
- **Controller:** `orgController.createStack`
- **Request Body:**
  Must conform to the "stack" schema defined in the validator.

---

3. Get All Locales

- Endpoint: `GET /locales`
- **Description: Retrieves all locales available in Contentstack for the organization.**
- **Controller:** `orgController.getLocales`

---

4. Get Stack Status

- Endpoint: `POST /stack_status`
- **Description: Retrieves the status of a destination stack, including content type counts.**
- **Validation: Uses** `validator("destination_stack")` **to validate the request body.**
- **Controller:** `orgController.getStackStatus`
- **Request Body:**
  Must conform to the "destination_stack" schema defined in the validator.

---

5. Get Stack Locales

- Endpoint: `GET /get_stack_locales`
- **Description: Retrieves all locales for a specific stack.**
- **Controller:** `orgController.getStackLocale`

---

6. Get Organization Details

- Endpoint: `GET /get_org_details`
- **Description: Retrieves details about the organization.**
- **Controller:** `orgController.getOrgDetails`

Middleware

- **asyncRouter: Wraps each controller to handle errors in async functions and pass them to Express error handlers.**
- **validator: Validates request bodies for specific routes to ensure data integrity.**

Projects Routes (`src/routes/projects.routes.ts`)

This module defines the Express router for handling all project-related API endpoints. It connects HTTP routes to controller methods, applies validation where necessary, and ensures asynchronous error handling.

Imports

- **express: Web framework for Node.js.**
- **projectController: Contains all controller methods for project operations.**
- **asyncRouter: Utility to wrap async route handlers for error handling.**
- **validator: Middleware for validating request bodies and parameters.**

Route Definitions

GET `/`

- **Description: Retrieve all projects.**
- **Controller: `projectController.getAllProjects`**
- **Validation: None**

GET `/:projectId`

- **Description: Retrieve a single project by its ID.**
- **Controller: `projectController.getProject`**
- **Validation: None**

POST /

- **Description: Create a new project.**
- **Controller:** `projectController.createProject`
- **Validation: None**

PUT `/:projectId`

- **Description: Update an existing project by its ID.**
- **Controller:** `projectController.updateProject`
- **Validation: None**

PUT `/:projectId/legacy-cms`

- **Description: Update the legacy CMS details for a project.**
- **Controller:** `projectController.updateLegacyCMS`
- **Validation:** `validator("cms")`

PUT `/:projectId/affix`

- **Description: Update the Affix details for a project.**
- **Controller:** `projectController.updateAffix`
- **Validation:** `validator("affix")`

PUT `/:projectId/affix_confirmation`

- **Description: Confirm the Affix update for a project.**
- **Controller:** `projectController.affixConfirmation`
- **Validation:** `validator("affix_confirmation_validator")`

PUT `/:projectId/file-format`

- **Description: Update the file format for a project.**
- **Controller:** `projectController.updateFileFormat`
- **Validation:** `validator("file_format")`

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

PUT `/:projectId/fileformat_confirmation`

- **Description: Confirm the file format update for a project.**
- **Controller:** `projectController.fileformatConfirmation`
- **Validation:** `validator("fileformat_confirmation_validator")`

PUT `/:projectId/destination-stack`

- **Description: Update the destination CMS/stack for a project.**
- **Controller:** `projectController.updateDestinationStack`
- **Validation:** `validator("destination_stack")`

PUT `/:projectId/current-step`

- **Description: Update the current step of a project.**
- **Controller:** `projectController.updateCurrentStep`
- **Validation: None**

DELETE `/:projectId`

- **Description: Delete a project by its ID.**
- **Controller:** `projectController.deleteProject`
- **Validation: None**

PATCH `/:projectId`

- **Description: Revert a project to a previous state.**
- **Controller:** `projectController.revertProject`
- **Validation: None**

PATCH `/:projectId/stack-details`

- **Description: Update stack details for a project.**
- **Controller:** `projectController.updateStackDetails`
- **Validation: None**

PUT `/:projectId/migration-excution`

- **Description: Update the migration execution key for a project.**
- **Controller: `projectController.updateMigrationExecution`**
- **Validation: None**

GET `/:projectId/get-migrated-stacks`

- **Description: Retrieve migrated stacks for a project.**
- **Controller: `projectController.getMigratedStacks`**
- **Validation: None**

Middleware

- **asyncRouter: Wraps each controller to handle errors in async functions.**
- **validator: Applies request validation for specific routes.**

User Routes (`src/routes/user.routes.ts`)

This file defines the Express routes related to user operations, specifically the user profile endpoint. It imports the necessary controller and utility for handling asynchronous route logic.

Imports

- **express: The Express framework for building web applications.**
- **userController: Contains the business logic for user-related operations.**
- **asyncRouter: Utility to wrap route handlers for proper async error handling.**

Route Definitions

GET **`/profile`**

Description:

- **Retrieves the profile information of the currently authenticated user.**

Handler:

- `userController.getUserProfile` **(wrapped with** `asyncRouter` **for async error handling)**
- **Request:**
  - **Method: GET**
  - **Endpoint:** `/profile`
  - **Authentication: Typically, this route should be protected and require user authentication (not shown in this snippet).**
- **Response:**
  - **200 OK: Returns the user's profile data in JSON format.**
  - **4xx/5xx: Returns an error object if the request fails or the user is not authenticated.**

src/services/contentful/jsonRTE.ts

jsonRTE.ts

This module provides utilities to parse Contentful Rich Text Editor (RTE) JSON structures into a custom intermediate format suitable for further processing or migration. It supports a wide range of node types, including text, headings, lists, tables, references, and assets, and is designed to be extensible and locale-aware.

Overview

The module is designed to convert Contentful RTE JSON nodes into a custom format, handling nested structures, locale-specific references, and asset lookups. It is used in migration and transformation pipelines where Contentful content needs to be adapted to another system or schema.

Configuration and Imports

```
import path from 'path';

import fs from 'fs';

import { MIGRATION_DATA_CONFIG } from '../../constants/index.js';
```

- **MIGRATION_DATA_CONFIG: Contains directory and file names for data, locales, references, and assets used during migration.**

Types

```
type NodeType = string;

type LangType = string;

type StackId = string;
```

- **NodeType: The type of node in the Contentful RTE JSON (e.g., 'paragraph', 'heading-1').**
- **LangType: The locale/language code (e.g., 'en-us').**
- **StackId: The identifier for the destination stack/environment.**

File Reading Utility

```
function readFile(filePath: string) {

  if (fs.existsSync(filePath)) {

    return JSON.parse(fs.readFileSync(filePath, 'utf-8'));

  }

  return undefined;

}
```

- **Reads and parses a JSON file if it exists, otherwise returns `undefined`.**
- **Used for loading reference and asset mappings.**

Parser Map

```
const parsers: Map<NodeType, (obj: any, lang?: LangType,
destination_stack_id?: StackId) => any> = new Map([

  // ...nodeType to parser function mappings...

]);
```

- **Maps each supported Contentful node type to its corresponding parser function.**
- **Enables dynamic dispatch based on node type.**

Main Entry Point

```
export default function jsonParse(obj: { nodeType: NodeType }, lang?:
LangType, destination_stack_id?: StackId) {

  const parser = parsers.get(obj.nodeType);

  if (parser) {

    return parser(obj, lang, destination_stack_id);

  }

  return null;

}
```

- **jsonParse: The main function to parse a Contentful RTE node.**
  - `obj`**: The node object (must have a** `nodeType` **property).**
  - `lang`**: Optional locale code.**
  - `destination_stack_id`**: Optional stack/environment ID.**
- **Returns the parsed node in the custom format, or** `null` **if the node type is unsupported.**

UID Generation

```
function generateUID(prefix: string): string {
```

```
  return `${prefix}${Math.floor(Math.random() * 100000000000000)}`;
}
```

- **Generates a pseudo-unique identifier for each node, prefixed by the node type.**

Node Parsers

Document and Paragraphs

- parseDocument: Parses the root document node, recursively parsing its children.
- parseParagraph: Parses a paragraph node and its children.

Text and Marks

- parseText: Parses a text node, applying any marks (e.g., bold, italic, code).

Lists and List Items

- parseUL: Parses an unordered list.
- parseOL: Parses an ordered list.
- parseLI: Parses a list item.

Headings

- parseHeading1 to parseHeading6: Parse heading nodes of levels 1 to 6.

Blockquote and Horizontal Rule

- parseBlockquote: Parses a blockquote node.
- parseHR: Parses a horizontal rule node.

Tables

- parseTable: Parses a table node, including rows and cells.
- parseTableRow, parseHeadTR, parseTableHead, parseTBody, parseBodyTR, parseTableBody: Handle various table substructures.

References and Assets

- parseBlockReference: Parses a block reference to another entry, using locale and stack mappings.
- parseInlineReference: Parses an inline reference to another entry.
- parseBlockAsset: Parses a block asset reference, resolving asset details from the stack.

Hyperlinks

- parseEntryHyperlink: Parses a hyperlink to another entry.
- parseAssetHyperlink: Parses a hyperlink to an asset.
- parseHyperlink: Parses a generic hyperlink.

Extending the Parser

To add support for a new node type:

1. **Implement a new parser function with the signature `(obj, lang?, destination_stack_id?) => any`.**
2. **Add the function to the `parsers` map with the appropriate node type key.**

Example Usage

```
import jsonParse from './jsonRTE';


const contentfulNode = { nodeType: 'paragraph', content: [/* ... */] };

const parsed = jsonParse(contentfulNode, 'en-us', 'my-stack-id');

console.log(parsed);
```

Notes

- **The module expects certain directory and file structures for references and assets, as defined in `MIGRATION_DATA_CONFIG`.**

- **Locale and stack awareness is built into reference and asset resolution.**
- **The output format is designed for compatibility with downstream systems that require a normalized, UID-based node structure.**

Auth Service

This module provides authentication-related services, including user login and requesting an SMS login token. It interacts with external APIs, manages authentication data, and handles error logging and reporting.

Dependencies

- **express.Request: For handling HTTP request data.**
- **config: Application configuration, including API endpoints.**
- **safePromise, getLogMessage: Utility functions for error handling and logging.**
- **https: Utility for making HTTP requests.**
- **LoginServiceType, AppTokenPayload: Type definitions for service responses and JWT payloads.**
- **HTTP_CODES, HTTP_TEXTS: Constants for HTTP status codes and messages.**
- **generateToken: Utility for generating JWT tokens.**
- **BadRequestError, InternalServerError, ExceptionFunction: Custom error classes.**
- **AuthenticationModel: Model for storing authentication data.**
- **logger: Logging utility.**

Functions

`login(req: Request): Promise<LoginServiceType>`

Logs in a user using the provided credentials and region. Handles authentication with the external API, validates user roles, updates the authentication model, and generates a JWT token.

Parameters

- **req (`Request`): The Express request object containing user credentials (`email`, `password`, optional `tfa_token`, and `region`).**

Returns

- **Promise: Resolves with an object containing either the login result or error details.**

Throws

- **ExceptionFunction: If an error occurs during the login process, such as missing admin roles or user data.**

Process

1. **Extracts user data from the request body.**
2. **Sends a POST request to the external API for user authentication.**
3. **Handles API errors and logs them.**
4. **Checks if the user has admin roles in any organization; throws an error if not.**
5. **Updates or inserts the user's authentication data in the local model.**
6. **Generates a JWT token for the authenticated user.**
7. **Returns a success response with the token, or an error response if applicable.**

Example

```
const response = await authService.login(req);

if (response.status === 200) {
  // Successful login
} else {
  // Handle error
}
```

```
requestSms(req: Request): Promise<LoginServiceType>
```

Requests an SMS login token for the user by sending their credentials to the external API.

Parameters

- **req (`Request`): The Express request object containing user credentials (`email`, `password`, and `region`).**

Returns

- **Promise: Resolves with the API response or error details.**

Throws

- **InternalServerError: If an error occurs while sending the request.**

Process

1. **Extracts user data from the request body.**
2. **Sends a POST request to the external API endpoint for requesting an SMS token.**
3. **Handles API errors and logs them.**
4. **Returns the API response or error details.**

Example

```
const response = await authService.requestSms(req);
if (response.status === 200) {
  // SMS token sent
} else {
  // Handle error
}
```

Exported Object

```
export const authService = {
  login,
```

```
  requestSms,
};
```

- **login: Function to authenticate a user and generate a JWT token.**
- **requestSms: Function to request an SMS login token.**

Error Handling

- **All errors are logged using the `logger` utility.**
- **Custom error classes (`BadRequestError`, `InternalServerError`, `ExceptionFunction`) are used to provide meaningful error messages and status codes.**

src/services/contentful.service.ts

Contentful Service

This module provides services for interacting with the Contentful CMS API. It includes functions for fetching, creating, updating, and deleting content entries, as well as handling API authentication and error management. The service abstracts the details of Contentful API requests, manages configuration, and ensures consistent error handling and logging.

Dependencies

- **express.Request: For handling HTTP request data.**
- **config: Application configuration, including Contentful API endpoints and credentials.**
- **safePromise, getLogMessage: Utility functions for error handling and logging.**
- **https: Utility for making HTTP requests.**
- **ContentfulServiceType, ContentfulEntryPayload: Type definitions for service responses and Contentful entry payloads.**
- **HTTP_CODES, HTTP_TEXTS: Constants for HTTP status codes and messages.**

- **BadRequestError, InternalServerError, ExceptionFunction: Custom error classes.**
- **logger: Logging utility.**

Functions

getEntries(req: Request): Promise

Fetches a list of entries from Contentful based on the provided query parameters.

Parameters

- `req (Request)`**: The Express request object containing query parameters (e.g., content type, filters).**

Returns

- `Promise<ContentfulServiceType>`**: Resolves with an object containing the list of entries or error details.**

Throws

- `ExceptionFunction`**: If an error occurs during the fetch process.**

Process

- **Extracts query parameters from the request.**
- **Sends a GET request to the Contentful API to fetch entries.**
- **Handles API errors and logs them.**
- **Returns the list of entries or error details.**

Example

```
const response = await contentfulService.getEntries(req);
if (response.status === 200) {
  // Entries fetched successfully
} else {
  // Handle error
```

```
}
```

---

createEntry(req: Request): Promise

Creates a new entry in Contentful using the provided data.

Parameters

- `req (Request)`: **The Express request object containing entry data in the body.**

Returns

- `Promise<ContentfulServiceType>`: **Resolves with the created entry or error details.**

Throws

- `BadRequestError`: **If required data is missing or invalid.**
- `InternalServerError`: **If an error occurs during the creation process.**

Process

- **Extracts entry data from the request body.**
- **Sends a POST request to the Contentful API to create the entry.**
- **Handles API errors and logs them.**
- **Returns the created entry or error details.**

Example

```
const response = await contentfulService.createEntry(req);
if (response.status === 201) {
  // Entry created successfully
} else {
  // Handle error
}
```

---

updateEntry(req: Request): Promise

Updates an existing Contentful entry with the provided data.

Parameters

- `req (Request)`: **The Express request object containing entry ID and updated data.**

Returns

- `Promise<ContentfulServiceType>`: **Resolves with the updated entry or error details.**

Throws

- `BadRequestError`: **If required data is missing or invalid.**
- `InternalServerError`: **If an error occurs during the update process.**

Process

- **Extracts entry ID and update data from the request.**
- **Sends a PUT/PATCH request to the Contentful API to update the entry.**
- **Handles API errors and logs them.**
- **Returns the updated entry or error details.**

Example

```
const response = await contentfulService.updateEntry(req);
if (response.status === 200) {
  // Entry updated successfully
} else {
  // Handle error
}
```

---

deleteEntry(req: Request): Promise

Deletes an entry from Contentful by ID.

Parameters

- `req (Request)`: **The Express request object containing the entry ID.**

Returns

- `Promise<ContentfulServiceType>`: **Resolves with a success message or error details.**

Throws

- `BadRequestError`: **If the entry ID is missing or invalid.**
- `InternalServerError`: **If an error occurs during the deletion process.**

Process

- **Extracts entry ID from the request.**
- **Sends a DELETE request to the Contentful API.**
- **Handles API errors and logs them.**
- **Returns a success message or error details.**

Example

```
const response = await contentfulService.deleteEntry(req);
if (response.status === 200) {
  // Entry deleted successfully
} else {
  // Handle error
}
```

Exported Object

```
export const contentfulService = {
```

```
  getEntries,
  createEntry,
  updateEntry,
  deleteEntry,
};
```

- `getEntries`: Function to fetch entries from Contentful.
- `createEntry`: Function to create a new entry.
- `updateEntry`: Function to update an existing entry.
- `deleteEntry`: Function to delete an entry.

Error Handling

- All errors are logged using the `logger` utility.
- Custom error classes (`BadRequestError`, `InternalServerError`, `ExceptionFunction`) are used to provide meaningful error messages and status codes.

contentMapperService

This service provides a set of functions to manage content type and field mappings for projects, including CRUD operations, reset, and integration with external APIs (such as Contentstack). It is designed to support content migration, mapping, and validation workflows in a multi-project environment.

Overview

The `contentMapperService` is responsible for:

- Creating, updating, and deleting content type and field mappings for projects.
- Fetching and searching content types and field mappings.
- Integrating with Contentstack APIs to fetch content types and global fields.

- **Resetting mappings to their initial state.**
- **Handling error cases and logging for traceability.**

All functions are asynchronous and expect an Express `Request` object or project ID as input.

Service Functions

putTestData

Description:

Creates or updates dummy content mapping data for a given project. It processes the provided content types, assigns unique IDs, and updates the project's mapping references.

Parameters:

- `req: Request` **– Express request containing** `projectId` **(params) and** `contentTypes` **(body).**

Returns:

- **Updated project data.**

Throws:

- `BadRequestError` **if the project or content types are not found.**
- `ExceptionFunction` **for internal errors.**

---

getContentTypes

Description:

Retrieves content types for a project, with support for pagination and search.

Parameters:

- `req: Request` – Express request with `projectId`, `skip`, `limit`, and optional `searchText` (params).

Returns:

- Object with `status`, `count`, and `contentTypes` array.

Throws:

- `BadRequestError` if the project is not found.
- `ExceptionFunction` for internal errors.

---

getFieldMapping

Description:

Fetches field mappings for a specific content type, with pagination and search support.

Parameters:

- `req: Request` – Express request with `contentTypeId`, `projectId`, `skip`, `limit`, and optional `searchText` (params).

Returns:

- Object with `status`, `count`, and `fieldMapping` array.

Throws:

- `BadRequestError` if the content type is not found.
- `ExceptionFunction` for internal errors.

---

getExistingContentTypes

Description:

Fetches all content types from Contentstack for a given project, and optionally details for a specific content type.

Parameters:

- `req: Request` – **Express request with `projectId`, optional `contentTypeUid` (params), and `token_payload` (body).**

Returns:

- **Object with `contentTypes` array and optional `selectedContentType`.**

---

getExistingGlobalFields

Description:

Fetches all global fields from Contentstack for a given project, and optionally details for a specific global field.

Parameters:

- `req: Request` – **Express request with `projectId`, optional `globalFieldUid` (params), and `token_payload` (body).**

Returns:

- **Object with `globalFields` array and optional `selectedGlobalField`.**

---

updateContentType

Description:

Updates a content type and its field mappings for a project. Validates mapping data and updates the status accordingly.

Parameters:

- `req: Request` – Express request with `orgId`, `projectId`, `contentTypeId` (params), and `contentTypeData`, `token_payload` (body).

Returns:

- Object with `status` and updated content type data.

Throws:

- Returns error object if validation fails or update is not allowed.

---

resetToInitialMapping

Description:

Resets the field and content mapping for a specific content type in a project to its initial state.

Parameters:

- `req: Request` – Express request with `orgId`, `projectId`, `contentTypeId` (params), and `token_payload` (body).

Returns:

- Object with `status`, `message`, and reset data.

Throws:

- `BadRequestError` if the project or content type is not found or not in a valid state.
- `ExceptionFunction` for internal errors.

---

resetAllContentTypesMapping

Description:

Resets all content type mappings for a project to their initial state.

Parameters:

- `projectId: string` – **Project ID.**

Returns:

- **Project details after reset.**

Throws:

- `BadRequestError` **if the project or content mapper is not found.**
- `ExceptionFunction` **for internal errors.**

---

removeMapping

Description:

Removes all content and field mappings for a project.

Parameters:

- `projectId: string` – **Project ID.**

Returns:

- **Project details after removal.**

Throws:

- `BadRequestError` **if the project is not found.**
- `ExceptionFunction` **for internal errors.**

---

removeContentMapper

Description:

Removes all content and field mappings for a project (alternative entry point, expects `Request`**).**

Parameters:

- `req: Request` – Express request with `projectId` (params).

Returns:

- Project details after removal.

Throws:

- `BadRequestError` if the project is not found.
- `ExceptionFunction` for internal errors.

---

getSingleContentTypes

Description:

Fetches a single content type from Contentstack for a project.

Parameters:

- `req: Request` – Express request with `projectId`, `contentTypeUid` (params), and `token_payload` (body).

Returns:

- Object with `title`, `uid`, and `schema` of the content type, or error object.

---

getSingleGlobalField

Description:

Fetches a single global field from Contentstack for a project.

Parameters:

- `req: Request` – Express request with `projectId`, `globalFieldUid` (params), and `token_payload` (body).

Returns:

- **Object with `title`, `uid`, and `schema` of the global field, or error object.**

---

updateContentMapper

Description:

Updates the content mapper details for a project.

Parameters:

- **`req: Request` – Express request with `orgId`, `projectId` (params), and `token_payload`, `content_mapper` (body).**

Returns:

- **Object with `status` and update message.**

Throws:

- **`ExceptionFunction` for internal errors.**

Usage Example

```
import { contentMapperService } from './services/contentMapper.service';


// Example: Update content type mapping

const result = await contentMapperService.updateContentType(req);
```

Error Handling

All functions throw or return structured error objects using custom error classes (`BadRequestError`, `ExceptionFunction`). Logging is performed for traceability.

---

extension.service.ts

This service provides utility functions for managing extension data during migration processes. It handles reading, generating, and writing extension configuration files for a given destination stack.

Dependencies

- **path: Node.js module for handling file and directory paths.**
- **fs: Node.js file system module for reading and writing files.**
- **MIGRATION_DATA_CONFIG, LIST_EXTENSION_UID: Constants used for configuration and logic branching.**

Constants

- **CUSTOM_MAPPER_FILE_NAME: Name of the custom mapper file (from config).**
- **EXTENSION_APPS_DIR_NAME: Directory name for extension apps (from config).**
- **EXTENSION_APPS_FILE_NAME: File name for extension apps (from config).**

Functions

`writeExtFile({ destinationStackId, extensionData })`

Description:

Writes the provided extension data to a JSON file in the appropriate directory for the given destination stack. If the directory does not exist, it is created recursively.

Parameters:

- `destinationStackId` **(string): The ID of the destination stack.**
- `extensionData` **(object): The extension data to be written.**

Behavior:

- **Ensures the target directory exists (creates it if not).**
- **Writes the extension data as a formatted JSON file.**
- **Logs errors to the console if directory creation or file writing fails.**

`getExtension({ uid, destinationStackId })`

Description:

Retrieves extension metadata for a given UID and destination stack. If the UID matches `LIST_EXTENSION_UID`, returns a hardcoded extension object; otherwise, returns `null`.

Parameters:

- `uid` **(string): The extension UID.**
- `destinationStackId` **(string): The ID of the destination stack.**

Returns:

- **An extension object if the UID matches `LIST_EXTENSION_UID`.**
- `null` **otherwise.**

`createExtension({ destinationStackId })`

Description:

Reads the custom mapper file for the given destination stack, extracts unique extension UIDs, retrieves their metadata, and writes the combined extension data to a file.

Parameters:

- `destinationStackId` **(string): The ID of the destination stack.**

Behavior:

- **Reads the custom mapper file (if it exists).**
- **Parses the file to extract unique extension UIDs.**

- For each UID, retrieves extension metadata using `getExtension`.
- Aggregates all extension data and writes it using `writeExtFile`.

Exported Object

`extensionService`

Description:

Exports the main service function(s) for use in other modules.

Properties:

- `createExtension`: **The function to generate and write extension data for a stack.**

Usage Example:

```
import { extensionService } from './services/extension.service';


await extensionService.createExtension({ destinationStackId:
'your_stack_id' });
```

Error Handling

- **All file system operations are wrapped in try/catch blocks.**
- **Errors during directory creation or file writing are logged to the console.**
- **If the custom mapper file does not exist, the process is silently skipped.**

Notes

- **The extension metadata for `LIST_EXTENSION_UID` is hardcoded and includes a sample HTML/Angular-based UI extension.**

- **The service assumes a specific directory structure and naming convention as defined in `MIGRATION_DATA_CONFIG`.**
- **All file operations are asynchronous and use Promises.**

marketplace.service.ts

This service provides utilities for managing marketplace app manifests during migration processes. It handles grouping extensions by app, removing sensitive keys, fetching app manifests, and writing the final manifest file for a destination stack.

Imports

- **path: Node.js module for handling file paths.**
- **fs: Node.js module for file system operations.**
- **getAuthtoken: Utility to fetch authentication tokens.**
- **MIGRATION_DATA_CONFIG, KEYTOREMOVE: Constants for migration configuration and keys to remove from objects.**
- **getAppManifestAndAppConfig: Utility to fetch app manifest and configuration from the marketplace.**
- **uuidv4: Utility to generate unique identifiers.**

Constants

- **EXTENSIONS_MAPPER_DIR_NAME, MARKETPLACE_APPS_DIR_NAME, MARKETPLACE_APPS_FILE_NAME: Directory and file names used for storing migration data, sourced from `MIGRATION_DATA_CONFIG`.**

Helper Functions

`groupByAppUid(data: any): object`

Groups extension UIDs by their associated app UID.

- **Parameters:**

    - `data`: Array of extension mapping objects, each containing `appUid` and `extensionUid`.

- **Returns:**

    - An object where each key is an `appUid` and the value is an array of associated `extensionUid`s.

`removeKeys(obj: object, keysToRemove: string[]): object`

Removes specified keys from an object.

- **Parameters:**

    - `obj`: The source object.
    - `keysToRemove`: Array of keys to remove from the object.

- **Returns:**

    - A new object with the specified keys removed.

`writeManifestFile({ destinationStackId, appManifest })`

Writes the app manifest array to a JSON file in the appropriate directory for the destination stack.

- **Parameters:**

    - `destinationStackId`: The target stack's unique identifier.
    - `appManifest`: Array of app manifest objects to write.

- **Behavior:**

    - Ensures the directory exists (creates it if not).
    - Writes the manifest as a formatted JSON file.

Main Function

`createAppManifest({ destinationStackId, region, userId, orgId })`

Generates and writes a marketplace app manifest for a given destination stack.

- **Parameters:**
    1. `destinationStackId`: **Target stack UID.**
    2. `region`: **API region.**
    3. `userId`: **User UID for authentication.**
    4. `orgId`: **Organization UID.**
- **Process:**
    1. **Fetches an authentication token.**
    2. **Reads the extension-to-app mapping file for the destination stack.**
    3. **Groups extensions by app UID.**
    4. **For each app:**
        - **Fetches the app manifest and configuration.**
        - **Removes sensitive keys.**
        - **Maps extension UIDs to their UI locations.**
        - **Adds a config location if present.**
        - **Sets status and target information.**
        - **Removes sensitive keys again and adds to the manifest array.**
    5. **Writes the final manifest array to the stack's manifest file.**

Exported Service

```
export const marketPlaceAppService = {
  createAppManifest
}
```

CONTENTSTACK

Example Usage

```
import { marketPlaceAppService } from './services/marketplace.service';


await marketPlaceAppService.createAppManifest({
  destinationStackId: 'your-stack-id',
  region: 'us',
  userId: 'user-uid',
  orgId: 'org-uid'
});
```

Notes

- **The service expects the extension mapping file to be present and formatted correctly.**
- **Sensitive or unnecessary keys are removed from manifest objects using the `KEYTOREMOVE` constant.**
- **The manifest file is written as a pretty-printed JSON for readability.**
- **Error handling is present for file system operations, but you may want to enhance it for production use.**

Migration Service

This service provides core migration-related operations for managing test and production stacks, handling migrations from legacy CMSs, and managing project locale data. It interacts with various utility and service modules to orchestrate stack creation, deletion, migration, and logging.

`createTestStack`

Creates a new test stack for a given project and organization.

Parameters:

- `req: Request`
  Express request object containing:
  - `params.orgId`: **Organization ID**
  - `params.projectId`: **Project ID**
  - `body.name`: **Name for the stack**
  - `body.token_payload`: **Auth token payload (region, user_id, etc.)**

Returns:

`Promise<LoginServiceType>`

- **On success: Object with stack data and dashboard URL.**
- **On failure: Object with error data and status.**

Throws:

- `ExceptionFunction` **if stack creation fails due to API or internal errors.**

Side Effects:

- **Updates the project in the database with the new test stack and step.**

`deleteTestStack`

Deletes a test stack associated with a project.

Parameters:

- `req: Request`
  Express request object containing:
  - `params.projectId`: **Project ID**
  - `body.token_payload`: **Auth token payload**
  - `body.stack_key`: **API key of the stack to delete**

Returns:

`Promise<LoginServiceType>`

- **On success: Object with deletion response data.**
- **On failure: Object with error data and status.**

Throws:

- `ExceptionFunction` **if deletion fails due to API or internal errors.**

Side Effects:

- **Updates the project in the database to remove the test stack.**

`startTestMigration`

Initiates a test migration for a project's test stack.

Executes migration steps based on the legacy CMS type (Sitecore, WordPress, Contentful).

Parameters:

- `req: Request`
  Express request object containing:
  - `params.orgId`**: Organization ID**
  - `params.projectId`**: Project ID**
  - `body.token_payload`**: Auth token payload**

Returns:

`Promise<any>`

Throws:

- **Propagates errors from underlying migration steps.**

Side Effects:

- **Writes migration logs.**
- **Updates stack with migrated content, locales, assets, and extensions.**

`startMigration`

Initiates the final migration for a project's production stack.

Executes migration steps based on the legacy CMS type (Sitecore, WordPress, Contentful).

Parameters:

- `req: Request`
  Express request object containing:
  - `params.orgId`: **Organization ID**
  - `params.projectId`: **Project ID**
  - `body.token_payload`: **Auth token payload**

Returns:

`Promise<any>`

Throws:

- **Propagates errors from underlying migration steps.**

Side Effects:

- **Marks migration as started in the project.**
- **Writes migration logs.**
- **Updates stack with migrated content, locales, assets, and extensions.**

`getLogs`

Retrieves and parses migration logs for a given project and stack.

Parameters:

- `req: Request`
  Express request object containing:
  - `params.projectId`: **Project ID**
  - `params.stackId`: **Stack ID**

Returns:

`Promise<any[]>`

- **Array of parsed log entries.**

Throws:

- `BadRequestError` **if the projectId or stackId is invalid or logs are not found.**
- `ExceptionFunction` **for internal errors.**

`createSourceLocales`

Stores or updates the source locales fetched from the legacy CMS in the project database.

Parameters:

- `req: Request`
  Express request object containing:
  - `params.projectId`: **Project ID**
  - `body.locale`: **Array of locale codes**

Returns:

`Promise<void>`

Throws:

- `ExceptionFunction` if the project ID is invalid or DB update fails.

Side Effects:

- Updates the `source_locales` field in the project.

`updateLocaleMapper`

Updates the mapped locales and master locale in the project database.

Parameters:

- `req: Request`
  Express request object containing:
  - `params.projectId`: Project ID
  - `body`: Object with `master_locale` and `locales` mapping

Returns:

`Promise<void>`

Throws:

- `ExceptionFunction` if the project ID is invalid or DB update fails.

Side Effects:

- Updates the `master_locale` and `locales` fields in the project.

Exported Service

```
export const migrationService = {
  createTestStack,
  deleteTestStack,
  startTestMigration,
  startMigration,
```

```
  getLogs,
  createSourceLocales,
  updateLocaleMapper,
};
```

Error Handling

All functions log errors using the logger utility and throw custom exceptions (`ExceptionFunction` or `BadRequestError`) with appropriate status codes and messages.

Dependencies

- **Project database (Lowdb)**
- **Various CMS-specific services (Sitecore, WordPress, Contentful)**
- **Utility modules for logging, authentication, HTTP requests, and file operations**

org.service.ts

This service provides organization-related operations for interacting with the Contentstack API, including stack management, locale retrieval, and organization details. It handles authentication, error logging, and response formatting for each operation.

Dependencies

- **express.Request: For handling HTTP requests.**
- **config: Application configuration, including Contentstack API endpoints.**
- **safePromise, getLogMessage: Utility functions for error handling and logging.**
- **https: HTTP client utility for making API requests.**
- **LoginServiceType: Type definition for service responses.**
- **getAuthtoken: Utility for retrieving authentication tokens.**
- **logger: Logging utility.**
- **HTTP_TEXTS, HTTP_CODES: Constants for HTTP status codes and messages.**

- **ExceptionFunction, BadRequestError: Custom error classes.**
- **ProjectModelLowdb: Local database model for project and stack filtering.**

Functions

getAllStacks

Retrieves all stacks for a given organization, with optional search and filtering based on local project data.

Parameters:

- `req: Request` **– Express request object, expects** `orgId` **and optional** `searchText` **in params, and** `token_payload` **in body.**

Returns:

- `Promise<LoginServiceType>` **– Object containing stack data and HTTP status.**

Behavior:

- **Authenticates the user and fetches stacks from Contentstack.**
- **Optionally filters stacks by search text (name/description).**
- **Further filters out stacks already present in local project data.**
- **Handles and logs errors, returning error data and status if the API call fails.**

---

createStack

Creates a new stack in the specified organization.

Parameters:

- `req: Request` **– Express request object, expects** `orgId` **in params, and** `token_payload`, `name`, `description`, `master_locale` **in body.**

Returns:

- `Promise<LoginServiceType>` – Object containing the created stack data and HTTP status.

Behavior:

- Authenticates the user and sends a POST request to create a stack.
- Handles and logs errors, returning error data and status if the API call fails.

---

getLocales

Retrieves all locales available in the Contentstack region.

Parameters:

- `req: Request` – Express request object, expects `token_payload` in body.

Returns:

- `Promise<LoginServiceType>` – Object containing locale data and HTTP status.

Behavior:

- Authenticates the user and fetches locales from Contentstack.
- Handles and logs errors, returning error data and status if the API call fails.

---

getStackStatus

Checks the status of a specific stack by verifying its existence and retrieving a count of its content types.

Parameters:

- `req: Request` – Express request object, expects `orgId` in params, and `token_payload`, `stack_api_key` in body.

Returns:

- `Promise<{ status: number, data: any }>` – Object containing the status and content type count.

Behavior:

- Authenticates the user and verifies the stack exists in the organization.
- Fetches content type count for the stack.
- Handles and logs errors, returning error data and status if the API call fails or stack is not found.

---

getStackLocale

Retrieves all locales for a specific stack.

Parameters:

- `req: Request` – Express request object, expects `token_payload`, `stack_api_key` in body.

Returns:

- `Promise<{ status: number, data: any }>` – Object containing the status and locale data.

Behavior:

- Authenticates the user and fetches locales for the specified stack.
- Handles and logs errors, returning error data and status if the API call fails.

---

getOrgDetails

Retrieves details and plan information for a specific organization.

Parameters:

- `req: Request` – Express request object, expects `orgId` in params, and `token_payload` in body.

Returns:

- `Promise<{ status: number, data: any }>` – **Object containing the status and organization details.**

Behavior:

- **Authenticates the user and fetches organization details from Contentstack.**
- **Handles and logs errors, returning error data and status if the API call fails.**

Error Handling

- **All functions use `safePromise` to handle async errors gracefully.**
- **Errors are logged using the `logger` utility with contextual information.**
- **API errors are returned with their status and message.**
- **Unhandled errors throw an `ExceptionFunction` with a generic or specific error message and status code.**

Export

```
export const orgService = {
  getAllStacks,
  getLocales,
  createStack,
  getStackStatus,
  getStackLocale,
  getOrgDetails,
};
```

Usage Example

```
import { orgService } from './services/org.service';


// Example: Get all stacks for an organization

app.get('/org/:orgId/stacks', async (req, res) => {

  const result = await orgService.getAllStacks(req);

  res.status(result.status).json(result.data);

});
```

Project Service (`projects.service.ts`)

This service provides a set of functions to manage project lifecycle operations in a multi-tenant, region-aware environment. It interacts with a Lowdb-based data store and supports CRUD operations, stepper-based workflow progression, and related project utilities.

Overview

The Project Service is responsible for all business logic related to project management, including creation, retrieval, update, deletion, and workflow progression. It ensures that all operations are performed in the context of the authenticated user, organization, and region, and enforces stepper-based workflow rules.

Service Functions

getAllProjects

```
getAllProjects(req: Request): Promise<Project[]>
```

- **Retrieves all non-deleted projects for the given organization, region, and user.**
- **Throws `NotFoundError` if no projects are found.**

getProject

```
getProject(req: Request): Promise<Project>
```

- Retrieves a single project by `orgId`, `projectId`, region, and owner.
- Throws `NotFoundError` if the project is not found.

createProject

```
createProject(req: Request): Promise<{ status: string, message: string, project: Partial<Project> }>
```

- Creates a new project with initial stepper and status values.
- Returns a summary of the created project.
- Throws `ExceptionFunction` on error.

updateProject

```
updateProject(req: Request): Promise<{ status: string, message: string, project: Partial<Project> }>
```

- Updates project fields such as name, description, stack details, and mapper keys.
- Throws `ExceptionFunction` on error.

updateLegacyCMS

```
updateLegacyCMS(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Updates the legacy CMS configuration for a project.
- Throws `BadRequestError` if the project is in a non-editable state.
- Throws `ExceptionFunction` on error.

updateAffix

```
updateAffix(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Updates the `affix` property in the project's legacy CMS section.

affixConfirmation

```
affixConfirmation(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Updates the `affix_confirmation` property in the project's legacy CMS section.

updateFileFormat

```
updateFileFormat(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Updates file format and related AWS details for the project's legacy CMS.
- Throws `BadRequestError` if the project is in a non-editable state.
- Throws `ExceptionFunction` on error.

fileformatConfirmation

```
fileformatConfirmation(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Updates the `file_format_confirmation` property in the project's legacy CMS section.

updateDestinationStack

```
updateDestinationStack(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Updates the destination stack for a project after validating the stack exists via an external API.
- Throws `BadRequestError` if the project is in a non-editable state or stack is not found.
- Throws `ExceptionFunction` on error.

updateCurrentStep

```
updateCurrentStep(req: Request): Promise<Project>
```

- Progresses the project to the next step in the workflow, enforcing stepper and status rules.
- Throws `BadRequestError` if the current step cannot be updated.
- Throws `ExceptionFunction` on error.

deleteProject

```
deleteProject(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Soft-deletes a project by setting `isDeleted` to `true`, or hard-deletes if the project is completed.
- Also deletes related content mappers and field mappings if applicable.
- Throws `NotFoundError` if the project is not found.

revertProject

```
revertProject(req: Request): Promise<{ status: number, data: { message: string, Project: Project } }>
```

- Reverts a soft-deleted project by setting `isDeleted` to `false`.
- Throws `NotFoundError` if the project is not found.

updateStackDetails

```
updateStackDetails(req: Request): Promise<{ status: number, data: { message: string } }>
```

- Updates the stack details for a project.
- Throws `ExceptionFunction` on error.

updateContentMapper

```
updateContentMapper(req: Request): Promise<{ status: number, data: { message: string } }>
```

- **Updates the content mapper details for a project.**
- **Throws `ExceptionFunction` on error.**

updateMigrationExecution

```
updateMigrationExecution(req: Request): Promise<{ status: number, data: { message: string } }>
```

- **Sets the `migration_execution` flag to `true` for a project.**
- **Throws `ExceptionFunction` on error.**

getMigratedStacks

```
getMigratedStacks(req: Request): Promise<{ status: number, destinationStacks: string[] }>
```

- **Returns the `destination_stack_id` of all completed projects (status and step both at 5).**
- **Throws `ExceptionFunction` on error.**

Error Handling

- **All functions throw custom errors (`BadRequestError`, `NotFoundError`, `ExceptionFunction`) for consistent error handling.**
- **Errors are logged with context for easier debugging.**

Logging

- **All major operations are logged using the `logger` utility, including both successes and failures, with contextual information such as function name, project ID, and user details.**

Exports

The service exports all functions as a single object:

```
export const projectService = {
  getAllProjects,
  getProject,
  createProject,
  updateProject,
  updateLegacyCMS,
  updateAffix,
  affixConfirmation,
  updateFileFormat,
  fileformatConfirmation,
  updateDestinationStack,
  updateCurrentStep,
  deleteProject,
  revertProject,
  updateStackDetails,
  updateContentMapper,
  updateMigrationExecution,
  getMigratedStacks
};
```

Usage Example

```
import { projectService } from './services/projects.service';


// Example: Creating a project
const result = await projectService.createProject(req);
```

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

```
console.log(result.status, result.message, result.project);
```

Notes

- **All functions expect a standard Express `Request` object, with required parameters and a `token_payload` in the request body for authentication and authorization.**
- **The service is designed to be used in an Express.js controller or route handler context.**
- **The stepper logic enforces a strict workflow for project progression.**

`src/services/runCli.service.ts`

Log Level Detection:

- **The service analyzes CLI output to classify log entries as `info`, `warn`, or `error` for structured logging.**

ANSI Stripping:

- **Removes color codes from CLI output before writing to log files, ensuring clean logs.**

Backup and Logging:

- **Before migration, the service creates a backup of the source data and sets up log files for both backup and main migration logs.**

CLI Execution:

- **Uses Node's `spawn` to run CLI commands asynchronously, streaming output to both the console and log files.**

Project Status Management:

- **Updates the local project database to reflect migration progress, supporting both test and production workflows.**

Authentication:

- **Reads user authentication data and configures the CLI session accordingly.**

Usage

Import and use the `runCli` function to trigger a migration:

```
import { utilsCli } from './services/runCli.service';


await utilsCli.runCli(
  'NA',                  // region
  'user_id_123',         // user ID
  'stack_uid_abc',       // stack UID
  'project_id_xyz',      // project ID
  false,                 // isTest
  '/path/to/log.log'     // log file path
);
```

Note:

- **The service expects certain directory structures and configuration constants to be defined in your project.**
- **Log files are written in JSON lines format for easy parsing and UI integration.**
- **The service is designed to be robust for both test and production migrations, with clear separation of concerns and error handling.**

Sitecore Service (`sitecore.service.ts`)

This service provides a set of utilities for transforming, migrating, and saving Sitecore content, assets, and locale data into a structure suitable for further processing or import into another

system (such as Contentstack). It handles reading Sitecore export packages, extracting and transforming entries and assets, mapping locales, and writing the processed data to disk.

Dependencies

- **Node.js core modules:** `fs`, `path`
- **Third-party modules:** `fs-readdir-recursive`, `uuid`, `lodash`
- **Project utilities: Constants, entry field creators, logging, path sanitization, and organization service**

Exported Service Methods

1. `createEntry`

```
async function createEntry({
  packagePath,
  contentTypes,
  master_locale,
  destinationStackId,
  projectId,
  keyMapper,
  project,
}): Promise<boolean>
```

Description:

Transforms Sitecore content entries into a new format, mapping fields and locales, and writes them to disk. Also triggers asset extraction and transformation.

Parameters:

- `packagePath` **(string): Path to the root of the Sitecore export package.**

- `contentTypes` (Array): List of content type definitions with field mappings.
- `master_locale` (string, optional): The master locale code.
- `destinationStackId` (string): Target stack identifier for output directory structure.
- `projectId` (string): Project identifier for logging.
- `keyMapper` (object): Maps Sitecore template IDs to Contentstack UIDs.
- `project` (object): Project configuration, including locales.

Returns:

`Promise<boolean>` — Returns `true` on success, logs errors otherwise.

Key Implementation Details:

- Reads and transforms all entries for each content type and locale.
- Maps Sitecore field keys to Contentstack field UIDs.
- Handles asset references and field value transformations.
- Writes each entry and its locale data to a structured directory.
- Logs progress and errors using a custom logger.

---

2. `createAssets`

```
async function createAssets({
  packagePath,
  baseDir,
  destinationStackId,
  projectId,
}): Promise<object>
```

Description:

Extracts, transforms, and saves Sitecore assets (media files) to disk, and generates a metadata JSON for all assets.

Parameters:

- `packagePath` **(string): Path to the Sitecore export package.**
- `baseDir` **(string): Base directory for output.**
- `destinationStackId` **(string): Target stack identifier.**
- `projectId` **(string): Project identifier for logging.**

Returns:

`Promise<object>` **— Returns an object mapping asset UIDs to their metadata.**

Key Implementation Details:

- **Reads asset metadata and binary blobs from the Sitecore package.**
- **Normalizes asset IDs and filenames.**
- **Writes asset files and metadata to the output directory.**
- **Logs asset processing status and errors.**

---

3. `createLocale`

```
async function createLocale(
  req: any,
  destinationStackId: string,
  projectId: string,
  project: any
): Promise<void>
```

Description:

Generates and writes locale configuration files based on the project and organization settings.

Parameters:

- `req` **(any): Request object for organization service (used to fetch locales).**
- `destinationStackId` **(string): Target stack identifier.**
- `projectId` **(string): Project identifier for logging.**
- `project` **(object): Project configuration, including locales.**

Returns:

`Promise<void>`

Key Implementation Details:

- **Fetches locale names from the organization service.**
- **Generates unique UIDs for each locale.**
- **Writes master and additional locale files to disk.**
- **Logs locale creation status and errors.**

---

4. `createVersionFile`

```
async function createVersionFile(destinationStackId: string):
Promise<void>
```

Description:

Writes a version info file to the output directory, indicating the content version and logs path.

Parameters:

- `destinationStackId` **(string): Target stack identifier.**

Returns:

`Promise<void>`

Helper Functions

`idCorrector`

Normalizes Sitecore IDs by removing dashes and braces, and converting to lowercase.

`uidCorrector`

Normalizes UIDs by replacing spaces and dashes with underscores, prepending with 'a' if the UID starts with a number, and converting to lowercase.

`AssetsPathSplitter`

Extracts the relative asset path from a full Sitecore asset path.

`mapLocales`

Maps a locale code to its corresponding key in the locales object, handling master locale mapping.

`writeFiles` and `writeOneFile`

Utility functions for writing entry and locale files to disk, ensuring directories exist.

Directory Structure

- **Entries: Saved under `DATA/STACK_ID/entries/CONTENT_TYPE/LOCALE/`**
- **Assets: Saved under `DATA/STACK_ID/assets/files/ASSET_UID/`**
- **Locales: Saved under `DATA/STACK_ID/locales/`**
- **Version Info: Saved under `DATA/STACK_ID/EXPORT_INFO_FILE`**

CONTENTSTACK

Logging

All major operations (entry transformation, asset processing, locale creation) are logged using a custom logger, with both info and error levels, including contextual messages for traceability.

Error Handling

- **All file operations are wrapped in try/catch blocks or error callbacks.**
- **Errors are logged to the console and to the custom logger for later review.**

Export

The service is exported as a singleton object:

```
export const siteCoreService = {
  createEntry,
  createAssets,
  createLocale,
  createVersionFile,
};
```

Note:

This service is designed for use in a Node.js environment and expects Sitecore export packages to follow a specific directory and file naming convention. It is intended to be used as part of a migration or integration pipeline.

User Service (`user.service.ts`)

This service provides user-related operations, primarily focused on retrieving user profile information from an external API, based on authentication and request context.

CONTENTSTACK

Overview

The `userService` module exposes methods to interact with user data, specifically to fetch a user's profile and their associated organizations and roles. It integrates with authentication models and external APIs, handling errors and logging as needed.

Dependencies

- **Express Request: For accessing request data and token payloads.**
- **Configuration: Uses API endpoints and settings from the app's config.**
- **HTTPS Utility: For making HTTP requests to external services.**
- **Authentication Model: For reading and validating user authentication data.**
- **Custom Errors: For standardized error handling.**
- **Logger: For error and event logging.**
- **Utility Functions: For safe promise handling and log message formatting.**

Functions

getUserProfile

```
const getUserProfile = async (req: Request): Promise<LoginServiceType>
```

Description:

Retrieves the user profile for the authenticated user making the request. It checks the authentication model for the user, fetches the profile from an external API, and returns structured user data including organizations where the user has admin roles.

Parameters:

- `req: Request`
  The Express request object, expected to contain a `token_payload` in the request body.

Returns:

A `Promise<LoginServiceType>` resolving to an object containing user profile data and HTTP status.

Throws:

- `BadRequestError` **if the user is not found in the authentication model or the external API response.**
- `ExceptionFunction` **for any other errors encountered during the process.**

Process:

1. **Reads the authentication model to ensure user data is loaded.**
2. **Locates the user by `user_id` and `region` from the token payload.**
3. **If the user is not found, throws a `BadRequestError`.**
4. **Makes a GET request to the external API to fetch the user profile, including organizations and roles.**
5. **Handles errors from the API call, logging them and returning the error response.**
6. **Extracts organizations where the user has admin roles.**
7. **Returns the user's email, first name, last name, and admin organizations.**

Error Handling

User Not Found:

- **If the user is not present in the authentication model or the API response, a `BadRequestError` is thrown with a relevant message.**

External API Errors:

- **If the API call fails, the error is logged and the error response is returned.**

Unexpected Errors:

- **Any other errors are logged and rethrown as an `ExceptionFunction` with a generic internal error message and status code.**

Usage Example

```
import { userService } from './services/user.service';


// Express route handler example
app.get('/profile', async (req, res, next) => {
  try {
    const profile = await userService.getUserProfile(req);
    res.status(profile.status).json(profile.data);
  } catch (error) {
    next(error);
  }
});
```

Exports

- `userService`:
  An object containing the `getUserProfile` function.

WordPress Service

This service provides a set of utilities for migrating WordPress data (posts, authors, assets, categories, tags, terms, references, etc.) into a Contentstack-compatible format. It handles reading, transforming, and writing data, as well as managing directories and logging.

Overview

The service is organized into several modules, each responsible for a specific aspect of the migration process:

- **Locale Management**
- **Asset Management**
- **Reference Management**
- **Chunk Management**
- **Author Management**
- **Content Type Management**
- **Term, Tag, and Category Management**
- **Post Management**
- **Global Fields Management**
- **Version File Management**

Modules and Functions

Locale Management

`createLocale(req, destinationStackId, projectId, project)`

Creates locale files and directories for the destination stack, including the master locale and all additional locales. Logs the process and handles errors.

- **Parameters:**
  - `req`**: Request object for API calls.**
  - `destinationStackId`**: Target stack ID.**
  - `projectId`**: Project identifier.**
  - `project`**: Project configuration object.**

Asset Management

`getAllAssets(affix, packagePath, destinationStackId, projectId)`

Reads WordPress export data, filters for attachments, and downloads assets to the local filesystem. Handles retries and logs failures.

- **Parameters:**
  - `affix`: **String to prefix asset IDs.**
  - `packagePath`: **Path to the WordPress export file.**
  - `destinationStackId`: **Target stack ID.**
  - `projectId`: **Project identifier.**

`createAssetFolderFile(affix, destinationStackId, projectId)`

Creates a folder JSON file for assets, used for organizing assets in Contentstack.

Reference Management

`getAllreference(affix, packagePath, destinationStackId, projectId)`

Processes and saves references (categories, terms, tags) from the WordPress export.

Chunk Management

`extractChunks(affix, packagePath, destinationStackId, projectId)`

Splits large post datasets into manageable chunks for processing and migration.

Author Management

`getAllAuthors(affix, packagePath, destinationStackId, projectId, contentTypes, keyMapper, master_locale, project)`

Extracts and saves author data, creating locale-specific files and indexes.

Content Type Management

`extractContentTypes(projectId, destinationStackId)`

Generates and saves Contentstack content type schemas for authors, categories, tags, terms, and posts.

Term, Tag, and Category Management

`getAllTerms(affix, packagePath, destinationStackId, projectId, contentTypes, keyMapper, master_locale, project)`

Extracts and saves term data.

`getAllTags(affix, packagePath, destinationStackId, projectId, contentTypes, keyMapper, master_locale, project)`

Extracts and saves tag data.

`getAllCategories(affix, packagePath, destinationStackId, projectId, contentTypes, keyMapper, master_locale, project)`

Extracts and saves category data.

Post Management

`extractPosts(packagePath, destinationStackId, projectId, contentTypes, keyMapper, master_locale, project)`

Processes post data in chunks, transforms it to Contentstack format, and writes locale-specific files.

Global Fields Management

`extractGlobalFields(destinationStackId, projectId)`

Copies global field and locale folders from a source directory to the migration data directory.

Version File Management

`createVersionFile(destinationStackId, projectId)`

Creates a version file in the migration data directory to track the content version.

Helper Functions

- `mapContentTypeToEntry(contentType, data)`: **Maps WordPress data fields to Contentstack fields based on a field mapping configuration.**
- `writeFileAsync(filePath, data, tabSpaces)`: **Writes data to a file asynchronously.**
- `idCorrector(id)`: **Normalizes and formats IDs for consistency.**
- `convertHtmlToJson(htmlString)`: **Converts HTML to Contentstack JSON RTE format.**
- `convertJsonToHtml(json)`: **Converts Contentstack JSON RTE to HTML.**
- `getParent(data, id)`: **Finds and returns parent reference objects for categories, terms, or tags.**

Exported Service

The following functions are exported as part of the `wordpressService` object:

```
export const wordpressService = {
  getAllAssets,
  createLocale,
  createAssetFolderFile,
  getAllreference,
  extractChunks,
  getAllAuthors,
  extractContentTypes,
```

```
  getAllTerms,
  getAllTags,
  getAllCategories,
  extractPosts,
  extractGlobalFields,
  createVersionFile
};
```

Usage Example

```
import { wordpressService } from './services/wordpress.service';

// Example: Extract all posts from a WordPress export
await wordpressService.extractPosts(
  'path/to/wordpress-export.json',
  'destinationStackId',
  'projectId',
  contentTypes,
  keyMapper,
  'en-us',
  projectConfig
);
```

async-router.utils.ts

Utility for handling errors in asynchronous Express route handlers.

Overview

CONTENTSTACK

When working with Express, asynchronous route handlers that throw errors or reject promises do not automatically pass those errors to the Express error handler. This utility provides a wrapper function, `asyncRouter`, that ensures any errors thrown in an async route handler are properly forwarded to the next middleware (typically, the error handler).

Usage

Import the `asyncRouter` function and use it to wrap your async route handlers:

```
import { asyncRouter } from './utils/async-router.utils';


app.get('/example', asyncRouter(async (req, res, next) => {
  // Your async logic here
  const data = await someAsyncFunction();
  res.json(data);
}));
```

API

asyncRouter

```
asyncRouter(fn: (req: Request, res: Response, next: NextFunction) =>
Promise<any>): (req: Request, res: Response, next: NextFunction) => void
```

Parameters

- **`fn`: An asynchronous function (route handler) that takes `req`, `res`, and `next` as arguments.**

Returns

- **A middleware function that executes the async function and catches any errors, passing them to Express's `next` function.**

Example

```
import { asyncRouter } from './utils/async-router.utils';


router.post('/users', asyncRouter(async (req, res) => {
```

```
  const user = await createUser(req.body);

  res.status(201).json(user);

}));
```

Implementation

```
import { Request, Response, NextFunction } from "express";


/**

 * Wraps an async function to handle errors and pass them to the Express error handler.

 * @param fn - The async function to be wrapped.

 * @returns A middleware function that handles async errors.

 */

export const asyncRouter =

  (fn: any) => (req: Request, res: Response, next: NextFunction) => {

    Promise.resolve(fn(req, res, next)).catch(next);

  };
```

Notes

- **This utility is especially useful for keeping your route handlers clean and avoiding repetitive try/catch blocks.**
- **It works with any async function that follows the Express middleware signature.**

`auth.utils.ts`

Utility for retrieving a user's authentication token based on region and user ID.

Overview

This utility provides a single asynchronous function that fetches the authentication token for a user in a specified region. It interacts with the `AuthenticationModel` to locate the user and extract their token. If the user is not found or the token is missing, it throws an `UnauthorizedError`.

Function Signature

```
/**
 * Retrieves the authentication token for a given user in a specific region.
 * @param region - The region of the user.
 * @param userId - The ID of the user.
 * @returns The authentication token for the user.
 * @throws UnauthorizedError if the user is not found or the authentication token is missing.
 */
export default async (region: string, userId: string): Promise<string>
```

Parameters

region (`string`):

- **The region associated with the user whose authentication token is being requested.**

userId (`string`):

- **The unique identifier of the user.**

Returns

- **Promise:**
  Resolves to the authentication token for the specified user.

Throws

- **UnauthorizedError:**
  Thrown if the user cannot be found in the specified region or if the authentication token is missing.

Example Usage

```
import getAuthToken from "./utils/auth.utils";


try {
  const token = await getAuthToken("us-east", "user-123");
  // Use the token for further authentication
} catch (error) {
  if (error instanceof UnauthorizedError) {
    // Handle unauthorized access
  }
}
```

Implementation Details

- **The function first ensures the authentication data is loaded by calling `AuthenticationModel.read()`.**
- **It searches for the user in the `users` array by matching both `region` and `user_id`.**
- **If a matching user is found, it retrieves the `authtoken` property.**
- **If the user is not found or the token is missing, it throws an `UnauthorizedError`.**

Content Type Creator Utilities

This module provides utility functions for transforming, mapping, and saving Contentstack content types and global fields during migration or integration processes. It handles schema conversion, UID correction, group arrangement, and file operations for content type definitions.

CONTENTSTACK

Interfaces

Group

Represents a group field in a content type schema.

```
interface Group {
  data_type: string;
  display_name?: string;
  field_metadata: Record<string, any>;
  schema: any[];
  uid?: string;
  multiple: boolean;
  mandatory: boolean;
  unique: boolean;
}
```

ContentType

Represents a content type with a title, UID, and schema.

```
interface ContentType {
  title: string | undefined;
  uid: string | undefined;
  schema: any[];
}
```

Helper Functions

extractFieldName(input: string): string

Extracts and cleans the field name from a string, removing "-App" and extracting text inside parentheses.

```
extractValue(input: string, prefix: string, another: string): any
```

Extracts a value from a string based on a prefix and a separator.

```
startsWithNumber(str: string): boolean
```

Checks if a string starts with a number.

```
uidCorrector({ uid }: any): string
```

Normalizes UIDs by replacing spaces and hyphens with underscores and prepending 'a_' if the UID starts with a number.

Schema Conversion

```
arrangGroups({ schema, newStack }: any): any[]
```

Arranges group fields and their nested schema from a flat schema array.

```
convertToSchemaFormate({ field, advanced = true, marketPlacePath }: any): any
```

Converts a field mapping object to the target schema format for Contentstack, handling all supported field types (text, boolean, json, dropdown, radio, checkbox, file, link, multi_line_text, markdown, number, isodate, global_field, reference, html, app, extension, and default).

File Operations

```
saveAppMapper({ marketPlacePath, data, fileName }: any): Promise<void>
```

Ensures the marketplace directory exists and appends app/extension mapping data to a file.

```
saveContent(ct: any, contentSave: string): Promise<void>
```

Saves a content type schema to a file and appends it to a master schema file.

`writeGlobalField(schema: any, globalSave: string): Promise<void>`

Saves a global field schema to a file, appending it to a global fields file.

Content Type Processing

`existingCtMapper({ keyMapper, contentTypeUid, projectId, region, user_id }: any): Promise<any>`

Fetches the existing content type schema from Contentstack using a mapping and service call.

`mergeArrays(a: any[], b: any[]): Promise<any[]>`

Merges two arrays of fields, avoiding duplicates by UID and data type.

`mergeTwoCts(ct: any, mergeCts: any): Promise<any>`

Merges two content type schemas, combining their fields and groups.

Exported Function

`contenTypeMaker`

Main function to transform and save a content type or global field.

```
export const contenTypeMaker = async ({
  contentType,
  destinationStackId,
  projectId,
  newStack,
  keyMapper,
  region,
  user_id
}: any) => { ... }
```

Parameters

- `contentType`: The source content type object to transform.
- `destinationStackId`: The target stack ID for saving files.
- `projectId`: The project ID for logging.
- `newStack`: Boolean indicating if this is a new stack migration.
- `keyMapper`: Mapping of source to destination content type UIDs.
- `region`: Contentstack region.
- `user_id`: User ID for API/service calls.

Workflow

1. **Setup Paths: Determines the save paths for content types and global fields.**
2. **Existing Content Type: If a mapping exists, fetches the current content type schema for merging.**
3. **Schema Transformation: Arranges groups and converts each field to the target schema format.**
4. **Merging: If an existing content type is found, merges the new and existing schemas.**
5. **Saving: Writes the transformed schema to the appropriate file (content type or global field).**
6. **Logging: Logs the transformation result.**

---

api/src/utils/custom-errors.utils.ts

Custom Error Utilities

This module provides a set of custom error classes for handling various error scenarios in your application. Each error class extends the base AppError class and is associated with a specific HTTP status code and message. These custom errors help standardize error handling and improve the clarity of error responses throughout the codebase.

AppError

export class AppError extends Error

The base class for all custom application errors.

- **Parameters:**
  - **statusCode (number): The HTTP status code for the error.**
  - **message (string): The error message.**

NotFoundError

export class NotFoundError extends AppError

Represents a "Resource Not Found" error (HTTP 404).

- **Default message: "Not Found"**

BadRequestError

export class BadRequestError extends AppError

Represents a "Bad Request" error (HTTP 400).

- **Default message: "Bad Request"**

DatabaseError

export class DatabaseError extends AppError

Represents a database operation error (HTTP 500).

- **Default message: "DB error"**

ValidationError

export class ValidationError extends AppError

Represents a user validation error (HTTP 422).

- **Default message: "User validation error"**

InternalServerError

export class InternalServerError extends AppError

Represents an internal server error (HTTP 500).

- **Default message: Uses HTTP_TEXTS.INTERNAL_ERROR.**

UnauthorizedError

export class UnauthorizedError extends AppError

Represents an unauthorized access error (HTTP 401).

- **Default message: Uses HTTP_TEXTS.UNAUTHORIZED.**

S3Error

export class S3Error extends AppError

Represents an error related to S3 operations (HTTP 500).

- **Default message: Uses HTTP_TEXTS.S3_ERROR.**

ExceptionFunction

export class ExceptionFunction extends AppError

A flexible custom exception class for any HTTP status code.

- **Parameters:**
  - **message (string): The error message.**
  - **httpStatus (number): The HTTP status code.**

Custom Logger Utility

This module provides a secure, flexible logging utility for writing project- and API-key-specific logs to disk, with strong protections against directory traversal and other file system attacks. It leverages Winston for structured logging and supports dynamic log levels.

Features

- **Safe Path Handling: Prevents directory traversal using path sanitization and validation.**
- **Per-Project and Per-API-Key Logging: Logs are organized by project and API key.**
- **Automatic Directory and File Creation: Ensures log directories and files exist before writing.**
- **Flexible Log Levels: Supports `error`, `warn`, `info`, and `debug` levels.**
- **Dual Logging: Writes to both file and console (or a global logger).**
- **Environment Awareness: Prints stack traces for new log files in non-production environments.**

Exports

`customLogger(projectId: string, apiKey: string, level: string, message: string): Promise<void>`

Logs a message to a file specific to the given project and API key, and also to the main logger/console.

Parameters:

- `projectId` (`string`): **The project identifier. Used as a directory name under `logs/`.**
- `apiKey` (`string`): **The API key. Used as the log file name within the project directory.**
- `level` (`string`): **The log level (`error`, `warn`, `info`, `debug`). Defaults to `info` if unrecognized.**
- `message` (`string`): **The message to log.**

Behavior:

- **Sanitizes `projectId` and `apiKey` to prevent path traversal.**
- **Ensures the log directory and file exist, creating them if necessary.**
- **Logs the message at the specified level to both the file and the main logger.**
- **In non-production environments, prints a stack trace when creating a new log file.**

Internal Utilities

`safeJoin(basePath: string, ...paths: string[]): string`

Safely joins and resolves paths, ensuring the result is within `basePath`. Throws an error if a directory traversal attempt is detected.

Parameters:

- `basePath` (`string`): **The base directory.**
- `...paths` (`string[]`): **Path segments to join.**

Returns:

A safe, absolute path within `basePath`.

`fileExists(path: string): Promise<boolean>`

Checks asynchronously if a file or directory exists.

Parameters:

- `path` (`string`): **The path to check.**

Returns:

`true` **if the file exists,** `false` **otherwise.**

Example Usage

```
import customLogger from './custom-logger.utils';


await customLogger('myProject', 'myApiKey', 'info', 'This is a log
message.')
```

This will write the message to `logs/myProject/myApiKey.log` and also output it via the main logger.

Security Notes

- **All file and directory names are sanitized to prevent directory traversal.**
- **Paths are validated to ensure logs are only written within the intended directory structure.**

Dependencies

- **winston**
- **fs**
- **path**
- **`getSafePath` from `sanitize-path.utils.js`**
- **A global `logger` instance from `logger.js`**

Error Handling

- **If a log file cannot be created or written, an error is printed to the console.**
- **If a directory traversal attempt is detected, an error is thrown and logged.**

entries-field-creator.utils.ts

This utility module provides a set of helper functions for processing and transforming Contentstack entry fields, including text, JSON RTE, dropdowns, references, assets, and more. It is designed to support migration, transformation, and normalization of content data, especially when working with Contentstack APIs and custom field mappings.

Dependencies

- **`lodash`: Utility library for object and string manipulation.**
- **`jsdom`: Used to parse and manipulate HTML content in a Node.js environment.**
- **`@contentstack/json-rte-serializer`: Converts HTML to Contentstack JSON RTE format.**
- **`html-to-json-parser`: Converts HTML to a JSON structure.**

Helper Functions

startsWithNumber

```
function startsWithNumber(str: string): boolean
```

Checks if a string starts with a numeric character.

---

uidCorrector

```
const uidCorrector = ({ uid }: any): string
```

Normalizes a UID by:

- **Prefixing with `"a_"` if it starts with a number.**
- **Replacing spaces and hyphens with underscores.**
- **Lowercasing the result.**

---

attachJsonRte

```
const attachJsonRte = ({ content = "" }: any): any
```

Converts HTML content to Contentstack JSON RTE format using `jsdom` and `@contentstack/json-rte-serializer`.

---

unflatten

```
export function unflatten                    any
```

Transforms a flat object with path-like keys (e.g., `a.b[<span class='inlineRef'>1</span>].c`) into a nested object structure.

---

htmlConverter

```
const htmlConverter = async ({ content = "" }: any): Promise<any>
```

Converts HTML content to a JSON structure using `html-to-json-parser`.

---

getAssetsUid

```
const getAssetsUid = ({ url }: any): string | undefined
```

Extracts the asset UID from a Contentstack asset URL, handling various URL formats.

---

flatten

```
function flatten(data: any): any
```

Flattens a nested object into a single-level object with path-like keys.

---

findAssestInJsoRte

```
const findAssestInJsoRte = (jsonValue: any, allAssetJSON: any,
idCorrector: any): any
```

Scans a JSON RTE structure for embedded images, matches them to asset metadata, and replaces them with Contentstack asset reference objects.

Main Export

entriesFieldCreator

```
export const entriesFieldCreator = async ({
```

```
  field,
  content,
  idCorrector,
  allAssetJSON,
  contentTypes,
  entriesData,
  locale
}: any): Promise<any>
```

Description

A central function that processes a field value based on its Contentstack field type. It handles various field types, including:

- Text fields (`multi_line_text`, `single_line_text`, `text`): **Returns the content as-is.**
- **JSON RTE (`json`): Converts HTML to JSON RTE and replaces embedded assets with references.**
- **Dropdowns (`dropdown`): Maps the value to the correct option, handling defaults and multiple selections.**
- **Numbers (`number`): Parses string numbers to integers.**
- **Files (`file`): Extracts asset references from JSON RTE.**
- **Links (`link`): Converts HTML links to a structured object with `title` and `href`.**
- **References (`reference`): Resolves references to other entries based on provided mappings.**
- **Global Fields (`global_field`): Recursively processes nested field mappings.**
- **Booleans (`boolean`): Converts string `"1"` to `true`, otherwise `false`.**
- **Default: Returns the content as-is and logs missing field types.**

Parameters

- `field`: Field schema object describing the Contentstack field.
- `content`: The raw value/content for the field.
- `idCorrector`: Function to normalize or correct IDs.
- `allAssetJSON`: Object mapping asset UIDs to asset metadata.
- `contentTypes`: Array of content type schemas (for global fields).
- `entriesData`: Array of entry data (for resolving references).
- `locale`: Current locale (for localized references).

Returns

- **The processed field value, ready for use in Contentstack or further transformation.**

---

field-attacher.utils.ts

This utility provides a function to attach field mappings to content types for a given project and then process those content types for migration or synchronization to a destination stack. It is typically used in the context of content migration or stack synchronization workflows.

Imports

- **ProjectModelLowdb: Handles reading and querying project data from a lowdb database.**
- **ContentTypesMapperModelLowdb: Handles reading and querying content type mappings from a lowdb database.**
- **FieldMapperModel: Handles reading and querying field mappings from a lowdb database.**
- **contenTypeMaker: Utility function to process and create content types in the destination stack.**

`fieldAttacher` Function

Signature

```
export const fieldAttacher = async ({
  projectId,
  orgId,
  destinationStackId,
  region,
  user_id
}: any) => Promise<any[]>
```

Description

The `fieldAttacher` function orchestrates the process of:

1. **Loading project, content type, and field mapping data from local databases.**
2. **For each content type associated with the project, it:**
    - **Attaches the corresponding field mapping objects.**
    - **Invokes the `contenTypeMaker` utility to process the content type for the destination stack.**
3. **Returns an array of processed content type objects, each with their field mappings attached.**

Parameters

- **projectId (`string`): The unique identifier of the project.**
- **orgId (`string`): The unique identifier of the organization.**
- **destinationStackId (`string`): The ID of the stack where content types will be created or updated.**
- **region (`string`): The region identifier for the destination stack.**
- **user_id (`string`): The user ID performing the operation.**

All parameters are passed as properties of a single object.

Returns

- **Promise<any[]>: Resolves to an array of content type objects, each with their field mappings attached and processed.**

Workflow

1. **Read Data: Loads the latest data from the project, content type mapper, and field mapper lowdb models.**
2. **Find Project: Retrieves the project data matching the given `projectId` and `orgId`.**
3. **Iterate Content Types: For each content type ID in the project's `content_mapper`:**
    - **Retrieves the content type mapping.**
    - **For each field UID in the content type's `fieldMapping`, replaces the UID with the full field mapping object.**
    - **Calls `contenTypeMaker` to process the content type for the destination stack.**
    - **Adds the processed content type to the result array.**
4. **Return: Returns the array of processed content types.**

get-project.utils.ts

Utility function for retrieving a project from the database by project ID and query, with robust error handling and logging.

Overview

This module exports an asynchronous function that retrieves a project (or its index) from the database using a provided project ID and query object. It validates the project ID, reads from the database, and returns the matching project or its index. The function includes detailed error handling and logging for invalid IDs, missing projects, and unexpected errors.

Function Signature

```
/**
 * Retrieves a project based on the provided project ID and query.
 * @param projectId - The ID of the project to retrieve.
```

```
 * @param query - The query to filter the projects.

 * @param srcFunc - The source function name (optional, for logging).

 * @param isIndex - If true, returns the index of the project instead of
the project object (default: false).

 * @returns The retrieved project object or its index.

 * @throws BadRequestError if the project ID is invalid or the project is
not found.

 * @throws ExceptionFunction for unexpected errors during retrieval.

 */

export default async (

  projectId: string,

  query: MigrationQueryType,

  srcFunc: string = "",

  isIndex: boolean = false

): Promise<any>
```

Parameters

projectId (`string`):

- **The unique identifier of the project to retrieve. Must be a valid UUID.**

query (`MigrationQueryType`):

- **An object specifying the query criteria to filter projects.**

srcFunc (`string`, optional):

- **The name of the source function calling this utility, used for logging context. Defaults to an empty string.**

isIndex (`boolean`, optional):

- If `true`, the function returns the index of the project in the collection; otherwise, it returns the project object. Defaults to `false`.

Returns

- The project object matching the query, or its index if `isIndex` is `true`.

Throws

- BadRequestError:
    - If the `projectId` is not a valid UUID.
    - If the project is not found (either as an object or index).
- ExceptionFunction:
    - For any unexpected errors during the retrieval process, with error details and status code.

Internal Logic

Validation:

1. Validates the `projectId` using the `uuid` package. Logs and throws a `BadRequestError` if invalid.

Database Read:

2. Reads the latest state from the `ProjectModel`.
3. Query Execution:
    - If `isIndex` is `true`, finds the index of the project matching the query.
    - Otherwise, finds the project object.
4. Result Handling:
    - If not found (index < 0 or object is falsy), logs and throws a `BadRequestError`.

5. **Error Handling:**
    - **Catches any unexpected errors, logs them, and throws an `ExceptionFunction` with details.**

Dependencies

- **`ProjectModel`: Lowdb model for projects.**
- **`custom-errors.utils`: Custom error classes.**
- **`constants/index`: HTTP status codes and texts.**
- **`uuid`: For UUID validation.**
- **`logger`: Logging utility.**
- **`getLogMessage`: Helper for formatted log messages.**

https.utils.ts

A utility module for sending HTTP requests using Axios with a unified interface and configurable options.

Overview

This module exports a single asynchronous function that wraps Axios to send HTTP requests. It supports custom headers, data payloads, configurable timeouts, and automatically includes data for specific HTTP methods. The function returns a simplified response object containing headers, status, and data.

Usage

```
import httpRequest from "./utils/https.utils";


const response = await httpRequest({
  url: "https://api.example.com/resource",
  method: "POST",
  headers: { "Authorization": "Bearer token" },
  data: { key: "value" },
```

```
  timeout: 5000,
});

console.log(response.status);  // e.g., 200
console.log(response.data);    // Response body
```

Type Definitions

httpType

```
type httpType = {
  url: string;              // The request URL
  method: string;           // HTTP method (e.g., 'GET', 'POST', etc.)
  headers?: any;            // Optional HTTP headers
  data?: any;               // Optional request payload (for methods like POST, PUT)
  timeout?: number;         // Optional request timeout in milliseconds
};
```

Function

Default Export

```
async function httpRequest(obj: httpType): Promise<{
  headers: any;
  status: number;
  data: any;
}>
```

Parameters:

- `obj` (`httpType`): **The HTTP request configuration object.**

- `url`: The endpoint to send the request to.
- `method`: The HTTP method to use.
- `headers` (optional): Custom headers to include in the request.
- `data` (optional): The request body, included only for methods specified in `METHODS_TO_INCLUDE_DATA_IN_AXIOS`.
- `timeout` (optional): Request timeout in milliseconds. Defaults to `AXIOS_TIMEOUT` if not provided.

Returns:

A Promise that resolves to an object containing:

- `headers`: The response headers.
- `status`: The HTTP status code.
- `data`: The response body.

Implementation Notes

- The function uses the `axios` library for HTTP requests.
- The `timeout` defaults to `AXIOS_TIMEOUT` if not specified.
- The `data` property is only included for HTTP methods listed in `METHODS_TO_INCLUDE_DATA_IN_AXIOS`.
- The response is normalized to always return `headers`, `status`, and `data`.

`api/src/utils/index.ts`

Utils Module (`api/src/utils/index.ts`)

This module provides a set of utility functions for error handling, value checking, logging, file system operations, and API requests. These utilities are designed to be reusable across the application.

`throwError`

```
export const throwError = (message: string, statusCode: number) => { ... }
```

Throws an error with a custom message and HTTP status code.

Parameters:

- `message` (`string`): **The error message.**
- `statusCode` (`number`): **The HTTP status code to associate with the error.**

Throws:

An `Error` object with the specified message and a `statusCode` property.

`isEmpty`

```
export const isEmpty = (val: unknown) => { ... }
```

Checks if a value is empty.

Parameters:

- `val` (`unknown`): **The value to check.**

Returns:

`true` if the value is `undefined`, `null`, an empty object, or an empty/whitespace-only string; otherwise, `false`.

`safePromise`

```
export const safePromise = (promise: Promise<any>): Promise<any> => { ...
}
```

Wraps a promise to always resolve with a tuple `[error, result]`.

Parameters:

- `promise` (`Promise<any>`): The promise to wrap.

Returns:

A promise that resolves to `[null, result]` on success or `[error]` on failure.

`getLogMessage`

```
export const getLogMessage = (
  methodName: string,
  message: string,
  user = {},
  error?: any
) => { ... }
```

Generates a structured log message object.

Parameters:

- `methodName` (`string`): The name of the method generating the log.
- `message` (`string`): The log message.
- `user` (`object`, optional): The user context (default: `{}`).
- `error` (`any`, optional): The error object, if any.

Returns:

An object containing the log details.

`copyDirectory`

```
export async function copyDirectory(srcDir: string, destDir: string): Promise<void> { ... }
```

Recursively copies a directory from a source to a destination.

Parameters:

- `srcDir` (`string`): **Source directory path.**
- `destDir` (`string`): **Destination directory path.**

Returns:

`Promise<void>`

Notes:

Logs success or error messages to the console.

`createDirectoryAndFile`

```
export async function createDirectoryAndFile(filePath: string, sourceFile: string) { ... }
```

Creates a directory (if it doesn't exist) and a file at the specified path, copying content from a source file.

Parameters:

- `filePath` (`string`): **The path where the file should be created.**
- `sourceFile` (`string`): **The path to the source file whose contents will be copied.**

Returns:

`Promise<void>`

Notes:

Logs whether the file was created or already exists, and logs errors if any occur.

`getAllLocales`

```
export async function getAllLocales () { ... }
```

Fetches all locales from the configured API endpoint.

Returns:

A promise that resolves to a tuple `[error, locales]`, where `locales` is the list of locales returned by the API.

Notes:

Uses the `safePromise` utility for error handling.

Dependencies

- **fs-extra: For file system operations.**
- **path: For path manipulations.**
- **mkdirp: For recursive directory creation.**
- **../config/index.js: Application configuration.**
- **../utils/https.utils.js: HTTP request utility.**

`src/utils/jwt.utils.ts`

This utility module provides a function to generate JSON Web Tokens (JWT) for authentication and authorization purposes in your application.

Dependencies

- jsonwebtoken: Used for creating and signing JWT tokens.
- AppTokenPayload: Type definition for the payload structure, imported from your application's models.

- config: Application configuration, which should provide the secret key and token expiration settings.

Function: `generateToken`

```
/**
 * Generates a JWT token with the provided payload.
 *
 * @param payload - The payload to be included in the token. Must conform to the AppTokenPayload type.
 * @returns The generated JWT token as a string.
 *
 * @example
 * const payload = { userId: "123", role: "admin" };
 * const token = generateToken(payload);
 */
export const generateToken = (payload: AppTokenPayload): string => {
  return jwt.sign(payload, config.APP_TOKEN_KEY, {
    expiresIn: config.APP_TOKEN_EXP,
  });
};
```

Parameters

- payload (`AppTokenPayload`):
  The data to be embedded in the JWT. This typically includes user identification and any claims required for your application's authentication logic.

Returns

- string:
  The signed JWT token as a string.

Usage Example

```
import { generateToken } from "./utils/jwt.utils";
import { AppTokenPayload } from "./models/types";

const payload: AppTokenPayload = {
  userId: "abc123",
  role: "user",
};

const token = generateToken(payload);
// Use the token for authentication headers, etc.
```

Configuration Requirements

- `config.APP_TOKEN_KEY`:
  A secret string used to sign the JWT. This should be kept secure and not exposed publicly.
- `config.APP_TOKEN_EXP`:
  A string or number indicating the token's expiration time (e.g., `"1h"`, `"7d"`, `3600`).

Notes

- The function uses the `jsonwebtoken` library's `sign` method to create the token.
- The token will expire based on the value set in `config.APP_TOKEN_EXP`.
- Ensure that the payload does not contain sensitive information unless it is encrypted or otherwise protected, as JWT payloads are only base64-encoded and not encrypted.

logger.ts

This module provides a pre-configured Winston logger instance for consistent and structured logging throughout the application.

Overview

The logger is set up to:

- Log messages at the `info` level and above.
- Output logs in JSON format with timestamps.
- Write logs to both the console and a file named `combine.log`.

Implementation

```
import { createLogger, format, transports } from "winston";


/**
 * The logger instance used for logging messages.
 *
 * Configuration:
 * - Level: "info" (logs at info, warn, error, etc.)
 * - Format: JSON with timestamp
 * - Transports:
 *    - Console: Outputs all logs to the console.
 *    - File: Writes all logs to 'combine.log'.
 */
const logger = createLogger({
  level: "info",
  format: format.combine(format.timestamp(), format.json()),
  transports: [
    // Write all logs with importance level of `info` or less to
`combine.log`
    new transports.File({ filename: "combine.log" }),
    new transports.Console({}),
  ],
});
```

```
export default logger;
```

Usage

Import and use the logger in any part of your application:

```
import logger from "api/src/utils/logger";


logger.info("Application started");

logger.error("An error occurred", { error });
```

Configuration Details

- Level:
  The logger captures all messages at the `info` level and above (`warn`, `error`, etc.).
- Format:
  Logs are formatted as JSON and include a timestamp for each entry.
- Transports:
  - File: All logs are written to `combine.log` in the root directory.
  - Console: All logs are also output to the console for real-time visibility.

`api/src/utils/lowdb-lodash.utils.ts`

Overview

This file defines a utility class, `LowWithLodash`, which extends the LowDB `Low` class and integrates Lodash chainable methods for convenient data manipulation. The class is generic and can be used with any data type supported by LowDB.

Imports

```
import lodash from "lodash";

import { Low } from "lowdb";
```

- **lodash: Provides utility functions for common programming tasks, including chainable data manipulation.**

- **Low: The base class from LowDB, a small local JSON database for Node, Electron, and browser.**

Class: `LowWithLodash<T>`

Description

`LowWithLodash` is a generic class that extends the LowDB `Low` class, adding a Lodash-powered `chain` property. This property allows you to perform complex, chainable queries and transformations on the database's data using Lodash's API.

Type Parameters

- **`T`: The type of the data stored in the LowDB instance.**

Properties

- **`chain: lodash.ExpChain<this["data"]>`**
    - **A Lodash chain object initialized with the database's data.**
    - **Enables chainable Lodash operations directly on the data.**

Example Usage

```
import LowWithLodash from "./utils/lowdb-lodash.utils";

// Assume MyDataType is the shape of your database
const db = new LowWithLodash<MyDataType>(adapter);

// Access Lodash chain methods on the data
const result = db.chain
  .filter(item => item.active)
  .map(item => item.name)
  .value();
```

Implementation

```
export default class LowWithLodash<T> extends Low<T> {
  chain: lodash.ExpChain<this["data"]> = lodash.chain(this).get("data");
}
```

- The `chain` property is initialized to a Lodash chain starting from the `data` property of the LowDB instance.
- This allows you to use Lodash's chainable methods (e.g., `filter`, `map`, `find`, etc.) on your database data.

Notes

- The class assumes that the `data` property is always available and up-to-date. If you modify the data directly, you may need to re-initialize the chain to reflect changes.
- This utility is especially useful for projects that require both persistent storage (via LowDB) and advanced data querying/manipulation (via Lodash).

Exports

- Default: `LowWithLodash<T>`

market-app.utils.ts

Utility functions for interacting with the Contentstack Marketplace API, including fetching all apps for an organization and retrieving app manifest/configuration details.

Dependencies

- `@contentstack/marketplace-sdk`: SDK for interacting with Contentstack Marketplace.
- `DEVURLS`: A mapping of region codes to API host URLs, imported from `../constants/index.js`.

Functions

1. **`getAllApps`**

Fetches all marketplace apps available to a given organization.

Signature

```
export const getAllApps = async ({ organizationUid, authtoken, region }: any) => { ... }
```

Parameters

- `organizationUid` (`string`): **The unique identifier for the organization.**
- `authtoken` (`string`): **The authentication token for API access.**
- `region` (`string`): **The region code (e.g., `'NA'`, `'EU'`). Used to select the appropriate API host.**

Returns

- `Promise<Array<any> | undefined>`: **Resolves to an array of app objects (`data.items`) if successful, or `undefined` if an error occurs.**

Example Usage

```
const apps = await getAllApps({
  organizationUid: 'org_uid',
  authtoken: 'your_auth_token',
  region: 'NA'
});
```

Notes

- If an error occurs, it is logged to the console and the function returns `undefined`.

---

2. `getAppManifestAndAppConfig`

Fetches the manifest and configuration for a specific app in the marketplace.

Signature

```
export const getAppManifestAndAppConfig = async ({ organizationUid, authtoken, region, manifestUid }: any) => { ... }
```

Parameters

- `organizationUid` (`string`): The unique identifier for the organization.
- `authtoken` (`string`): The authentication token for API access.
- `region` (`string`): The region code (e.g., `'NA'`, `'EU'`). Used to select the appropriate API host.
- `manifestUid` (`string`): The unique identifier for the app manifest.

Returns

- `Promise<any | undefined>`: Resolves to the app manifest and configuration object if successful, or `undefined` if an error occurs.

Example Usage

```
const appDetails = await getAppManifestAndAppConfig({
  organizationUid: 'org_uid',
  authtoken: 'your_auth_token',
  region: 'NA',
  manifestUid: 'manifest_uid'
});
```

Notes

- If an error occurs, it is logged to the console and the function returns `undefined`.

Error Handling

Both functions log errors to the console using `console.info` and return `undefined` if an error is encountered.

Host Selection

The API host is determined by the `region` parameter using the `DEVURLS` mapping. If the region is not found, it defaults to the `'NA'` (North America) host.

CONTENTSTACK

Example

```
import { getAllApps, getAppManifestAndAppConfig } from
'./market-app.utils';


(async () => {

  const apps = await getAllApps({ organizationUid: 'org_uid', authtoken:
'token', region: 'NA' });

  console.log(apps);


  const appConfig = await getAppManifestAndAppConfig({

    organizationUid: 'org_uid',

    authtoken: 'token',

    region: 'NA',

    manifestUid: 'manifest_uid'

  });

  console.log(appConfig);

})();
```

pagination.utils.ts (src/utils/pagination.utils.ts)

Overview

This utility provides a function to fetch all paginated data from an API endpoint that supports `limit` and `skip` query parameters. It repeatedly requests data in batches until all items are retrieved, handling errors and response validation along the way.

Function: `fetchAllPaginatedData`

Fetches all items from a paginated API endpoint by making repeated requests until all data is collected.

Signature

```
const fetchAllPaginatedData = async (
  baseUrl: string,
  headers: Record<string, string>,
  limit = 100,
  srcFunc = '',
  responseKey = 'items'
): Promise<any[]>
```

Parameters

baseUrl (`string`):

- **The base URL of the API endpoint (e.g., `https://api.example.com/resources`).**

headers (`Record<string, string>`):

- **An object containing HTTP headers to include in each request (e.g., authentication tokens).**

limit (`number`, optional, default: `100`):

- **The number of items to fetch per request/page.**

srcFunc (`string`, optional, default: `''`):

- **The name of the source function for logging or error reporting purposes.**

responseKey (`string`, optional, default: `'items'`):

- **The key in the API response object that contains the array of items to collect.**

Returns

- `Promise<any[]>`:
  A promise that resolves to an array containing all items fetched from the paginated endpoint.

Usage Example

```
import fetchAllPaginatedData from './pagination.utils';


const allUsers = await fetchAllPaginatedData(
  'https://api.example.com/users',
  { Authorization: 'Bearer <token>' },
  50,
  'getAllUsers',
  'users'
);
console.log(allUsers.length); // Total number of users fetched
```

How It Works

Initialization:

1. Starts with an empty array and a `skip` **counter set to 0.**
2. **Fetching Loop:**
   - **Makes a GET request to the endpoint with** `limit` **and** `skip` **as query parameters.**
   - **Uses the** `safePromise` **utility to handle promise resolution and errors.**
   - **Extracts the array of items from the response using** `responseKey`**.**
   - **Appends the fetched items to the result array.**
   - **If the number of items fetched is less than** `limit`**, the loop ends (all data fetched).**

- **Otherwise, increments `skip` by `limit` and repeats.**

3. **Error Handling:**
    - **Throws an error if the request fails or if the response does not contain an iterable array at `responseKey`.**

Error Handling

- Throws an error if:
    - The HTTP request fails (includes error details and the source function name).
    - The response does not contain an array at the specified `responseKey`.

Dependencies

- **`safePromise`: Utility for handling async/await errors.**
- **`https`: HTTP request utility (must support the same interface as Axios or similar).**

Notes

- The endpoint must support **`limit` and `skip` query parameters for pagination.**
- **The function is generic and can be used for any paginated resource as long as the response structure is consistent.**

sanitize-path.utils.ts

Utility functions for sanitizing filenames and securely resolving file paths.

Functions

`sanitizeFilename(filename: string): string`

Sanitizes a filename by removing unsafe characters.

Only allows alphanumeric characters, underscores (`_`), dots (`.`), hyphens (`-`), and spaces.

Parameters:

- `filename` **— The input filename to sanitize.**

Returns:

A safe, sanitized filename string.

Example:

```
sanitizeFilename('my*unsafe:file.txt'); // 'myunsafefile.txt'
```

`getSafePath(inputPath: string, baseDir?: string): string`

Resolves and validates a safe, absolute file path.

Supports absolute and relative paths, as well as `path.join()` and `path.resolve()` usage.

Ensures the filename is sanitized and, if a base directory is provided, the resulting path does not escape it.

Parameters:

- `inputPath` **— The file path (absolute or relative) to resolve and sanitize.**
- `baseDir` **(optional) — The base directory for resolving relative paths and enforcing directory containment.**

Returns:

A safe, absolute file path as a string.

If the resolved path attempts to escape the base directory, returns a default path (`default.log` in the base directory).

Example:

```
getSafePath('../etc/passwd', '/var/log'); // '/var/log/default.log'
```

```
getSafePath('user data.log', '/var/log'); // '/var/log/user data.log'

getSafePath('/tmp/unsafe*file.txt');      // '/tmp/unsafefile.txt'
```

test-folder-creator.utils.ts

Utility functions for managing, sanitizing, and organizing test data folders and files, primarily for migration or testing scenarios.

Handles content types, assets, global fields, and directory cleanup.

Exports

`testFolderCreator({ destinationStackId })`

Main function to process and organize test data for a given stack ID.

Parameters:

- `destinationStackId` — **The stack identifier (string or object with a string property).**

Behavior:

- **Reads all entry files from the stack's entries directory.**
- **Normalizes and aggregates entry data by content type and locale.**
- **Processes only a subset of entries per content type for efficiency.**
- **Sorts and cleans up asset files.**
- **Deletes the original entries directory.**
- **Writes sanitized and organized entry files back to disk.**

Example:

```
await testFolderCreator({ destinationStackId: 'my-stack-id' });
```

Internal Utilities

`writeOneFile(indexPath, fileMeta)`

Writes a single file as JSON to the specified path.

`writeFiles(entryPath, fileMeta, entryLocale, locale)`

Ensures the target directory exists, then writes both the master file and locale-specific entry file.

`startsWithNumber(str)`

Checks if a string starts with a number.

`uidCorrector({ uid })`

Sanitizes UIDs by replacing spaces and hyphens with underscores, converting to lowercase, and prefixing with `a_` if the UID starts with a number.

`saveContent(ct, contentSave)`

Saves a content type object as a JSON file and appends it to a schema file in the specified directory.

`cleanDirectory(folderPath, foldersToKeep)`

Deletes all subfolders in a directory except those whose names are in the `foldersToKeep` list.

`deleteFolderAsync(folderPath)`

Recursively deletes a folder and its contents.

`lookForReference(field, finalData)`

Recursively processes schema fields to update references based on available content types.

`sortAssets(baseDir)`

Sorts asset metadata, cleans up asset files, and updates the asset schema file.

`writeGlobalField(schema, globalSave, filePath)`

Writes global field schema data to the specified file path, ensuring the directory exists.

`sortGlobalField(baseDir, finalData)`

Processes and updates global field references based on available content types.

CONTENTSTACK

watch.utils.ts

Utility for watching a log file and merging its updates into a destination file in real time.

Exports

`watchLogs(sourceFile: string, destinationFile: string): Promise<void>`

Watches a source log file for changes and appends new content to a destination log file whenever the source is updated.

Parameters:

- `sourceFile` — **Path to the log file to watch.**
- `destinationFile` — **Path to the log file where updates will be merged.**

Behavior:

- **Uses the `chokidar` library to efficiently watch for changes in the source file.**
- **On every change, reads the entire source file and appends its content (with a newline) to the destination file.**
- **Logs actions and errors to the console for traceability.**

Example:

```
import watchLogs from './watch.utils';


await watchLogs('logs/source.log', 'logs/merged.log');
```

Internal Functions

`mergeLogs(destinationFile: string, sourceFile: string): Promise<void>`

Reads the content of the source file and appends it (with a newline) to the destination file.

Handles errors gracefully and logs success or failure.

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

CONTENTSTACK

affix-confirmation.validator.ts

Validator for the `affix_confirmation` field in API requests, using express-validator.

Overview

This module exports a validation schema that ensures the `affix_confirmation` field is present in the request body and is a boolean value.

Validation Rules

- **Field: `affix_confirmation`**
    - **Location: `body`**
    - **Type: Must be a boolean (`true` or `false`)**
    - **Error Message: Uses the template from `VALIDATION_ERRORS.BOOLEAN_REQUIRED`, replacing `$` with `affix_confirmation`**
    - **Bail: Stops running further validations if this one fails**

Usage Example

```
import affixConfirmationValidator from './affix-confirmation.validator';


app.post('/your-endpoint', affixConfirmationValidator, (req, res) => {

  // Your handler logic here

});
```

If the `affix_confirmation` field is missing or not a boolean, the request will fail validation and return an appropriate error message.

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

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

auth.validator.ts

Validator for authentication request bodies using express-validator.

Overview

This module exports a validation schema to ensure that authentication requests contain valid `email`, `password`, and `region` fields, with an optional `tfa_token` field. Each field is validated for type, format, and value constraints.

Validation Rules

`email`

- **Location: `body`**
- **Type: Must be a string**
    - ***Error Message:*** **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "Email"**
- **Format: Must be a valid email address**
    - ***Error Message:*** **Uses `VALIDATION_ERRORS.INVALID_EMAIL`**
- **Trim: Removes leading/trailing whitespace**
- **Length: Must be between 3 and 350 characters**
    - ***Error Message:*** **Uses `VALIDATION_ERRORS.EMAIL_LIMIT`**

`password`

- **Location: `body`**

- **Type: Must be a string**
    - ***Error Message:*** **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "Password"**
- **Trim: Removes leading/trailing whitespace**

`region`

- **Location: `body`**
- **Type: Must be a string**
    - ***Error Message:*** **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "Region"**
- **Trim: Removes leading/trailing whitespace**
- **Allowed Values: Must be one of the values in `CS_REGIONS`**
    - ***Error Message:*** **Uses `VALIDATION_ERRORS.INVALID_REGION`**

`tfa_token` (optional)

- **Location: `body`**
- **Type: Must be a string if provided**
    - ***Error Message:*** **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "2FA Token"**
- **Trim: Not applied**

Usage Example

```
import authValidator from './auth.validator';


app.post('/auth/login', authValidator, (req, res) => {
  // Your authentication logic here
})
```

If any required field is missing or invalid, the request will fail validation and return an appropriate error message.

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

cms.validator.ts

Validator for the `legacy_cms` field in API requests, using express-validator.

Overview

This module exports a validation schema that ensures the `legacy_cms` field in the request body is a non-empty string with a maximum length of 200 characters.

Validation Rules

- **Field: `legacy_cms`**
    - **Location: `body`**
    - **Type: Must be a string**
      Error Message: **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "legacy_cms"**
    - **Trim: Removes leading and trailing whitespace before validation**
    - **Length: Must be between 1 and 200 characters**
      Error Message: **Uses `VALIDATION_ERRORS.LENGTH_LIMIT` with "legacy_cms"**

Usage Example

```
import cmsValidator from './cms.validator';
```

```
app.post('/your-endpoint', cmsValidator, (req, res) => {
  // Your handler logic here
}
```

If the `legacy_cms` field is missing, not a string, or does not meet the length requirements, the request will fail validation and return an appropriate error message.

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

destination-stack.validator.ts

Validator for the `stack_api_key` field in API requests, using express-validator.

Overview

This module exports a validation schema that ensures the `stack_api_key` field in the request body is a non-empty string with a maximum length of 100 characters.

Validation Rules

- **Field: `stack_api_key`**
    - **Location: `body`**
    - **Type: Must be a string**
      Error Message: **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "stack_api_key"**
    - **Trim: Removes leading and trailing whitespace before validation**

- **Length: Must be between 1 and 100 characters**
  Error Message: **Uses `VALIDATION_ERRORS.LENGTH_LIMIT` with "stack_api_key"**

Usage Example

```
import destinationStackValidator from './destination-stack.validator';


app.post('/your-endpoint', destinationStackValidator, (req, res) => {
  // Your handler logic here
})
```

If the `stack_api_key` field is missing, not a string, or does not meet the length requirements, the request will fail validation and return an appropriate error message.

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

file-format.validator.ts

Validator for the `file_format` field in API requests, using express-validator.

Overview

This module exports a validation schema that ensures the `file_format` field in the request body is a non-empty string with a maximum length of 200 characters.

Validation Rules

- **Field:** `file_format`
    - **Location:** `body`
    - **Type: Must be a string**
      Error Message: **Uses** `VALIDATION_ERRORS.STRING_REQUIRED` **with "file_format"**
    - **Trim: Removes leading and trailing whitespace before validation**
    - **Length: Must be between 1 and 200 characters**
      Error Message: **Uses** `VALIDATION_ERRORS.LENGTH_LIMIT` **with "file_format"**

Usage Example

```
import fileFormatValidator from './file-format.validator';


app.post('/your-endpoint', fileFormatValidator, (req, res) => {
  // Your handler logic here
});
```

If the `file_format` **field is missing, not a string, or does not meet the length requirements, the request will fail validation and return an appropriate error message.**

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

fileformat-confirmation.validator.ts

Validator for the `fileformat_confirmation` **field in API requests, using express-validator.**

Overview

CONTENTSTACK

This module exports a validation schema that ensures the `fileformat_confirmation` field in the request body is a boolean value.

Validation Rules

- **Field: `fileformat_confirmation`**
    - **Location: `body`**
    - **Type: Must be a boolean**
      Error Message: **Uses `VALIDATION_ERRORS.BOOLEAN_REQUIRED` with "fileformat_confirmation"**

Usage Example

```
import fileformatConfirmationValidator from
'./fileformat-confirmation.validator';


app.post('/your-endpoint', fileformatConfirmationValidator, (req, res) =>
{

  // Your handler logic here

});
```

If the `fileformat_confirmation` field is missing or not a boolean, the request will fail validation and return an appropriate error message.

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

CONTENTSTACK

index.ts

Centralized validator middleware for API request validation, using express-validator and custom error handling.

Overview

This module exports a function that returns an Express middleware for validating incoming requests based on the specified route. It dynamically selects the appropriate validator and throws a `ValidationError` if validation fails.

Supported Validators

The following validators are available and mapped by route name:

- `auth`: **Validates authentication fields**
- `project`: **Validates project-related fields**
- `cms`: **Validates CMS-related fields**
- `file_format`: **Validates file format fields**
- `destination_stack`: **Validates destination stack fields**
- `affix`: **Validates affix fields**
- `affix_confirmation_validator`: **Validates affix confirmation fields**
- `fileformat_confirmation_validator`: **Validates file format confirmation fields**
- `stack`: **Validates stack fields**

Usage Example

```
import validator from './validators';
```

```
app.post('/api/some-route', validator('file_format'), (req, res) => {
  // Your handler logic here
});
```

Pass the route name as a string to select the corresponding validator.

How It Works

1. **The middleware receives a route name and selects the corresponding validator from the internal mapping.**
2. **The selected validator is executed against the incoming request.**
3. **If validation errors are found, a `ValidationError` is thrown with the first error message.**
4. **If validation passes, the request proceeds to the next middleware or handler.**

Error Handling

If validation fails, a `ValidationError` is thrown with the first error message from the validation result. This should be caught by your global error handler to return an appropriate response to the client.

project.validator.ts

Validator for the `project` data in API requests, using express-validator.

Overview

This module exports a validation schema that ensures the `name` and `description` fields in the request body are non-empty strings with specific maximum lengths.

CONTENTSTACK

Validation Rules

- **Field:** `name`
    - **Location:** `body`
    - **Type: Must be a string**
      Error Message: **Uses** `VALIDATION_ERRORS.STRING_REQUIRED` **with "Name"**
    - **Trim: Removes leading and trailing whitespace before validation**
    - **Length: Must be between 1 and 200 characters**
      Error Message: **Uses** `VALIDATION_ERRORS.LENGTH_LIMIT` **with "Name"**
- **Field:** `description`
    - **Location:** `body`
    - **Type: Must be a string**
      Error Message: **Uses** `VALIDATION_ERRORS.STRING_REQUIRED` **with "Description"**
    - **Trim: Removes leading and trailing whitespace before validation**
    - **Length: Must be between 1 and 255 characters**
      Error Message: **Uses** `VALIDATION_ERRORS.LENGTH_LIMIT` **with "Description"**

Usage Example

```
import projectValidator from './project.validator';


app.post('/api/projects', projectValidator, (req, res) => {
  // Your handler logic here
});
```

If either the `name` **or** `description` **field is missing, not a string, or does not meet the length requirements, the request will fail validation and return an appropriate error message.**

CONTENTSTACK

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

---

stack.validator.ts

Validator for the `stack` data in API requests, using express-validator.

Overview

This module exports a validation schema that ensures the `name` and `description` fields in the request body meet specific type and length requirements.

Validation Rules

- **Field: `name`**
    - **Location: `body`**
    - **Type: Must be a string**
      Error Message: **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "Name"**
    - **Trim: Removes leading and trailing whitespace before validation**
    - **Length: Must be between 1 and 255 characters**
      Error Message: **Uses `VALIDATION_ERRORS.LENGTH_LIMIT` with "Name"**
- **Field: `description`**
    - **Location: `body`**
    - **Type: Must be a string**
      Error Message: **Uses `VALIDATION_ERRORS.STRING_REQUIRED` with "Description"**
    - **Trim: Removes leading and trailing whitespace before validation**

- **Length: Must be between 0 and 512 characters**
  Error Message: **Uses `VALIDATION_ERRORS.LENGTH_LIMIT` with "Description"**

Usage Example

```
import stackValidator from './stack.validator';


app.post('/api/stacks', stackValidator, (req, res) => {

  // Your handler logic here

});
```

If the `name` or `description` field is missing, not a string, or does not meet the length requirements, the request will fail validation and return an appropriate error message.

Returned Value

- **Returns an array of validation middlewares compatible with Express.js routes.**

database.ts

Handles the initialization and connection logic for the application's database directory.

Overview

This module provides an asynchronous function to ensure the required database folder exists before the application starts interacting with the database. It uses Node.js's `fs` module for file system operations and a custom logger for logging status and errors.

Function: `connectToDatabase`

Ensures the `./database` directory exists, creating it if necessary. Logs the connection status or any errors encountered.

Signature

```
const connectToDatabase: () => Promise<void>
```

Description

- **Checks if the `./database` directory exists.**
- **If it does not exist, creates the directory.**
- **Logs a success message upon successful setup.**
- **If an error occurs during this process, logs the error and terminates the process.**

Example Usage

```
import connectToDatabase from './database';


await connectToDatabase();

// Proceed with application startup
```

Error Handling

If an error occurs while checking for or creating the database directory, the function logs the error using the application's logger and exits the process with a non-zero status code.

server.ts

Main entry point for the API server. Sets up the Express application, middleware, routes, database connection, and real-time log streaming via Socket.IO.

Overview

CONTENTSTACK

This module initializes and configures the Express server, applies security and parsing middleware, sets up API routes, handles errors, and manages real-time log file updates to connected clients using Socket.IO. It also provides a utility to dynamically change the log file being watched.

Key Features

- **Security: Uses `helmet` for HTTP header security and `cors` for cross-origin requests.**
- **Request Parsing: Supports JSON and URL-encoded bodies up to 10MB.**
- **Custom Middleware: Includes logging, request header processing, and error handling.**
- **API Routing: Organizes endpoints for authentication, users, organizations, projects, content mapping, and migrations.**
- **Database Initialization: Ensures the database is ready before serving requests.**
- **Real-Time Log Streaming: Watches a log file and streams updates to clients via Socket.IO.**
- **Dynamic Log File Watching: Allows changing the watched log file at runtime.**

Middleware Stack

- **`helmet`: Sets security-related HTTP headers (with `crossOriginOpenerPolicy` disabled).**
- **`cors`: Enables CORS for all origins.**
- **`express.urlencoded` and `express.json`: Parses incoming request bodies.**
- **`loggerMiddleware`: Logs incoming requests.**
- **`requestHeadersMiddleware`: Processes custom request headers.**
- **`authenticateUser`: Protects routes that require authentication.**
- **`unmatchedRoutesMiddleware`: Handles 404s for undefined routes.**
- **`errorMiddleware`: Handles errors globally.**

CONTENTSTACK

API Routes

- `/v2/auth`: Authentication endpoints.
- `/v2/user`: User endpoints (requires authentication).
- `/v2/org/:orgId`: Organization endpoints (requires authentication).
- `/v2/org/:orgId/project`: Project endpoints within an organization (requires authentication).
- `/v2/mapper`: Content mapper endpoints (requires authentication).
- `/v2/migration`: Migration endpoints (requires authentication).

Real-Time Log Streaming

Watches the log file specified by `config.LOG_FILE_PATH` using `chokidar`. When the log file changes:

- **Reads new data from the file since the last update.**
- **Emits log updates in 1MB chunks to all connected Socket.IO clients via the `logUpdate` event.**

Example: Listening for Log Updates on the Client

```
const socket = io('http://localhost:PORT');

socket.on('logUpdate', (chunk) => {

  // Handle new log data

});
```

Dynamic Log File Path

The exported `setLogFilePath(newPath: string)` function allows changing the watched log file at runtime. It:

- **Validates and resolves the new path.**
- **Stops watching the old log file.**

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

- **Starts watching the new log file.**
- **Handles errors and can revert to the previous path if needed.**

Server Startup

On startup:

1. **Connects to the database using `connectToDatabase`.**
2. **Starts the Express server on the port defined in `config.PORT`.**
3. **Initializes Socket.IO for real-time communication.**

If any error occurs during startup, it is logged and the process exits.