CONTENTSTACK

React UI Implementation Guide

This document outlines the starting point and key components of the React app codebase. It serves as a guide for developers to understand the application's structure and architecture.

Starting Point

The application's entry point is typically the `App.js` file located in the `src` folder. This file renders the root component (`App`) into the DOM. Here is a representation of how this looks.

```javascript
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(
        <Suspense fallback={<FullPageLoader resourceName="Migration" />}>
            <Provider store={store}>
              <PersistGate
                loading={<FullPageLoader resourceName="Migration" />}
                persistor={persistor}
              >
                <AppLayout>
                  <AppRouter />
                </AppLayout>
              </PersistGate>
            </Provider>
          </Suspense>
        </ErrorBoundary>,
  document.getElementById('root')
);
```

The `App` component serves as the main container for the entire application. The texts showing on the UI side are getting fetched from the folder located src/cmsData.

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

CONTENTSTACK

Key Components

The React app uses several components to structure its UI and functionality. Below is a table that lists the components and their descriptions.

Component Name	Description
App	The main container for the entire application.
Login	Handles user authentication. Displays a login form where users input credentials and submit for validation.
Region page	Displays the card of regions available for login
Project	Displays the project dashboard page after login
Sidebar	Displays the settings page
Migration	Displays the migration stepper for each project

These are some of the primary components. The actual components might vary based on the application's requirements and complexity.

Folder Structure

The folder structure of the React app looks like the following:

- `src/`
  - `components/`
    - `Header/`
    - `Footer/`

        - MainContent/
        - Sidebar/
        - Form/
        - Button/
        - List/
        - Card/
    - assets/
        - images/
        - styles/
    - pages/
        - Home/
        - Login/
        - Sidebar/
        - RegionPage/
        - Projects/
        - Migration/
        - Errors/

    - utilities/
    - App.tsx
    - index.tsx

This structure organises components, assets, and other utility files for better maintainability.

Dependencies

The app uses the following key dependencies:

- React
- React DOM

Additional dependencies include:

- React Router

- Redux
- Venus components
- bootstrap

These libraries are essential for handling routing, state management, and API calls and components .

Detailed Component Overview:

Entry point:

- **Location**: ui/src/index.tsx
- **Type:** Functional Component
- **Purpose:** This file serves as the main entry point of the React application. It initialises the app, attaches it to the DOM, and sets up routing and global state management.

Renders the main `App` component wrapped in:

- `BrowserRouter` for client-side routing,
- `Provider` to supply the Redux store to the entire app.

`App.tsx` – Root Component Wrapper :

This file serves as the **main wrapper** around your entire application. It defines the **global context**, manages application-wide logic like error boundaries, Redux persistence, lazy loading, and network status checks.

- Custom hook that monitors online/offline status of the app

```
TypeScript
useEffect(() => {
```

```
    const selectModal = document.querySelector('.ReactModalPortal');
    if (selectModal instanceof HTMLElement && !isOnline) {
      selectModal.style.display = 'none';
    }
  }, [isOnline]);
```

Online: Loads the full application wrapped in error boundaries and global providers.
Offline: Displays a friendly network error message.

`AppRouter.tsx` – Application Routing Configuration

Handles the **entire routing logic** of the application using React Router. It defines public and protected routes, implements lazy loading for performance optimization, and renders appropriate pages based on the URL.

Key Features:

- Improves performance by loading page components only when needed.
- Supports dynamic parameters such as `projectId` and `stepId` for nested routing.
- A catch-all route (`path="*"`) to show a custom error page when the route doesn't exist.
- Prevents access to internal pages unless authenticated

Routes used in this application are:

Route Path	Component	Access Type

`/`	`Home`	Public
`/region-login`	`RegionalLogin`	Public
`/login`	`Login`	Public
`/projects`	`Projects`	Protected
`/projects/:projectId/settings`	`Settings`	Protected
`/projects/:projectId/migration/steps/:stepId`	`Migration`	Protected
`*`	`ErrorPage`	Fallback

Home.tsx – Home Page Component

This component serves as the **landing page** for the application. It pulls content from offline CMS data, blocks browser navigation (back/forward), and displays a button to login with contentstack.

Key Responsibilities

1. **Fetching Offline CMS Content**
   On initial render, the component fetches homepage content from a CMS data file using the `getCMSDataFromFile()` function. This content includes fields like `heading`, `description`, and a `CTA` object with title and URL.

2. **Handling Navigation Control**
   The component integrates two custom hooks:

   - `useBlockNavigation(true)` prevents the user from leaving the page unintentionally.
   - `usePreventBackNavigation()` disables back button behaviour to prevent navigating to a previous route.

3. **Rendering CMS Content**
   Once the data is fetched:

   - The `heading` is displayed in a header element.
   - The `description`, which might contain HTML formatting, is parsed using `html-react-parser` and rendered accordingly.
   - If a CTA is provided, it's rendered as a styled button inside a React Router `<Link>` that navigates to the provided URL.

4. **Styling and Layout**
   The layout is vertically and horizontally centred using utility classes, ensuring it looks visually appealing regardless of screen size.

Login Component:

Location : src/pages/Login

This `Login` component is a React functional component that handles user login with Two-Factor Authentication (TFA), built using `react-final-form`, Redux, and `react-router-dom`.

- Login Data and CMS Integration
    - It uses a function `getCMSDataFromFile` to load login page content dynamically (like heading, subtitle, labels, etc.) from CMS data (`CS_ENTRIES.LOGIN`).
    - The result is stored in state and used to populate UI fields like labels or button text.
- Routing and Redux
    - It uses `useNavigate` and `useLocation` from React Router to manage redirects and to read a `region` query param.
    - Redux's `useSelector` gets the `user` from the auth state, and `useDispatch` is used to update auth info on successful login (`setUser`, `setAuthToken`, etc.).

- Login Form Logic
        - `onSubmit` handles both first login and TFA scenarios:
            - Updates local login state (`loginStates`) with user info.
            - Calls `userSession` API.
            - Handles three cases:
                - **TFA required** (`status === 294`): shows TFA input.
                - **Login failed** (`status === 104/400/422`): shows error notification.
                - **Login success** (`status === 200`): stores token, updates Redux, redirects to `/projects`.

- Two-Factor Auth Support
    - When TFA is required, a different form shows up asking for a token (`tfa_token`).
    - Users can click a link to **send SMS** for the code. This calls `requestSMSToken`, and shows a notification if SMS is sent.

- Form Validations

    - Simple email, password, and TFA token validations are provided (`emailValidation`, `passwordValidation`, `TFAValidation`) using plain functions.
    - Back Button Handling
    - If the user is in the TFA step, the browser back button is blocked using
        - `window.history.pushState` to fake a history entry.
        - `popstate` event is intercepted to **redirect back to a safe route** (either `/region-login` or `/login?region=...`), ensuring users don't leave TFA mid-process.

Final Rendering

- UI comes from `<AccountPage>` and renders either:
    - TFA form (with TFA token input and SMS link), or
    - Email/password form (not included in the snippet but likely appears when `loginStates.tfa === false`)

Project : project dashboard

Location: src/pages/Projects
Interfaces used are stored in src/pages/Projects/projects.interface file

The `Projects` component is a **landing page** that displays a list of projects available for the currently selected organisation. It supports **search**, **project creation via modal**, and handles **empty states**. The data is fetched from both an internal API (`getAllProjects`) and a local CMS file (`getCMSDataFromFile`).

Redux and User Context

- Uses `useSelector` to get the current selected `selectedOrganisation` from the Redux `authentication` state.
- This value is used to fetch organisation-specific projects.

Project List & API Integration

- Fetches projects via `getAllProjects`, passing the selected organisation's ID.
- Maintains two states: `projects` (filtered/displayed) and `allProjects` (raw/full list).
- Also maintains `loadStatus` (loading spinner) and `searchText`.

Search Functionality

- Reads initial search query from URL on load using `useLocation`.
- Filters all `allProjects` by project name as the user types into the search box.
- Displays different empty states for no results vs. no projects.

Modal Support

- A modal for creating a new project is launched using the `cbModal` function.
- Modal content is passed using the `Modal` component, which receives dynamic content from CMS.
- After modal close, it refetches projects to reflect changes.

Back Button Handling

- Uses a custom `usePreventBackNavigation` hook.
- Prevents users from navigating away via the browser's back button while the modal is open or when needed.

UI & Rendering

- Header is rendered using the `ProjectsHeader` component with props like `headingText`, `searchText`, `setSearchText`, `cta`, etc.
- Projects are rendered using the `CardList` component.
- Skeleton loaders appear while data is loading.
- Empty states rendered using `EmptyState` with appropriate icons and messaging (for no projects or no search results).

CONTENTSTACK

Migration component

Location: src/pages/Migration

The `Migration` component orchestrates a multi-step migration flow for content migration projects. It manages state, interacts with Redux, fetches data from APIs, and coordinates the UI steps for migrating content from a legacy CMS to a new stack.

Features:

- Step Management: Handles navigation and state for each step in the migration process.
- Data Fetching: Retrieves project, CMS, stack, and file configuration data from APIs.
- Redux Integration: Reads and updates `migration`-related data in the Redux store.
- UI Coordination: Renders step-specific components and manages their completion state.
- User Feedback: Provides notifications and modals for user actions and errors.

Key State & References

- `projectData`: Holds the current project's migration data.
- `isLoading`: Indicates if data is being loaded or an action is in progress.
- `currentStepIndex`: Tracks the current step in the migration flow.
- `isCompleted`: Flags if the current step is completed.
- `isProjectMapper`: Controls when project mapping is in progress.
- `disableMigration`: Disables migration execution after starting.
- `isModalOpen`: Controls visibility of the save changes modal.
- `stepperRef`, `legacyCMSRef`, `saveRef`: Refs to child components for imperative actions

Main Steps in Migration Flow

1. Select Legacy CMS
    - Choose CMS, enter affix, and upload file.

    - Validates file format and affix.
2. Configure Destination Stack
    - Select the destination stack and map locales.
    - Validates stack selection and locale mapping.
3. Map Content Fields
    - Map legacy content types to new stack types.
    - Handles unsaved changes with a modal.
4. Run Test Migration
    - Initiates a test migration to validate mapping.
5. Execute Migration
    - Starts the final migration process.

Core Functions

- fetchData: Loads initial CMS flow data and updates Redux with flow steps.
- fetchProjectData: Loads project-specific data, including stack info, file config, and updates Redux state accordingly.
- fetchExistingContentTypes / fetchExistingGlobalFields: Fetches existing content types and global fields for the destination stack.
- getFileExtension/fetchFileFormat/getFileInfo: Utility functions to extract file info and format from config responses.
- createStepper: Returns an array of step definitions with associated components for the stepper UI.
- handleOnClick[Step]: Functions to handle "Continue" or "Save and Continue" actions for each step, including API updates and navigation.

Notifications & Modals

- Uses `Notification` from Venus Components to show user feedback on errors or required actions.
- Uses `cbModal` to show a save changes modal when unsaved changes are detected in content mapping.

Component structure

```
TypeScript
<MigrationFlowHeader />
<HorizontalStepper />
  ├─ <LegacyCms />
  ├─ <DestinationStackComponent />
  ├─ <ContentMapper />
  ├─ <TestMigration />
  └─ <MigrationExecution />
```

Components Directory Overview:

The `components/` directory contains **reusable, presentational UI components** used throughout the application. These are usually stateless or receive data via props and are designed for composability and reuse.Below is a conceptual breakdown of common component types and their purpose.

AccountPage

The `AccountPage` component serves as a **layout wrapper** for user account-related pages (e.g., login, sign-up, reset password).

Component Location

- File: `components/AccountPage/index.tsx`
- Styles: `components/AccountPage/index.scss`
- Interface: `components/AccountPage/accountPage.interface.ts`

Props:

```
TypeScript
interface AccountObj {
```

```
  children: React.ReactNode;
  data: LoginType;
}
```

AdvancedProperties

Location : src/components/AdvancePropertise
Interface: src/components/AdvancePropertise/advanceProperties.interface.ts
purpose : Used for customizing a field's behavior in a form-driven UI
Props :

```TypeScript
interface SchemaProps {
  fieldtype: string;
  value: UpdatedSettings;
  rowId: string;
  updateFieldSettings: (rowId: string, value: Advanced,
checkBoxChanged: boolean) => void;
  isLocalised: boolean;
  closeModal: () => void;
  data: FieldMapType;
  projectId?: string;
}
```

State Variables

- `options`: Options for dropdown or radio-type fields.
- `toggleStates`: Holds the boolean state of all switches and `default_value`.
- `showIcon`: Tracks index of the default option selected.
- `showOptions`: Controls visibility of dropdowns.
- `draggedIndex`: Used for tracking drag-and-drop reordering.

- `embedObjectsLabels`: Stores labels of selected embedded objects.

Functions and Their Purpose

Function Name	Purpose
`validateArray(arr)`	Utility function to check if an array is non-empty and valid.
`buildUpdatedSettings(updatedFields)`	Combines updated fields with current values and toggle state to form the new field settings payload for `updateFieldSettings`.
`handleToggleChange(field, value)`	Handles toggle switch changes (like `mandatory`, `multiple`, etc.), updates local state and parent field settings.
`updateDefaultValue(index, option)`	Updates `default_value` when a default dropdown item is selected.
`handleDrop(index)`	Handles drop action in drag-and-drop reordering of options. Updates order in local state and triggers `updateFieldSettings`.
`handleDragStart(index)`	Stores the index of the dragged item. Used for tracking during drag-and-drop.

`handleIconClick(index)`	Opens the dropdown menu to choose a new default value for a dropdown item.
`handleOptionSelect(labelArray)`	Called when multi-select dropdown changes. Updates `embedObjectsLabels` and sends selected UIDs to parent component via `updateFieldSettings`

CardList

The `CardList` component is a React functional component designed to display individual project cards within a list. Each card presents key information about a project, such as its name, source CMS, status, and last modified date. Additionally, it provides navigation functionality to the project's migration steps when you click on it.

Location: src/components/Card
Click Handler:
It handles the click event when a project card is clicked and navigates to the project migration steps page.
Prop used: param id- The ID of the project.

```typescript
const onClickProject = (id: string) => {
    if (isEmptyString(id)) return;

navigate(`/projects/${id}/migration/steps/${project?.current_step
}`);
  };
```

Status Mapping : Maps project status codes to corresponding icon names and CSS class names for styling

```typescript
const iconMapping: { [key: string]: string } = {
    '0': 'Information',
    '1': 'Warning',
    '2': 'Warning',
    '3': 'Warning',
    '4': 'Warning',
    '5': 'CheckCircleDark',
    '6': 'Close'
  };

  const statusClassMapping: { [key: string]: string } = {
    '0': 'draft',
    '1': 'pending',
    '2': 'pending',
    '3': 'pending',
    '4': 'pending',
    '5': 'completed',
    '6': 'failed'
  };
```

It renders below project information

- **Project Name**: Displayed with a tooltip for better UX.
- **Source CMS**: Shows the CMS associated with the project or a dash if unavailable.
- **Project Status**:
    - Icon and text representation based on the project's status.
    - Styled using the appropriate CSS class from `statusClassMapping`.
- **Last Modified Date**:
    - Uses the `getDays` utility to display how long ago the project was updated.

CONTENTSTACK

Content Mapper

The `ContentMapper` component is a core part of the migration flow, responsible for mapping content types and fields from a legacy CMS to the destination stack. It provides a UI for users to select, map, and manage content types and fields and integrates closely with Redux for state management.

Location: src/components/ContentMapper
Props:

- handleStepChange: Function to change the current step in the migration flow.
- ref: Exposes imperative methods (e.g., save actions) to parent components.

State & Context

- Redux State:
    - `migrationData`: General migration context and UI strings.
    - `newMigrationData`: Current migration progress and mappings.
    - `selectedOrganization`: The active organisation context.
- Local State:
    - `tableData`: Data for the mapping table (fields to be mapped).
    - `loading`, `isLoading`: Loading indicators for async actions.
    - `itemStatusMap`: Tracks the status of individual items (e.g., mapped/unmapped).
    - `totalCounts`: Number of items in the table.
    - `fieldValue`: Current field value being edited or mapped.
    - `searchText`, `searchContentType`: Search filters for content types.
    - `contentTypes`, `filteredContentTypes`: Lists of available content types.
    - `otherCmsTitle`, `otherCmsUid`: Identifiers for "other" CMS content types.
    - `isContentType`: Flag for content type vs. global field mapping.

- `contentModels`: List of content models from Redux (cloned for local use).
- `selectedContentType`: Currently selected content type for mapping.
- `existingField, deletedExstingField`: Track existing and deleted fields.
- `selectedOptions`: Selected options in dropdowns or multi-selects.
- `isDropDownChanged`: Tracks if dropdown selection has changed (for save prompts).
- `contentTypeMapped`: The current mapping between legacy and destination content types.
- `otherContentType`: State for "other" content type selection.
- `active`: Index of the currently active row or item.
- `rowIds, selectedEntries, initialRowSelectedData`: Track selected rows and entries.
- `contentTypeSchema`: Schema details for the selected content type.
- `showFilter`: Toggles filter UI visibility.
- `count`: General counter (e.g., for pagination or selection).
- `isModalOpen`: Controls modal visibility (e.g., save changes).
- `nestedList`: For nested field mapping scenarios.
- `isUpdated, isFieldDeleted, isContentDeleted, isCsCTypeUpdated`: Flags for various update/delete states.
- `isLoadingSaveButton`: Loading indicator for save actions.

Functions :

1. `fetchContentTypes`

Fetches the list of content types for a given project.

- Sets loading state.
- Calls the API to get content types.
- Updates state with the fetched content types, sets the first as selected, and fetches its fields.

2. `handleSearch`

Handles searching/filtering of content types by a search string.

- Updates the search filter state.
- Fetches filtered content types from the API and updates the list.

3. `fetchFields`
Fetches field mappings for a specific content type.

- Initialises item status as "loading".
- Calls the API to get field mappings.
- Updates item status to "loaded" after fetching.
- Filters and sets valid table data for mapping.

4. `fetchData`
Fetches table data (field mappings) for the currently selected content type, using a search filter if provided.

5. `loadMoreItems`
Implements "load more" functionality for paginated field mapping data.

- Sets the loading state for the specified range.
- Fetches additional field mappings and appends them to the table data.

6. `handleOpenContentType`
Handles switching between content types in the UI.

- If there are unsaved changes, it prompts the user to save before switching.
- Otherwise, switches to the selected content type.

7. openContentType
Switches to a specific content type by index.

- Updates selection state and fetches its fields.

8. updateFieldSettings
Updates advanced settings for a specific field in the mapping table.

- Marks dropdown as changed if a checkbox is toggled.
- Updates the table data and selected entries with new settings.

9. handleSchemaPreview
Opens a modal to preview the schema of a selected content type.

- Fetches schema data and displays it in a modal dialogue.

10. accessorCall
Renders field information for display in the mapping table, including type and UID.

11. getParentId
Finds the parent ID of a field by UID from the table data.

12. modifiedObj
Creates a simplified object representing a field, including parent ID if applicable.

13. getLastElements
Returns the last action performed on each row from a history object.

14. findLatest
Finds the most recent action performed across all rows.

15. updateRowHistoryObj
Updates the row history object when a row is selected or deselected.

16. handleSelectedEntries
Handles selection/deselection of rows in the mapping table.

- Updates row history and selection state.
- Handles group/child relationships for selection logic.

17. handleValueChange
Handles changes to dropdown values in the mapping table.

- Marks dropdown as changed.
- Updates table data and triggers a Redux state update.

18. handleDropDownChange
Handles changes to the selected destination content type in dropdowns.

19. handleAdvancedSetting
Opens a modal for advanced field settings for a specific row.

20. SelectAccessor
Renders a select dropdown for mapping fields, with options based on field type and schema.

21. handleFieldChange
Handles changes to mapped fields, including group/child logic and updating selection arrays.

22. generateSourceGroupSchema
Generates a nested group structure from the source CMS schema for use in mapping UI.

23. getMatchingOption
Utility function to create an option object for dropdowns, marking as disabled if already selected.

24. checkConditions
Utility function to check if a destination field matches source field requirements based on type.

25. processSchema
Recursively processes nested group structures in content type schemas to generate mapping options.

26. SelectAccessorOfColumn
Renders select dropdowns for each column in the mapping table, handling auto-mapping and advanced options.

27. handleSaveContentType
Saves the current content type mapping to the backend.

- Updates mapping state and notifies user on success or failure.

28. handleDropdownState
Resets dropdown change state after saving or discarding changes.

29. handleResetContentType
Resets all mappings for the selected content type to their initial state.

- Updates backend and local state, clears selections, and notifies user.

30. handleCTDeleted
Purpose: Resets the mapping and state when a content type or global field is deleted from the destination stack.

31. handleFetchContentType
Purpose: Fetches existing content types or global fields from the destination stack, and updates the mapping state accordingly.

Destination Stack

`DestinationStackComponent` is a React functional component used in a migration tool to manage and display step-by-step configurations for a "Destination Stack." It asks user for stack selection and language mapping.

Location: src/components/DestinationStack

Props :

```typescript
type DestinationStackComponentProps = {
  isCompleted: boolean;
  projectData: MigrationResponse;
  handleOnAllStepsComplete: (flag: boolean) => void;
};
```

State variables :

- `isMigrationLocked` (`boolean`)
  Determines whether the destination stack is editable or locked based on project status (`status === 2 || 5`).
- `stepperKey` (`string`)
  Static key used to force re-render the `AutoVerticalStepper`. Defaults to `'destination-Vertical-stepper'`.
- `internalActiveStepIndex` (`number`)
  Used to programmatically control which step is active in the stepper. Currently set to `-1`.
- `isLoading` (`boolean`)
  Tracks whether the component is still loading CMS data. Prevents rendering until data is ready.
- isProjectMapped (`boolean`)
  Tracks whether the current project has already been mapped to a destination stack, disabling the stepper if so.

Component flow:

Initial Data Fetching:

- Loads data from a local CMS file via `getCMSDataFromFile`.
- Updates migration data and sets a lock based on `projectData.status`.
- Ends with `setIsLoading(false)`

Dynamic Step Control (placeholder for future):

- If `internalActiveStepIndex` is set, dynamically changes the step using the ref.
- Currently not utilised (`internalActiveStepIndex` remains `-1`).

Project Mapping Status Sync:

- Watches `newMigrationData.isprojectMapped` and updates the internal `isProjectMapped` state.

getDestinationStackSteps:

This function dynamically processes an array of step objects (`allSteps`) for the Destination Stack Stepper, assigning UI components and statuses (active, completed, locked) based on the migration state.

Parameters:

- **`isCompleted`** (`boolean`):
  Whether all steps are completed.
- **`isMigrationLocked`** (`boolean`):
  Whether the project is locked from further editing (based on migration status).
- **`allSteps`** (`IStep[]`):
  Array of all step configuration objects coming from the CMS or default data.

Returns: A new array of steps with updated components (`data`), `status`, and `lock` state, ready for rendering inside the `AutoVerticalStepper`.

Load Organisation:

The `LoadOrganization` component is a **read-only step UI** in a multi-step migration workflow. It is responsible for displaying the **selected organisation** for the destination stack. This selection is either pulled from the existing migration data (`newMigrationData.destination_stack.selectedOrg`) or defaults to the globally selected organisation (`selectedOrganisation` from the authentication store).

Location: src/components/DestinationStack/Actions/LoadOrganisation.tsx

LoadStacks

The `LoadStacks` component is a React functional component used in a migration workflow that allows users to select or create a Contentstack "stack."

Location : src/components/DestinationStack/Actions/LoadStacks.tsx
Props:

```typescript
interface LoadFileFormatProps {
  stepComponentProps?: () => {};
  currentStep: number;
  handleStepChange: (stepIndex: number, closeStep?: boolean) =>
void;
}
```

Functions used :

1. fetchData():
    - Loads all stacks via `getAllStacksInOrg`.
    - Maps stacks into `IDropDown` format.
    - Sorts and updates the Redux store and local state (`allStack`, `selectedStack`).
    - Updates `csLocale` in `destination_stack`.
2. handleDropdownChange('stacks'): updates `selectedStack` and dispatches to Redux.
3. `handleCreateNewStack(): opens` a modal with `the AddStack` form.
4. handleOnSave(): is triggered from `AddStack`. Calls `createStacksInOrg()` and updates `selectedStack`

Language Mapper

This Mapper component is a React functional component designed to manage **language (locale) mappings** between two systems (presumably Contentstack and a source system) for a migration process.

Location : src/components/DestinationStack/Actions/LoadLanguageMapper.tsx

State variables:

- selectedMappings: Stores current locale mappings like { "en-us-master_locale": "fr-fr" }.
- existingField / existingLocale: Track the currently selected locales on each side.

- selectedCsOptions / selectedSourceOption: Arrays tracking selected values to prevent duplicates.
- csOptions / sourceoptions: Filtered dropdown options excluding already-selected ones.
- placeholder: Static placeholder text for selects.

Functions used:

- handleSelectedCsLocale(...) : Handles destination CMS locale selection:
  - Updates `existingField`
  - Updates selected list
  - Updates `selectedMappings` with destination key

- handleSelectedSourceLocale(...) : Handles source system locale selection:
  - Updates `existingLocale`
  - Ensures label is synced
  - Updates `selectedMappings` using `csLocaleKey`
  -
- handleLanguageDeletaion(...) : Triggered on removing a mapping:
  - Removes locale mapping from all states (`existingField`, `existingLocale`, `selectedMappings`)
  - Also calls `handleLangugeDelete` (external prop function)

Table Header

`The TableHeader` component renders two `FieldLabel` components: one labelled **"Contentstack"** and another labelled with the dynamic `CMS` prop.

Location : src/components/DestinationStack/Actions/tableHeader.tsx

Legacy CMS

The `LegacyCMSComponent` is a React component designed to facilitate the integration and configuration of legacy CMS data within a migration workflow.

Location : src/components/LegacyCms
Props :

```typescript
type LegacyCMSComponentProps = {
  legacyCMSData: LegacyCmsData;
  isCompleted: boolean;
  handleOnAllStepsComplete: (flag: boolean) => void;
};
```

State Variables

- isMigrationLocked: boolean
  Determines if the migration process is locked based on project status.
- isLoading: boolean
  Indicates if the component is in a loading state.
- internalActiveStepIndex: number
  Tracks the current active step in the stepper.
- stepperKey: string
  Unique key for the stepper component to manage its state.
- isAllStepsCompleted: boolean
  Flags whether all steps in the migration process are completed.
- isProjectMapped: boolean
  Indicates if the project has been mapped in the new migration data.

LoadSelectCms

This component is a React functional component used in a stepper workflow to **load and select a legacy CMS** from a list based on configurations fetched from an API.

Location : src/components/Steps/LoadSelectCms/LoadSelectCms.tsx

The `LoadSelectCms` component is responsible for

1. Fetching and displaying CMS options.

2. Allowing the user to select one CMS.
3. Auto-selecting a CMS if only one match is found.
4. Updating the Redux state with the selected CMS.
5. Triggering step transition on selection.

Props :

```typescript
interface LoadSelectCmsProps {
  stepComponentProps?: () => {};
  currentStep: number;
  handleStepChange: (stepIndex: number, closeStep?: boolean) =>
void;
}
```

Functions used :

1. `handleCardClick(data: ICMSType)`
Triggered when a CMS card is clicked.
- Updates selected card in local state.
- Dispatches updateNewMigrationData to Redux.
- Calls handleStepChange to move to the next step.

2. `filterCMSData(searchText: string)`
Handles filtering of CMS options.
- Retrieves all CMS from migrationData.legacyCMSData.
- Calls getConfig() to determine default cmsType.
- Filters CMS data using:
  - CMS parent match
  - Search text match (title or cms_id)
- If only one CMS remains, it's selected by default.
- Dispatches new migration data to Redux and triggers step change.

This component returns error state if `isError` is true, shows loader if `isLoading` is true.
Otherwise, renders CMS options as `Card` components. Cards are clickable unless already selected or disabled by step.

LoadPreFix

The `LoadPrefix` component allows users to input a prefix (called "affix") during a migration process. It validates the prefix according to specific business rules, including restrictions against keywords and invalid characters. It also integrates with Redux to store and manage the affix value.

Location : src/components/legacyCms/LoadPrefix.tsx

Props :

```typescript
interface LoadSelectCmsProps {
  currentStep: number;
  handleStepChange: (stepIndex: number, closeStep?: boolean) =>
void;
}
```

Functions used:

1. fetchRestrictedKeywords: Fetches a list of restricted keywords from the backend using an API and stores them in component state. It internally uses `getRestrictedKeywords()` service and updates `restrictedKeywords` state with an array of strings if the API call is successful.
2. handleOnChange: Handles user input changes in the affix `TextInput`, applies validation, and updates the Redux store accordingly.

CONTENTSTACK

LoadFileFormat

`LoadFileFormat` is a UI component that reads the uploaded file format and displays it with an appropriate icon. It performs validation against allowed CMS formats and triggers a step change in a migration wizard.

Location : src/components/LegacyCms/Actions/LoadFileFormat.tsx

Props :

```typescript
interface LoadSelectCmsProps {
  currentStep: number;
  handleStepChange: (stepIndex: number, closeStep?: boolean) =>
void;
}
```

State variables used :

Variable	Type	Purpose
`selectedCard`	`ICardType`	Stores the selected file format object from Redux
`isCheckedBoxChecked`	`boolean`	Stores checkbox status; defaults to true if not defined
`fileIcon`	`string`	The name of the file format to be shown as an icon
`isError`	`boolean`	Controls visibility of an error message
`error`	`string`	Stores the actual error message

Refs

Ref	Purpose

`newMigrationDataRef`	Stores a mutable reference to `newMigrationData`

Functions used:

1. `handleBtnClick`

```
Unset
const handleBtnClick = async () => {
  if (!isEmptyString(selectedCard?.fileformat_id) && isCheckedBoxChecked)
{
    dispatch(updateNewMigrationData({...}));
    props.handleStepChange(props?.currentStep);
  }
};
```

Purpose

- Validates that a file format is selected and checkbox is checked.
- Updates Redux store and advances to the next step.

Side Effects

- Dispatches updateNewMigrationData
- Calls handleStepChange

2. `handleFileFormat`

Purpose

- Determines the current file format from uploaded file data.
- Validates it against allowed formats for the selected CMS.
- Sets an error if invalid or updates the icon label otherwise.

Key Logic

- Uses `filePath`, `selectedCms`, and `all_cms` from Redux
- Checks `allowed_file_formats` against detected file extension

- If file is ZIP, formats it to `Zip`; otherwise uses uppercase label

Side Effects

- Updates `fileIcon`, `isError`, and `error` state

LoadUploadFile

The component manages the process of uploading a file, ensuring that the file is properly validated before proceeding. It provides visual feedback to the user about the status of the validation, such as progress percentages and validation errors or success messages.

Location: src/components/LegacyCms/Actions/LoadUploadFile.tsx

Props :

```typescript
interface LoadSelectCmsProps {
  currentStep: number;
  handleStepChange: (stepIndex: number, closeStep?: boolean) =>
void;
}
```

The user flow :

1. The user selects a file to upload.
2. The component validates the file type and format.
3. If valid, the file is uploaded with progress feedback displayed to the user.
4. State, progress, and validation status are stored in `sessionStorage` for persistence.
5. The component interacts with Redux to update global state regarding the file upload status.
6. Once the file is uploaded, messageis displayed to the user, and the process continues.

Functions used:

1. getConfigDetails(): It calls the config API from the upload service and extracts the file extension from the file path by calling the getFileExtension().
2. handleOnFileUploadCompletion(): This function is called when the user validates the file. It is integrated with the validation API from the upload service. Upon validation, it stores the values in Redux and session storage as well.

LogScreen

`TestMigrationLogViewer` is a React functional component that connects to a WebSocket server to listen for real-time migration log updates. It displays these logs with formatting based on log level (info, success, warning, error), handles zoom and scroll controls, and updates the Redux store and localStorage based on migration state.

Location: src/components/LogScreen
Props:

State Variable	Type	Purpose
`isLogsLoading`	`boolean`	Indicates if logs are being received.
`logs`	`LogEntry[]`	Array of parsed log entries displayed in the viewer.
`migratedStack`	TestStacks	Holds the current test stack status.
`zoomLevel`	`number`	Controls the zoom level for log display.

Functions used:

`useEffect` for Stack Info Update

Updates `migratedStack` when `test_migration` changes in `newMigrationData`.

`useEffect` for WebSocket Connection

- Initialises connection to the server using `socket.io-client`.
- Listens to the `logUpdate` event to receive logs.
- Parses and stores new log entries.
- Handles reconnection logic on disconnection.

`handleScrollToTop` and `handleScrollToBottom`

Scrolls the logs container to the top or bottom smoothly.

`handleZoomIn` and `handleZoomOut`

Controls the zoom level of the logs container within defined limits (0.6 to 1.4).

`useEffect` on `logs`

- Auto-scrolls to the bottom on new logs.
- Detects `Test Migration Process Completed` log to:
    - Mark test migration as completed.
    - Save state to localStorage.
    - Notify parent.
    - Update the Redux state.

`useEffect` on `isLogsLoading` and `migratedStack`

Resets logs if no logs are loading and the stack is not migrated.

MainHeader

The `MainHeader` component is a **React functional component** that serves as the **main header for a migrationstepper**. It incorporates **branding, organisation selection, user profile access, and guarded navigation** logic, all of which are essential for the application's structure and usability.

Location : src/components/MainHeader

Functions used :

1. updateOrganisationListState()
    - Maps through `organisationsList`
    - Marks selected org as default
    - Saves selected org to localStorage

2. fetchData()
    - Loads CMS data for `MainHeader`
    - Sets logo and other config

3. handleOnDropDownChange(data)
    - If new selection differs from current, updates the selected org in Redux and localStorage
4. handleonClick()
    - Main click handler for logo
    - Gets current project step from backend
    - If conditions met, opens confirmation modal before navigation
    - Otherwise, resets migration state and navigates to `/projects`

UI Structure

1. **Logo (left-aligned)**
    - Click opens modal or navigates to `/projects`
    - Tooltip: “Projects”

2. **Organization Switcher**
    - Dropdown appears only on `/projects`

3. **User Profile (right-aligned)**
    - Appears on `/projects` or `/projects/...` paths
    - Uses `ProfileCard` inside dropdown

CONTENTSTACK

MigrationExecution

The `MigrationExecution` component is a crucial part of the migration workflow in the application. It serves to:

- Display configured migration details such as Legacy CMS, Organization, Stack, and Locale.
- Provide a visual representation of the migration process through execution logs.

Location : src/components/MigrationExecution

Props:

handleStepChange

- Type: `(currentStep: number) => void`
- Description: A callback function to handle the change in the current step of the migration process.

Helper function :

getPlaceHolder

- **Parameters**: `title: string`
- **Returns**: A string representing the placeholder value based on the provided title.
- **Logic** :
    - Matches the title to corresponding fields in `newMigrationData`.
    - Returns the appropriate label or value for display.

Rendering Logic

- Loading State:
    - If `isLoading` is `true` or `newMigrationData?.isprojectMapped` is `true`, displays a `CircularLoader`.

- Main Content:
    - Displays a message indicating that the migration configurations are set.
    - Iterates over `MigrationInformation` to display configured details:
        - Each item is displayed within a `Field` component.
        - Uses `Tooltip` to show the full value on hover.
        - Displays an `ArrowRight` icon between fields for visual flow.
    - Renders the `MigrationLogViewer` component to show execution logs.

MigrationFlowHeader

The `MigrationFlowHeader` component serves as the header section for the migration flow interface within the application. It displays the current project's name and provides a call-to-action (CTA) button that allows users to proceed to the next step in the migration process. The component dynamically adjusts the CTA button's label and disabled state based on the current migration step and the project's status.

Location : src/components/MigrationFlowHeader

Props:

```TypeScript
type MigrationFlowHeaderProps = {
  handleOnClick: (event: MouseEvent, handleStepChange:
(currentStep: number) => void) => void;
  isLoading: boolean;
  isCompleted: boolean;
  legacyCMSRef: React.MutableRefObject<any>;
  projectData: MigrationResponse;
  finalExecutionStarted?: boolean;
};
```

Hooks and Selectors

- `useNavigate`: Provides navigation capabilities to programmatically change routes.
- `useParams`: Accesses URL parameters, specifically `projectId` and `stepId` in this context.
- `useSelector`: Retrieves data from the Redux store:
    - `selectedOrganisation`: The organization currently selected by the user.
    - `newMigrationData`: Contains the latest migration data, including statuses and configurations.

The component renders a header containing:

- **Project Name**: Displayed with a tooltip for full visibility, especially if the name is truncated.
- **CTA Button**: Labeled appropriately based on the current step and disabled based on the conditions outlined above.

ProfileCard:

The `ProfileCard` component displays the currently logged-in user's profile information, including their initials, name, email, and region. It also provides a logout button that clears local storage and redirects the user to the login page.

Location: src/components/ProfileHeader

Logout function :

```typescript
const handleLogout = () => {

  if (clearLocalStorage()) { navigate('/', { replace: true });

  }};
```

- Clears local storage via clearLocalStorage().
- Redirects to the root path (/), usually the login page.

ProjectsHeader

The `ProjectsHeader` component provides a standardized header for the "Projects" page. It includes :

- A title with an integrated search bar.
- A CTA (Call To Action) button for creating new projects.
- Logic to conditionally disable the create button based on migration status.

Location : src/components/ProjectsHeader

Props :

```typescript
export interface ProjectsHeaderType {
  cta?: CTA;
  restore_cta?: CTA;
  headingText: string | undefined;
  searchText: string;
  setSearchText: (value: string) => void;
  searchPlaceholder: string;
  handleModal?: () => void;
  allProject: ProjectsObj[] | null;
}
```

HorizontalStepper

The `HorizontalStepper` is a reusable, interactive component designed to guide users through a multi-step process in a horizontal layout. This component supports navigation control, step completion tracking, and modal-based confirmation prompts. It's primarily used within a migration workflow UI.

Location : src/components/Stepper/HorizontalStepper

Props :

```typescript
export type stepperProps = {
  steps: Array<stepsArray>;
  className?: string;
  emptyStateMsg?: string | JSX.Element;
  hideTabView?: boolean;

  stepContentClassName?: string;
  stepTitleClassName?: string;
  testId?: string;
  handleSaveCT?: () => void;
  changeDropdownState: () => void;
  projectData: MigrationResponse;
  isProjectMapped: boolean;
};
```

Functions used :

1. handleTabStep(idx): Validates and navigates to the selected step:
    - If dropdown changes were made, prompts modal confirmation.
    - Validates legacy CMS setup and stack selection.
    - Displays warning notifications if required conditions are not met.
    - Calls `setTabStep(idx)` if validation passes.
2. setTabStep(idx) : Navigates to a step only if:
    - The step is either completed or is the next logical step.
    - The project is not mapped.
3. StepsTitleCreator: Renders a horizontal visual stepper:
    - Completed steps show a checkmark.
    - Active step is highlighted.
    - Future or invalid steps are disabled.

SchemaModal

The `SchemaModal` component provides a **modal-based UI** to preview a **nested schema structure** of a content type. It visualises fields using icons and supports nested groups (like Contentstack's "group" or "modular blocks").

Location: src/components/SchemaModal
Interface: src/components/SchemaModal/schemaModal.interface.ts
Styles: src/components/SchemaModal/index.scss

Libraries and Dependencies

- **React :** React hooks for managing component state and lifecycle `useState`, `useEffect`
- **Venus Components:** UI components `ModalBody`, `ModalHeader`, `Icon`.

Utility: getTopLevelIcons

Maps field types to corresponding Contentstack icons.

```typescript
const getTopLevelIcons = (field: FieldMapType) => { ... }
```

- **Purpose**: Returns the corresponding icon name for a field type using the `Icons` map.
- Handles different field types and variations, ensuring backward compatibility and flexibility.

Component: `TreeView`

Parses and renders a nested, expandable tree of schema fields.

```TypeScript
const TreeView = ({ schema = [] }: schemaType) => { ... }
```

- **Props**: Accepts a `schema` array (field definitions).
- **State**: `nestedList` holds a structured version of the schema, with nested fields grouped under parent fields (like groups).
- **Effect**: Transforms the flat schema list into a nested format for rendering.

Functions :

1. `hasNestedValue(field)`: Checks if a field has child elements.
2. `getChildFieldName(text, groupName)`: Removes group prefix from child field names for cleaner display.
3. `handleClick(event)`: Handles expand/collapse toggle logic and sets the active class.
4. `generateNestedOutline(item, index)`: Recursively renders child fields under parent nodes.

Component: `SchemaModal`

A modal container for rendering the schema outline.

```TypeScript
const SchemaModal = (props: SchemaProps) => { ... }
```

Props :

- `contentType: Name of the content type being previewed.`

- `schemaData`: Raw field schema data to be displayed.
- `closeModal`: Function to close the modal.

Test Migration

The `TestMigration` component is a React functional component that facilitates the testing phase of a CMS data migration project. It handles UI rendering, state management, stack creation, migration initialization, and interaction with Redux and backend APIs.

Render Logic:

- Displays a loader if the component is loading or project mapping is not done.
- Otherwise:
  - Shows buttons and tooltips for creating a test stack.
  - Displays test stack information and a migration start button.
  - Renders logs via `TestMigrationLogViewer` component.

Location: src/components/TestMigration
Interface: src/components/TestMigration/testMigration.interface.ts
Styles: src/components/TestMigration/index.scss

Libraries and Dependencies

- **React & React Router:** Component hooks and routing.
- **Venus Components:** UI components like `Tooltip`, `Button`, `Notification`,`TextInput`, `FieldLabel`, etc.
- **Redux:** State management using `useSelector` and `useDispatch`.
- **Custom Services & Utilities:** API services for CMS and stack handling, constants, and utility functions.

Redux Usage

- `useSelector`: Accesses `newMigrationData` and `selectedOrganisation`.
- `useDispatch`: Updates `newMigrationData` after stack creation or migration.

Local Storage

- Key: `testmigration_<projectId>`
- Stores migration progress flags like `isTestMigrationStarted` and `isTestMigrationCompleted`.

API Calls

- `getCMSDataFromFile` - Retrieves UI data.
- `getOrgDetails` - Fetches stack limits.
- `getAllStacksInOrg` - Gets existing stacks in the org.
- `createTestStack` - Creates a temporary stack for testing.
- `createTestMigration` - Starts the test migration process.

Error Handling

- API failures are logged to the console.
- Error messages shown via `Notification`.

Environment Variables

- `REACT_APP_BASE_API_URL`: Used as server path for the `TestMigrationLogViewer` component.

State Variables

- `data`: Stores CMS static content for button labels and text.
- isLoading: Controls initial loading spinner
- isStackLoading: Controls loader for stack creation.
- disableTestMigration: Disables Test Migration button based on conditions.
- disableCreateStack: Disables test stack creation button.
- stackLimitReached: Flag indicating whether the stack limit has been reached.

- isProjectMapped: Controls when project mapping is in progress.

useEffect Hooks

1. **Fetch CMS data:** Loads static UI text content from a local CMS file.
2. **Stack creation and migration logic:** Determines when buttons should be disabled based on the Redux state.
3. **State from Local Storage:** Retrieves and applies previous migration state from session/local storage.

Functions :

5. `handleCreateTestStack:`
   Checks current stack count and compares with organization limit.
   Shows warning if limit is reached.
   On success:
   - Calls backend API to create a test stack.
   - Updates Redux state with new stack data.
   - Saves state in local storage.
6. handleTestMigration:
   Triggers test migration by calling backend service.

   On success:

   - Updates Redux state.
   - Saves migration progress to local storage.
   - Send notification to the user.

7. formatErrorMessage:

   Converts error object into a readable string with field-level errors.

8. handleMigrationState:
   Updates button states in the parent based on the new migration state.

CONTENTSTACK

API Integration :

It is used to **organize reusable logic for communicating with external systems**, such as APIs, authentication providers, local storage, or other utilities that don't belong in UI components.
Location : src/services

Following are the services used:

File Name	Responsibility
`login.service.tsx`	Handles all user-related API calls (e.g., login, logout)
`user.service.ts`	Handles all user-related API calls (e.g. getProfile)
`project.service.tsx`	Handles CRUD operations for projects
`stacks.service.tsx`	Handles CRUD operations for stacks
`migration.service.tsx`	Migration-specific API's
`upload.service.tsx`	Handles config, validation related API's