# DEPRECATED: Standard-workflow (One-click)

This section describes the structure of the modules:

1. Standard-workflow
2. Standard-workflow-interface

This section also outlines the principles of creating workflows based on the manager + worker scheme

# System overview

# Overview

The workflow is designed according to the following principles:

1. **Manager** - a workflow that handles state files, specifically:
    1. Reads the list of state files from the `/states` folder.
    2. Sorts them by status.
    3. Initiates workers.
    4. Reads and copies the statuses of the workers.
    5. Updates the information in the state files.
2. **Worker** - a workflow that:
    1. Executes functional tasks.
    2. Reads, saves, and updates its own state file.
    3. Receives a payload from the manager in a specific format that helps customize the input data.

This principle is required to customize the list of repetitive steps in Finmars using JSON files (and provides the ability to execute this customization through an interface, which is currently not implemented), as well as to use the interface to analyze completed steps, identify errors, and restart steps.

# How It Works

The workflow manager (`com.finmars.standard-workflow:workflow-manager`) operates on a cron schedule (by default - every 1 minute) and performs the following actions:

1. Reads the `global_state_manager.json` file.
    1. If the file does not exist, it creates the file and adds files from `/states/managers` according to the following format:

```
{
   "to-do": [],
   "in-progress": [
      "com.finmars.standard-workflow:workflow-manager-20240821095322.json"
   ],
   "done": [
      "com.finmars.standard-workflow:workflow-manager-20240724101635.json",
```

```
        "com.finmars.standard-workflow:workflow-manager-20240724104102.json"
    ],
    "paused": []
}
```

2. Selects the first file from the in-progress array and begins managing it.

- Potentially, this could be improved by running multiple threads, but there are certain limitations since these state files are independent of each other (see the input file requirements).

3. The manager goes through the workers and checks their statuses:

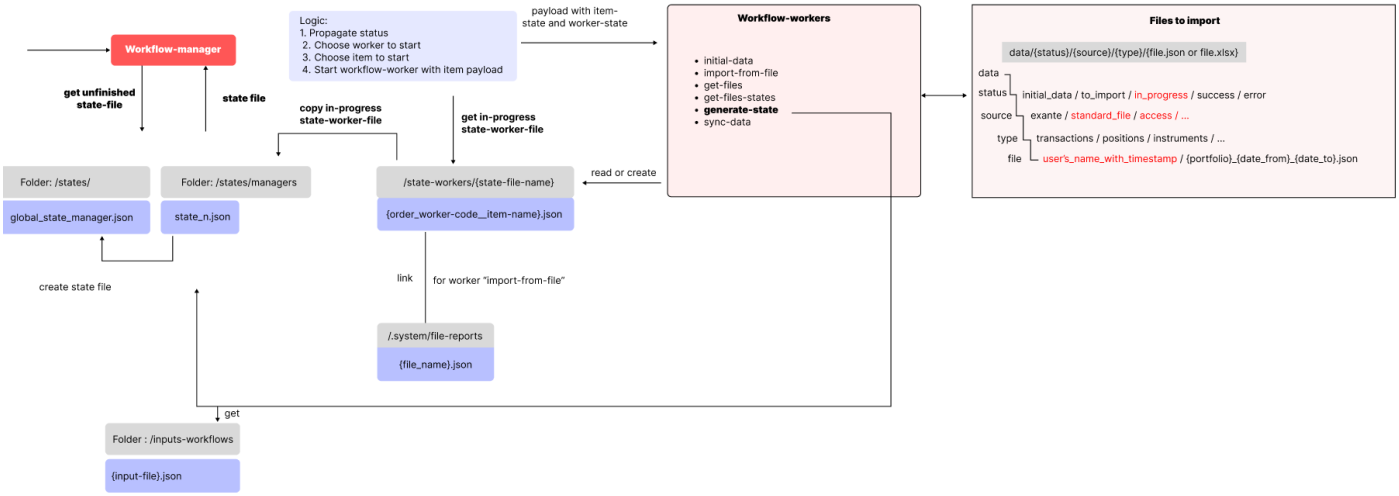
- `to-do` - launches the workflow worker for a specific item
- `in-progress` - checks the status of items:
- `error`, `skip`, `success` - skips to next worker

4. The manager uses the propagate status logic, where:

- `success` - will propagate to the entire worker if all items have this status
- `error` - will propagate the status from the item to the worker level

The `manager` does not wait for the `worker` to complete and finishes after starting the `worker`.

# Scheme of process




Detailed description provided in Finmars University

# Workflow Manager

## Overview

**User Code:** `com.finmars.standard-workflow:workflow-manager`
**Payload:** Optional

The Workflow Manager is designed to execute workers in a specific order with a defined payload. Its primary use case is when no payload is provided.

## Global State Manager Processing Logic

1. Attempt to read the `global_state_manager.json` file. If it doesn't exist, create it.
2. Read the list of files from `/states/managers/`.
3. Categorize by status and save in `global_state_manager.json`:
   - If the file doesn't exist, add it to "to-do"
   - If it's in "to-do", change the status to "in-progress"
   - If the file exists, update its status
4. Read the contents of the file in "in-progress" status


```
{
  "to-do": [],
  "in-progress": [
    "com.finmars.standard-workflow:workflow-manager-20240821095322.json"
  ],
  "done": [
    "com.finmars.standard-workflow:workflow-manager-20240724101635.json",
    "com.finmars.standard-workflow:workflow-manager-20240724104102.json"
  ],
  "paused": []
}
```

# Interaction with global_state_manager.json

The `global_state_manager.json` file serves as a central registry for all workflow states in the system:

1. **Initialization**: If `global_state_manager.json` doesn't exist, it's created by scanning the `/states/managers/` directory and categorizing all existing state manager files.
2. **Retrieval**: The existing `global_state_manager.json` is loaded and its data is updated.
3. **Updating**: The global state manager is updated by:
   - Identifying and adding new state manager files to the "in-progress" category
   - Checking and updating the status of each tracked state manager file
   - Moving state managers between categories based on their current status
4. **Saving**: After updates, the modified `global_state_manager.json` is saved back to storage.
5. **Workflow Processing**: The main `workflow_manager` function uses data from `global_state_manager.json` to determine which state managers to process when no specific payload is provided.

# State Managers and Workers/Items Interaction

Each state manager file represents an individual workflow instance, containing information about its workers and their respective items (tasks):

1. **Loading**: Individual state manager files are loaded based on paths stored in `global_state_manager.json` or provided in the payload.
2. **Status Propagation**: Each worker's status is updated based on the statuses of its items.
3. **Worker Processing**: For each worker in a state manager:
   - Skipped if status is 'success', 'skip', or 'ignore'
   - If 'in-progress', each item's status is checked:
     - 'in-progress' items: latest state is fetched and updated
     - 'to-do' items: a new workflow is initiated
   - If 'to-do', processing starts with the first 'to-do' item
4. **Item State Management**: Each item within a worker has its own state, including a `state_path` used to fetch and update item-specific states.
5. **Workflow Initiation**: New items are processed by calling the `start_workflow` function with the appropriate user code and payload.
6. **State Updating**: After any changes, the modified state manager is saved back to its file.

# Generate State Workflow

## Overview

**User Code:** `com.finmars.standard-workflow:generate-state`

This script is designed to generate state files for state managers. It processes input data to create a structured state file that includes information about workers and their respective states.

## Workflow Function

The main `workflow` function is a task that generates the state file based on the provided payload.

## Parameters:

- `payload`: A dictionary containing the necessary information for state generation.

## Process:

1. Validates the presence of a payload
2. Retrieves the input data from the specified path `input_file`
3. Generates a new state file using the `generate_state_file` function
4. Saves the generated state file
5. Updates the status of the operation in the state file

## Helper Functions

1. `get_first_transaction_date`: Retrieves the earliest transaction date from the API.
2. `get_next_date`: Calculates the next date based on the given periodicity.
3. `get_period_end_date`: Calculates the end date of a period based on the start date and periodicity.
4. `build_url`: Constructs the full API URL for a given endpoint.

5. `get_headers`: Generates headers with authorization token for HTTP requests.
6. `log_message`: Logs a message with a timestamp.
7. `save_file`: Saves data to a specified file path in JSON format.
8. `get_files`: Lists files in the specified directory.
9. `get_folders`: Lists directories in the specified path.
10. `get_data`: Reads data from a JSON file.

# Input Files

## Overview

The input file is a JSON file that contains the configuration for generating the state file. It specifies the overall workflow and individual worker configurations. Based on the provided examples, here's a detailed breakdown of the input file structure and its options.

## Input File Structure

```
{
  "user_code": "string",
  "configuration_code": "string",
  "name": "string",
  "schedule": null,
  "workers": [
    {
      "order": "integer",
      "configuration_code": "string",
      "name": "string",
      "user_code": "string",
      "state_type": "string",
      "download_options": {
        "date_from": "string | null",
        "date_to": "string | null",
        "type": "string | null",
        "periodicity": "string | null",
        "portfolios": ["string"] | null,
        "secret": "string | null"
      },
      "data_options": {
        "global_status": "string | null",
        "source": "string | null",
```

```
      "type": "string | null",
      "portfolios": null
    },
    "import_options": {
      "scheme": "string | null",
      "import_type": "string | null",
      "pricing_policy": "string | null"
    },
    "calculation_options": {
      "date_from": null,
      "date_to": null,
      "portfolios": null
    },
    "state_options": {
      "input_path": "string | null"
    }
   }
  ]
}
```

# Field Descriptions

1. **Root Level Fields**
   - `user_code`: Always "workflow-manager".
   - `configuration_code`: Always "com.finmars.standard-workflow".
   - `name`: A descriptive name for the workflow, e.g., "Exante Historical - Step 1 (Download): positions, transactions".
   - `schedule`: Deprecated. Always null.
2. **Worker Fields**
   - `order`: The execution order of the worker within the workflow (integer).
   - `configuration_code`: Usually "com.finmars.standard-workflow".
   - `name`: A descriptive name for the worker's task.
   - `user_code`: Identifies the specific task, e.g., "download-exante-positions", "preprocess-exante-transactions", "generate-state".
   - `state_type`: Can be "period", "files", or "fixed".
3. **Download Options**
   - `date_from`: Start date for data retrieval, e.g., "2024-01-01" or null.
   - `date_to`: End date for data retrieval, e.g., "2024-07-14" or null.
   - `type`: Can be "day", "period", or null.
   - `periodicity`: Can be "monthly" or null.
   - `portfolios`: An array of portfolio identifiers or null.

- `secret`: A string identifier for authentication, e.g., "itech-demo" or null.

4. **Data Options**
   - `global_status`: Can be "initial_data", "to_import", or null.
   - `source`: Usually "exante" or null.
   - `type`: Can be "positions", "transactions", "instruments", or null.
   - `portfolios`: Usually null in the provided examples.
   - `sync_to`: Used in "sync-files" tasks, e.g., "preprocessed".
5. **Import Options**
   - `scheme`: Import scheme identifier, e.g., "com.finmars.standard-import-from-file:accounts.account:account".
   - `import_type`: Can be "simple", "transaction", or null.
   - `pricing_policy`: Usually "com.finmars.standard-pricing:standard" or null.
6. **Calculation Options**
   - `date_from`: Usually null in the provided examples.
   - `date_to`: Usually null in the provided examples.
   - `portfolios`: Usually null in the provided examples.
7. **State Options**
   - `input_path`: Path to the next workflow file, e.g., "/input-workflows/exante/exante_historical_step_2.json" or null.

# Example Input File

Here's an example based on the "exante_historical_step_1.json" file:


```
{
  "user_code": "workflow-manager",
  "configuration_code": "com.finmars.standard-workflow",
  "name": "Exante Historical - Step 1 (Download): positions, transactions",
  "workers": [
    {
      "order": 1,
      "configuration_code": "com.finmars.standard-workflow",
      "name": "Download Positions",
      "user_code": "download-exante-positions",
      "state_type": "period",
      "download_options": {
        "date_from": "2024-01-01",
        "date_to": "2024-07-14",
        "type": "day",
        "periodicity": "monthly",
        "portfolios": ["Portfolio_007"],
```

```
      "secret": "secret-path-demo"
    },
    "data_options": {
      "global_status": null,
      "source": null,
      "type": null,
      "portfolios": null
    },
    "import_options": {
      "scheme": null,
      "import_type": null,
      "pricing_policy": null
    },
    "calculation_options": {
      "date_from": null,
      "date_to": null,
      "portfolios": null
    },
    "state_options": {
      "input_path": null
    }
  },
  // ... other workers ...
 ]
}
```

# Notes on Input Files

- The structure of the input file remains consistent across different steps of the workflow.
- Each step (represented by a separate JSON file) focuses on specific tasks such as downloading, preprocessing, or importing data.
- The "Generate State" worker is typically the last worker in each step, setting up the next step in the workflow.
- Options that are not relevant for a particular worker are usually set to `null`.
- The `state_type` determines how the worker processes data: "period" for date-based operations, "files" for file-based operations, and "fixed" for single-execution tasks.
- The `user_code` in the worker configuration specifies the exact workflow to be performed, such as downloading, preprocessing, or importing specific types of data.

# Interface (States & Input blocks)

## Overview

This Vue component, named `StateTable`, provides a user interface for managing state files and input files. It allows users to view, search, filter, and interact with these files in a tabular format.

## Key Features

1. Tabbed interface for States and Inputs
2. Searchable and filterable table
3. Expandable rows with detailed file content
4. Ability to start workflows
5. File status management
6. File deletion

## Component Structure

### Data Properties

- `activeTab`: Controls which tab is currently active (0 for States, 1 for Inputs)
- `pagination`: Controls table pagination
- `tabs`: Defines the available tabs and their configurations
- `apiBaseUrl`: Base URL for API requests
- `search`: Stores the current search term
- `workerStatusOptions`: Available status options for workers
- `selectedStatus`, `selectedGroup`: For filtering in respective tabs
- `rows`: Stores the main data displayed in the table

# Computed Properties

- `filteredData`: Returns the filtered and searched data for the table
- `filteredWorkerFields`: Filters specific fields from worker data for display

# Methods

1. `selectTab(index)`: Switches between States and Inputs tabs
2. `fetchApiData(endpoint)`: Fetches data from the API
3. `refreshFiles()`: Refreshes the file list
4. `startWorkflow(user_code, payload)`: Initiates a new workflow
5. `checkWorkflowStatus(workflowID, resolve, reject)`: Checks the status of a running workflow
6. `fetchFiles()`: Fetches and processes the list of files
7. `fetchFileContent(filePath, row)`: Retrieves detailed content for a specific file
8. `startWorkflowForRow(row, activeTab)`: Starts a workflow for a specific row
9. `toggleExpand(row)`: Expands/collapses a row to show/hide details
10. `saveStatus(row)`: Saves the current status of a file
11. `uploadFile(jsonData, fileName, path, currentFileIndex, totalFiles)`: Uploads a file to the server
12. `deleteFile(row)`: Deletes a file from the server

# Usage Guide

# Viewing Files

1. The component displays two tabs: "States" and "Inputs"
2. Each tab shows a table with relevant information about the files
3. Use the search bar to filter files by any field
4. Use the status/group dropdown (depending on the active tab) to filter files

# Interacting with Files

1. Click the '+' button on a row to expand and view detailed information
2. In the expanded view:
   - For States: You can view worker details and their individual states
   - For Inputs: You can view the structure of the input file

# File Actions

1. **Start Workflow**:
   - Click "Start Workflow" in the expanded view to initiate a workflow for that file
   - For States, it uses the `workflow-manager` user code
   - For Inputs, it uses the `generate-state` user code
2. **Save State** (States tab only):
   - Click "Save State" to update the file on the server with any changes
3. **Delete State** (States tab only):
   - Click "Delete State" to remove the file from the server

# Managing Worker and Item States

1. In the expanded view, each worker and item has a status dropdown
2. Click on the status to open the dropdown and select a new status

# Refreshing Data

- Click the refresh button (circular arrow icon) to fetch the latest data from the server

# API Integration

The component interacts with several API endpoints:

1. File listing: `${apiBaseUrl}/api/v1/explorer/view/?path=components/states_files.json` or `${apiBaseUrl}/api/v1/explorer/view/?path=components/inputs_files.json`
2. File content: `${apiBaseUrl}/api/v1/explorer/view/?path=${filePath}`
3. Workflow start: `${apiBaseUrl}/workflow/api/workflow/run-workflow/`
4. Workflow status check: `${apiBaseUrl}/workflow/api/workflow/${workflowID}/`
5. File upload: `${apiBaseUrl}/api/v1/explorer/upload/`
6. File deletion: `${apiBaseUrl}/api/v1/explorer/delete/?path=${filePath}&is_dir=false`

# Notes for Developers

- The component uses custom icons (`IconRefresh`, `IconCopy`, `IconDownload`, `IconTabActive`) which should be properly imported
- Authorization is handled by the `authorization()` function from `@/utils/customFetch`
- The component relies on the Quasar framework for UI components (e.g., `q-table`, `q-input`, `q-select`)

# Interface (Files Block)

## Overview

The `FileTable` component is a Vue component that provides a user interface for managing and uploading files. It allows users to view existing files in a table format, filter and search through files, and upload new files to specific locations.

## Key Features

1. Table view of existing files with sorting and filtering capabilities
2. File upload functionality with source and data type selection
3. Progress bar for file uploads
4. Refresh functionality to update the file list

## Component Structure

### Data Properties

- `pagination`: Controls table pagination
- `search`: Stores the current search term
- `selectedType`, `selectedSource`, `selectedStatus`: For filtering the file table
- `typeOptions`, `sourceOptions`, `statusOptions`: Options for the filter dropdowns
- `rows`: Stores the main data displayed in the table
- `columns`: Defines the structure of the table
- `uploadView`: Toggles between file table view and upload view
- `selectedDataTypeUpload`, `selectedSourceUpload`: For selecting upload options
- `dataTypeUploadOptions`, `sourceUploadOptions`: Options for upload dropdowns
- `filesToUpload`: Stores files selected for upload
- `uploadInProgress`, `uploadProgress`: Manages upload state and progress

# Computed Properties

- `filteredData`: Returns the filtered and searched data for the table

# Methods

1. `changeToUploadFiles()`: Toggles between file table view and upload view
2. `refreshFiles()`: Refreshes the file list by triggering a workflow
3. `startWorkflow(user_code, payload)`: Initiates a new workflow
4. `checkWorkflowStatus(workflowID, resolve, reject)`: Checks the status of a running workflow
5. `fetchFiles()`: Fetches and processes the list of files
6. `handleFileUpload(event)`: Handles file selection for upload
7. `uploadFiles()`: Manages the file upload process
8. `getPath()`: Generates the upload path based on selected options
9. `uploadFile(file, path, currentFileIndex, totalFiles)`: Uploads a single file

# Usage Guide

# Viewing Files

1. The component displays a table of files with columns for File, Type, Source, and Status
2. Use the search bar to filter files by any field
3. Use the Data Type, Source, and Status dropdowns to further filter the table

# Uploading Files

1. Click the '+' button to switch to the upload view
2. Select the Source and Data Type for the upload
3. Choose files to upload using the file input
4. Click "Upload Files" to start the upload process
5. A progress bar will display the upload progress

# Refreshing Data

- Click the refresh button (circular arrow icon) to fetch the latest data from the server

# API Integration

The component interacts with several API endpoints:

1. File listing: `${apiBaseUrl}/api/v1/explorer/view/?path=components/data_files.json`
2. Workflow start: `${apiBaseUrl}/workflow/api/workflow/run-workflow/`
3. Workflow status check: `${apiBaseUrl}/workflow/api/workflow/${workflowID}/`
4. File upload: `${apiBaseUrl}/api/v1/explorer/upload/`

# Notes for Developers

- The component uses custom icons (`IconPlus`, `IconRefresh`) which should be properly imported
- Authorization is handled by the `authorization()` function from `@/utils/customFetch`
- The component relies on the Quasar framework for UI components (e.g., `q-table`, `q-input`, `q-select`)
- File uploads include a timestamp in the filename to ensure uniqueness

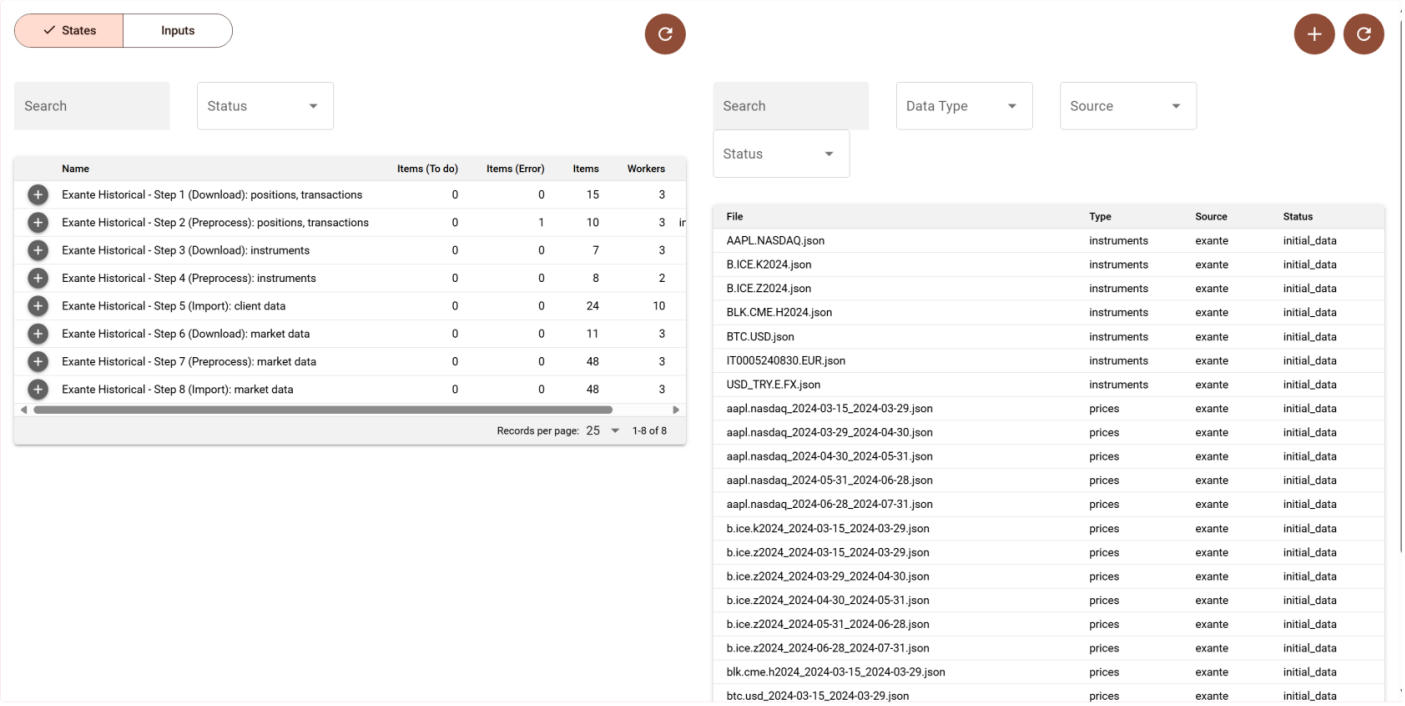
# Potential Enhancements

1. Implement pagination on the server-side for better performance with large datasets
2. Add more robust error handling and user notifications for upload failures
3. Implement file chunking for large file uploads (commented out in the current version)
4. Add a preview feature for uploaded files
5. Implement drag-and-drop functionality for file uploads

# Interface

## Overview

This `standard-workflow-interface` module provides a comprehensive interface for managing states and files within a workflow system. The interface is divided into two main components: StateTable and FileTable, which are displayed side by side in the main application view.




# Main Application Structure

The main application is defined in `App.vue` and consists of two primary components:

1. StateTable
2. FileTable

These components are laid out side by side using a flex container for optimal screen utilization.

# Key Components

## 1. StateTable

The StateTable component manages and displays workflow states.

Interface (States & Input blocks)

Key features:

- Displays a list of workflow states
- Allows filtering and searching of states
- Provides detailed view of each state
- Enables starting workflows and managing state status

## 2. FileTable

The FileTable component manages and displays files within the system.

Interface (Files Block)

Key features:

- Displays a list of files with their types, sources, and statuses
- Allows filtering and searching of files
- Provides file upload functionality
- Enables refreshing the file list

# User Interface

The interface is divided into two main sections:

1. **Left Side (StateTable)**:
   - Tabs for switching between "States" and "Inputs"
   - Search and filter options
   - Table displaying state information
   - Expandable rows for detailed state information
2. **Right Side (FileTable)**:
   - Table displaying file information

- Search and filter options
- File upload functionality

# Common Features

- Both components feature a refresh button (circular arrow icon) to update their respective data.
- The interface uses a consistent design language, with similar table layouts and filter options.

# API Integration

Both components interact with backend APIs for data retrieval and manipulation. The API calls are managed through custom fetch functions defined in `customFetch.js`.

# Development Notes

- The project uses Vue 3 with the Composition API.
- Quasar framework is used for UI components.
- Custom icons are used and should be properly imported in each component.
- API base URL is dynamically determined based on the current URL.

# Interface Authorization

## Overview

This project uses token-based authentication with Keycloak integration. The authentication process is managed through custom functions defined in `customFetch.js`.

## Key Functions

`getCookie(name)`

Retrieves a cookie value by name.

`authorization()`

Prepares the authorization header for API requests using the access token stored in cookies.

`refreshTokenAndRetry(request)`

Handles token refresh when a request fails due to an expired token:

1. Updates the token using Keycloak.
2. Updates cookies with new token information.
3. Retries the original request with the new token.

`customFetch(request)`

A wrapper around the fetch API that handles authentication:

1. Attempts the initial request.
2. If a 401 (Unauthorized) response is received, it attempts to refresh the token and retry the request.

`getApiBaseUrl()`

Dynamically determines the API base URL based on the current page URL.

`uploadFileWithProgress(endpoint, formData, onUploadProgress)`

Handles file uploads with progress tracking and token refresh capabilities.

# Authentication Flow

1. Initial requests include the access token from cookies.
2. If a request fails due to an expired token (401 response): a. The token is refreshed using Keycloak. b. Cookies are updated with the new token information. c. The original request is retried with the new token.
3. If token refresh fails, the user is redirected to the Keycloak login page.

# Security Notes

- Access tokens are stored in cookies and are included in the `Authorization` header of each request.
- Token refresh is handled automatically when a request fails due to an expired token.
- HTTPS should be used in production to secure token transmission.

# Development Considerations

- The `getDevToken()` function is available for development purposes but should not be used in production.
- Ensure Keycloak is properly configured in your development and production environments.
- Be cautious when modifying the authentication flow to maintain security.

For any changes to the authentication process, consult with the security team to ensure best practices are followed.

# Configuration instructions

Configuration instructions

# Pipeline Setup for Data Import

1. **Document the Data Import Pipeline**:
   Outline the steps required to set up the data import pipeline.
2. **Identify Missing Workers**:
   Compile a list of workers that are not included in the current implementation.
3. **Determine Appropriate Module**:
   Identify the module where each worker should be located.
   - If the necessary module does not exist, create a new one.
4. **Define State File Format**:
   Specify the state file format that each worker should operate with.
5. **Evaluate Existing Options**:
   Determine if the current options are sufficient to complete the scenario.
   - If not, additional development of the `generate-state` function is required.
6. **Update Vault (if applicable)**:
   - If Vault is used, add the necessary data to Vault.
7. **Create Input Files for the Pipeline**:
   Generate input files for the pipeline, considering the missing workers.
8. **Run State Generator for the First Input File**:
   Execute the state generator on the first input file.
9. **Reach Missing Worker via Workflow Manager**:
   - Proceed to the missing worker by running the workflow manager.
10. **Create Worker Stub for Payload Display**:
    - Develop a stub for the worker to display the payload.
11. **Write Missing Worker Script**:
    Develop the script for the missing worker.
12. **Run Workflow Manager Post-Debugging**:
    - After debugging, restart the workflow manager.
13. **Create Production Input Files**:
    - Create input files for production (daily and historical data).
14. **Set Up Cron Job for Daily Tasks**:
    - Schedule the daily tasks using a cron job (pending update to version 1.9.0).