FRUITION

The future begins today...

USER GUIDE

Fruition Configuration Manager

Fruition Consultancies
info@fruitionconsultancies.com

Contents

Description

Simplify the way you manage configurations, translations, and storefront labels — without code changes or redeployments.

With this Configuration Manager, store administrators gain full control over language-specific elements for every store and locale. Instead of hardcoding storefront labels, everything can be managed dynamically within the app — making your store agile, scalable, and localization-ready.

Key Capabilities:

✅ Save configuration and translation values per store/locale

✅ Edit or delete saved configurations instantly

✅ Powerful search by key, value, locale, or store

✅ API access to fetch all or selected configurations by key, locale, or store

✅ Centralized list of all configurations for easy management

✅ No redeployment needed — update storefront labels and translations on the fly

Why it matters:

1. Eliminate the need for hardcoded labels

2. Launch faster in new markets with easy translation management

3. Empower business users to make changes without relying on developers

4. Reduce deployment cycles and accelerate time-to-market

Prerequisites

Admin UI SDK – Installation & Configuration

Installation:

- **PaaS**
  - Please follow the instructions Install or update Adobe Commerce Admin UI SDK
  - Admin UI SDK version 3.0 or higher
- **SaaS**
  - Included already

IMS Module for Authentication

For PaaS: Please follow the instructions Adobe Identity Management Service (IMS) for Adobe Commerce

For SaaS: SaaS instances already include IMS configuration

Project Setup

Installing from App Page

This step walks you through installing the app for **Adobe Commerce** via the Adobe Exchange marketplace.

1. Go to Adobe Exchange, select "Experience Cloud" and search for **Configuration Manager**.
2. Click button **Buy** or **Install** and follow the prompts. The application will be automatically deployed to your Adobe App Builder environment.

Please refer to the page https://developer.adobe.com/developer-distribution/experience-cloud/docs/guides/discoverAndManage/app-builder-discover

Setting up the required configurations

- Adobe Commerce Base URL

  Note: When configuring the COMMERCE_BASE_URL environment variable, the format differs between PaaS and SaaS:

  For PaaS (On-Premise/Cloud):

  - Must include your base site URL + /rest/ suffix
  - Example: https://[environment-name].us-4.magentosite.cloud/rest/

  For SaaS:

  - Must be the REST API endpoint provided by Adobe Commerce
  - Example: https://na1-sandbox.api.commerce.adobe.com/[tenant-id]/

  Make sure to use your actual environment name or tenant ID in the URL **and the URL should end in a slash(/)**. The examples above use placeholder values.

Please provide below configurations

- OAUTH_CLIENT_ID
- OAUTH_CLIENT_SECRETS
- OAUTH_TECHNICAL_ACCOUNT_ID

- OAUTH_TECHNICAL_ACCOUNT_EMAIL
- OAUTH_IMS_ORG_ID
- COMMERCE_CONSUMER_KEY (PaaS, OAuth, if IMS Auth isn't being used)
- COMMERCE_CONSUMER_SECRET (PaaS, OAuth, if IMS Auth isn't being used)
- COMMERCE_ACCESS_TOKEN (PaaS, OAuth, if IMS Auth isn't being used)
- COMMERCE_ACCESS_TOKEN_SECRET (PaaS, OAuth, if IMS Auth isn't being used)

Note:

For PaaS:

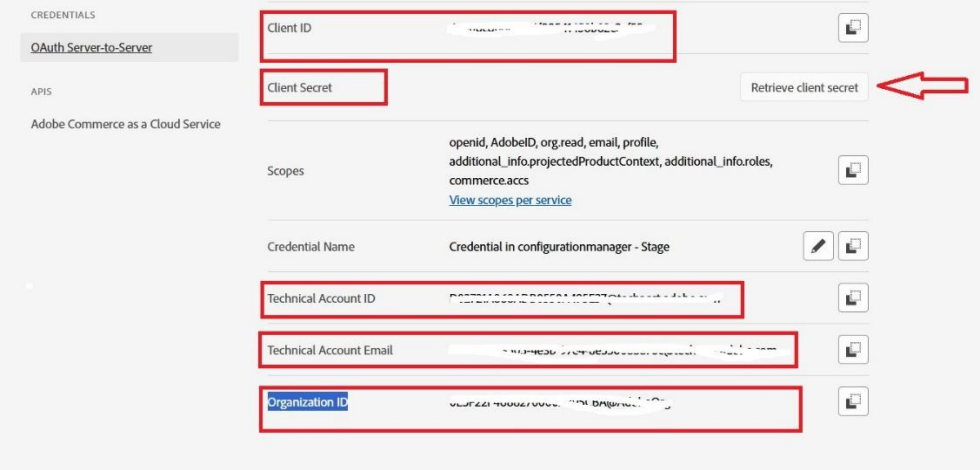
Case1: With IMS Auth - Given that the IMS module will be enabled for use with Admin UI SDK, the OAuth credentials could also be used to authenticate with PaaS. This process requires a Commerce instance with Adobe Identity Management Service (IMS) for Adobe Commerce configured.

Case2: If IMS Auth isn't being used, but OAuth is used – The values (COMMERCE_CONSUMER_KEY, COMMERCE_CONSUMER_SECRET, COMMERCE_ACCESS_TOKEN, COMMERCE_ACCESS_TOKEN_SECRET) can be retrieved from a Commerce integration, please refer to https://experienceleague.adobe.com/en/docs/commerce-admin/systems/integrations

For SaaS: SaaS instances already include IMS configuration

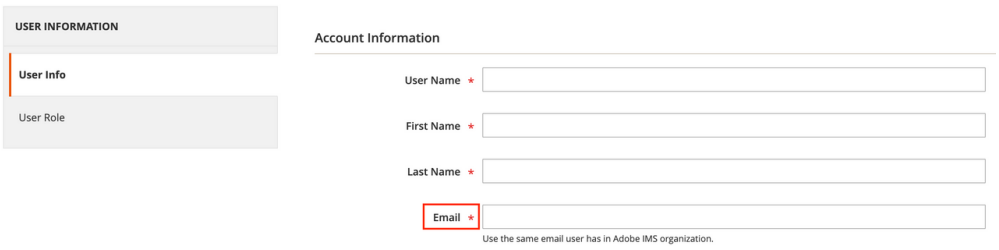
Use the following steps to create OAuth credentials for App Builder authentication:

- Go to Adobe Developer Console
  - https://developer.adobe.com/console
  - Log in with your Adobe ID (same org where your project is created).
- Access your IMS credentials through the Adobe Developer Console. Select any project and workspace within the IMS organization. Then click OAuth Server-to-Server in the side-navigation menu.
- NOTE: These credentials are automatically populated in Configure OAuth Server-to-Server Credential.
  - OAUTH_CLIENT_ID=<client id> (Found in Developer Console → your Project → Credentials → **Client ID**.)
  - OAUTH_CLIENT_SECRETS=<client secrets> (Found in Developer Console → your Project → Credentials → **Client Secrets**.)
  - OAUTH_TECHNICAL_ACCOUNT_ID=<technical account id> (Found in Developer Console → your Project → Credentials → **Technical Account ID**.)
  - OAUTH_TECHNICAL_ACCOUNT_EMAIL=<technical account email> (Found in Developer Console → your Project → Credentials → **Technical Account Email**.)
  - OAUTH_SCOPES=<scopes>
  - OAUTH_IMS_ORG_ID=<ims org> (Found in Developer Console → your Project → Credentials → **Organization ID**.)

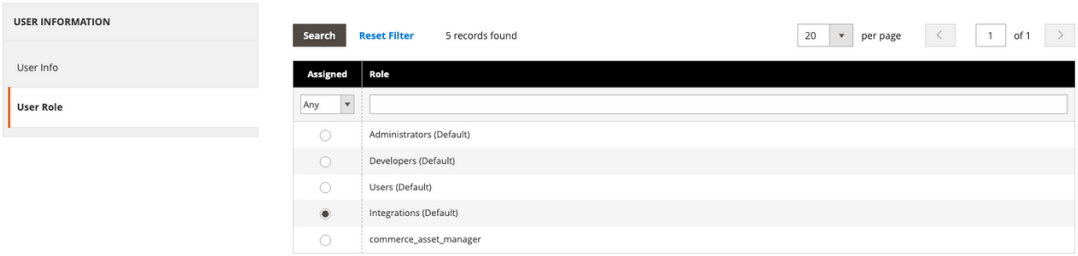


- Provide the technical account with access to the Commerce instance:
  - **SaaS** only The technical account is automatically created and associated with the Commerce instance once the first request is made using the OAuth credentials.
  - **PaaS** only Add a technical account with server-to-server credentials to the Commerce Admin with the appropriate permissions using the Admin User Creation Guide.

    When associating the user with the account, find your Technical Account email as a part of generated IMS credentials with following pattern: ***<technical-account>@techacct.adobe.com*** and use that value in the Email field during user creation:



On the User Role tab, select the role that provides all necessary permissions for API integrations.

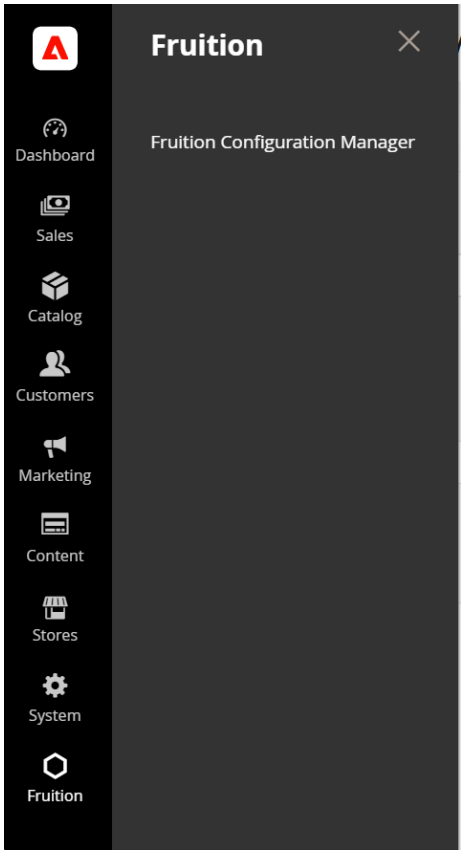


Locate Configurations in Admin Panel

Configuration:

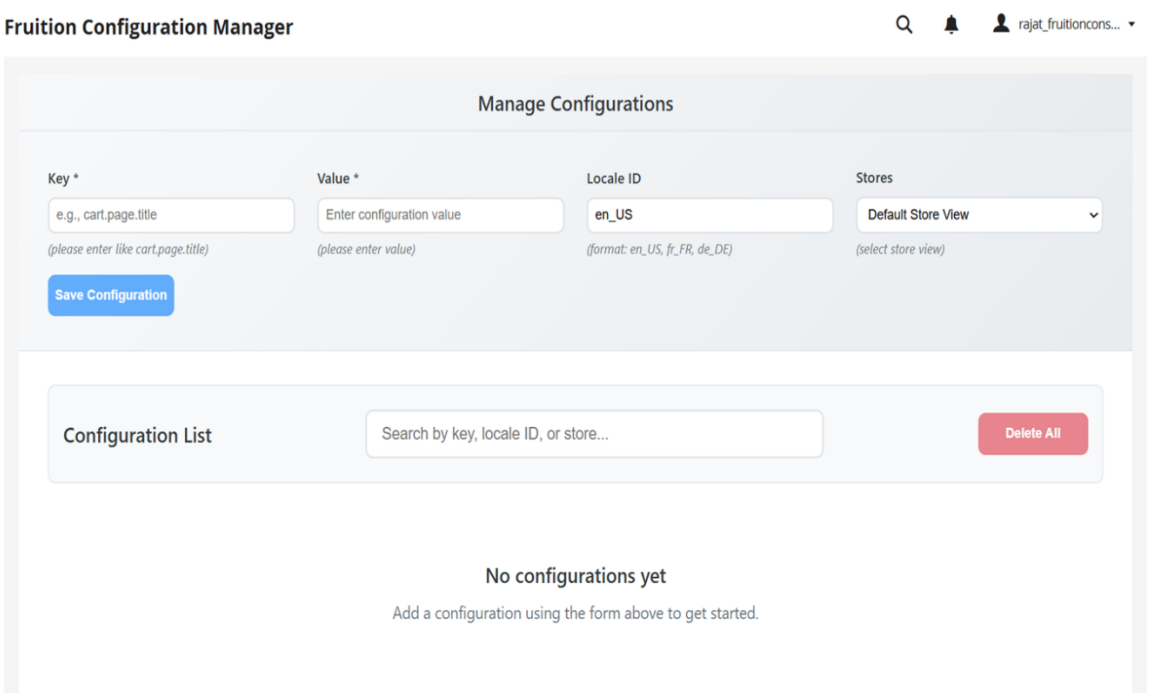
Please follow the instructions Admin configuration and testing

Once the App is installed and configured via Admin SDK UI, you will be able to see the Menu with name “Fruition”, when you click on it, a sub menu will appear “Fruition Configuration Manager”.



Landing on Configuration Manager

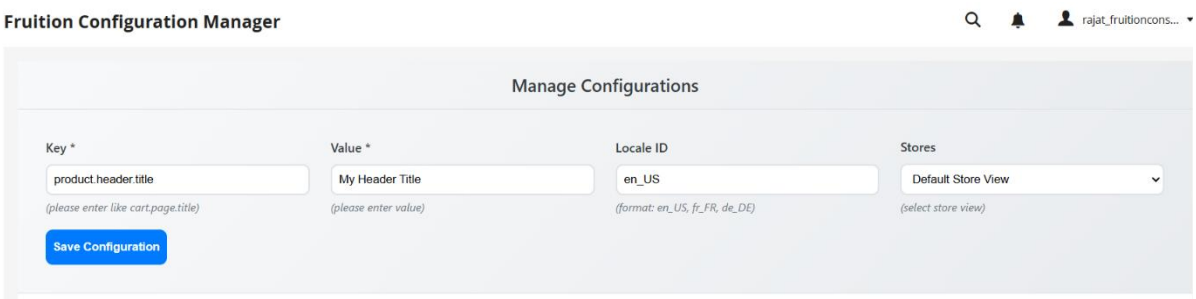
When you click on **Fruition Configuration Manager**, the **Configuration Manager screen** opens. From here, you can **add, edit, delete, or view** existing configurations.



Add a Configuration

To add a new configuration:

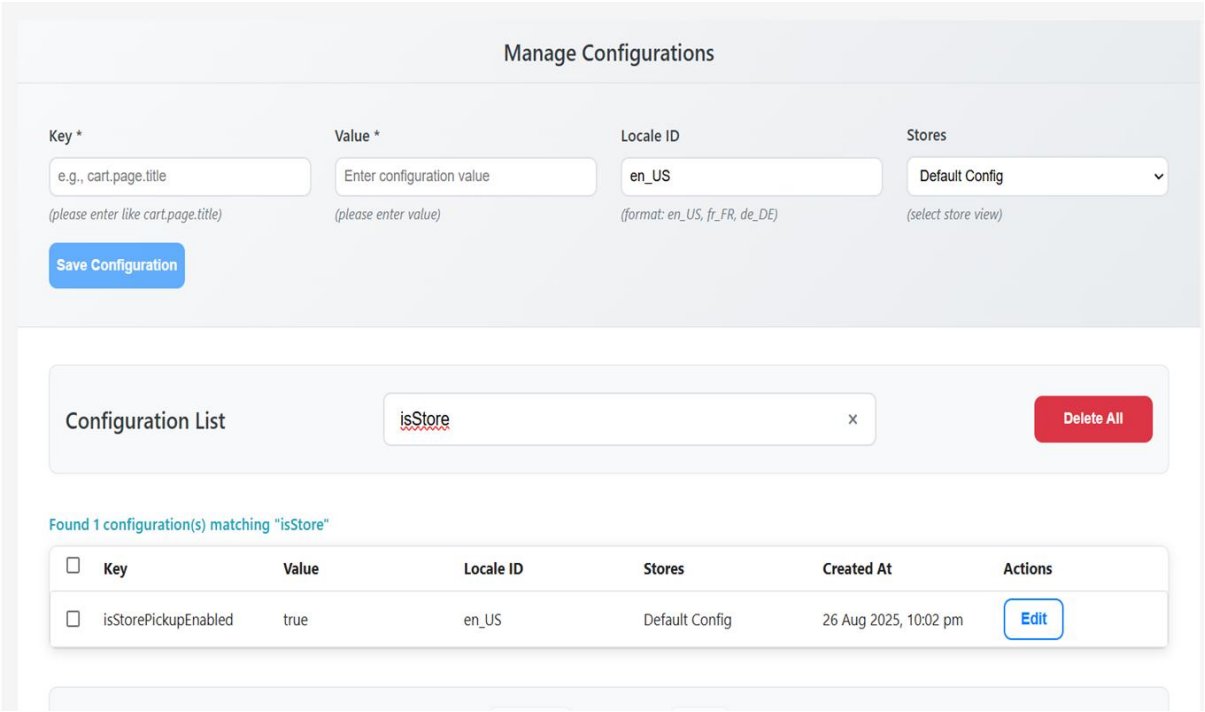
- Enter the **Key** and **Value**.
- Specify the **Locale ID**.
- Select the appropriate **Store** where this configuration should apply.
- Save the configuration.



Search Configuration

You can search existing configurations using filters such as **Key**, **Value**, **Locale ID**, or **Store ID**.

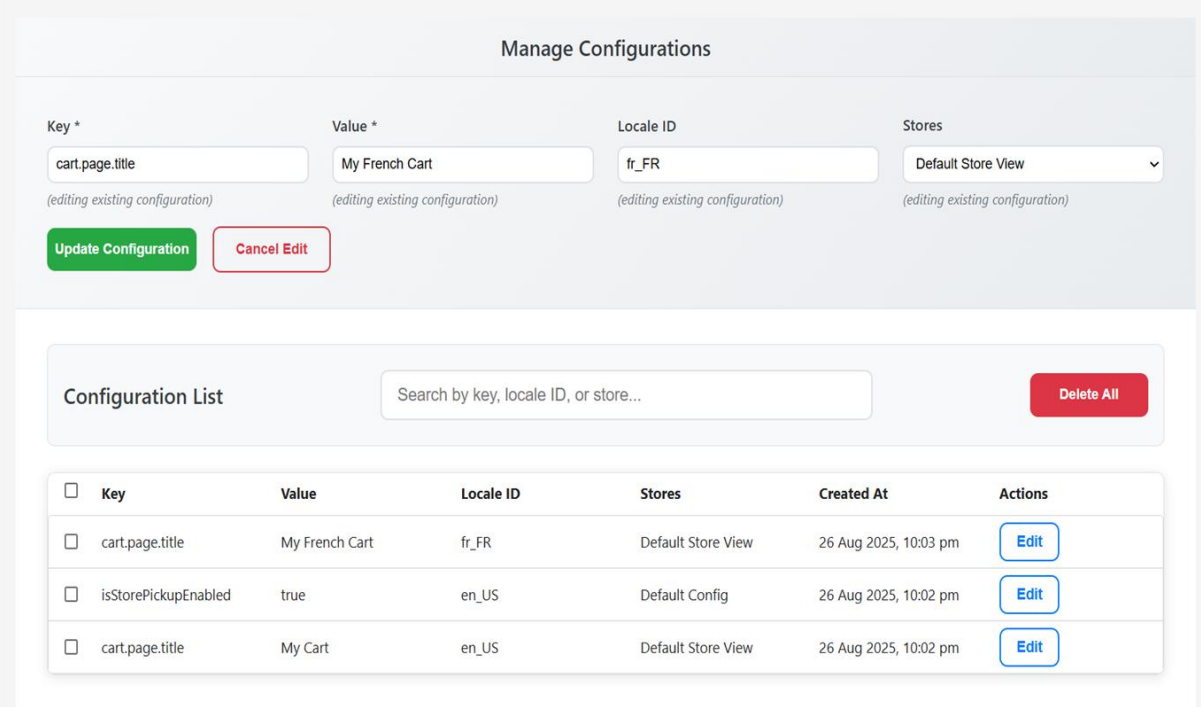
- If no filters are applied, the system will display **all available configurations**.



Edit Configuration

To edit an existing configuration:

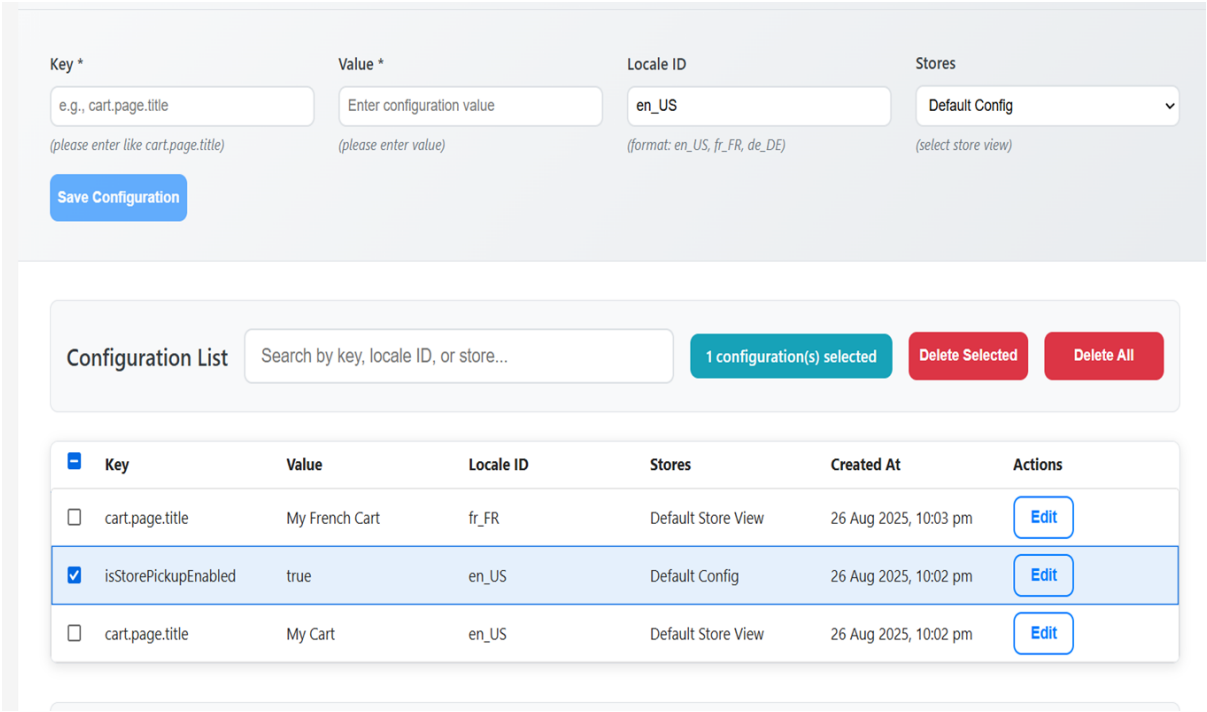
1. Select the configuration you want to modify.
2. Click the **Edit** link.
3. The current values will be displayed with editable fields.
4. Make the required changes.
5. Click **Update Configuration** to save your changes.



Delete Configuration

To delete a configuration:

1. Select the checkbox next to the configuration you want to remove.
2. You can select multiple configurations if needed.
3. Click the **Delete Selected** button.
4. Confirm the deletion when prompted.

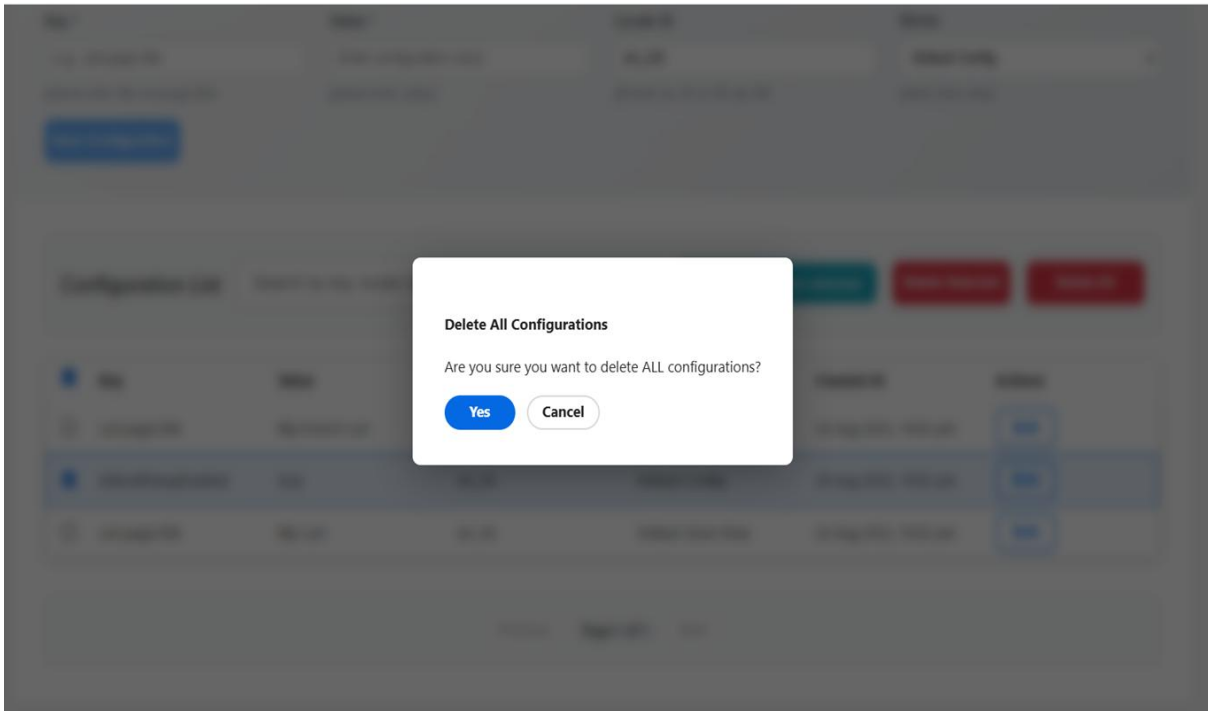


Delete All Configurations

To remove all existing configurations in one step:

1. Click the **Delete All** button.
2. Confirm the deletion when prompted.

⚠️ **Note:** This action is irreversible and will permanently delete all configurations from the system.



API Endpoints

Base URL

<URL>/api/v1/web/configuration-manager

Get All Configurations

This endpoint retrieves configuration data from the Configuration Manager. It is a simple HTTP GET request that returns the current configurations stored in the system.

HEADER

x-gw-ims-org-id: <OrganizationID>

Authorization: Bearer <Auth Token>

Request

Endpoint: /getConfigurations

Method: GET

Response

- **Status Code**: 200 OK
- **Content-Type**: application/json

Response Body

The response contains the following structure:

- **JSON**

  { "data": [{ "id": 0, "key": "", "value": "", "localeid": "", "storeid": "", "created_at": "" }], "total": 0, "filters": null}

- **data**: An array of configuration objects. Each object contains:
  - id: A unique identifier for the configuration.
  - key: The key associated with the configuration.
  - value: The value of the configuration.
  - localeid: The locale identifier for the configuration.
  - storeid: The store identifier associated with the configuration.
  - created_at: The timestamp when the configuration was created.
- **total**: The total number of configurations returned. This will indicate how many configurations are available in the response.
- **filters**: This field may contain any applied filters, but in this case, it is null.

Notes

- This endpoint does not require any request parameters.
- The response may return an empty data array if no configurations are available.
- Ensure to handle the response appropriately based on the total value.

Get Configurations by Filter

This endpoint retrieves configuration settings based on specified filters. It allows users to query configurations by providing a filter object that includes the desired parameters.

HEADER

x-gw-ims-org-id: <OrganizationID>

Authorization: Bearer <Auth Token>

Request

- **Method**: POST
- **Endpoint**: /getConfigurations
- **Request Body** (JSON):
  - filter: An object containing the following keys:

- key (string): The specific configuration key to filter by (e.g., "cart.page.title").
- localeid (string): The locale identifier for the configuration (optional).
- storeid (string): The store identifier for the configuration (optional).

Example Request Body

- **JSON**

```
{ "filter": {  "key": "cart.page.title",  "localeid": "",  "storeid": "" }}
```

Response

- **Status Code**: 200 OK
- **Content-Type**: application/json
- **Response Body** (JSON):
  - data: An array of configuration objects that match the filter criteria, with the following properties:
    - id (integer): The unique identifier of the configuration.
    - key (string): The configuration key.
    - value (string): The value associated with the configuration.
    - localeid (string): The locale identifier for the configuration.
    - storeid (string): The store identifier for the configuration.
    - created_at (string): The timestamp of when the configuration was created.
  - total (integer): The total number of configurations returned.
  - filters: An object reflecting the filters used in the request.

Example Response Body

- **JSON**

```
{ "data": [  {  "id": 0,  "key": "",  "value": "",  "localeid": "",  "storeid": "",
"created_at": ""  } ], "total": 0, "filters": {  "key": "",  "localeid": "",  "storeid": "" }}
```

- **data**: An array of configuration objects. Each object contains:
  - id: A unique identifier for the configuration.
  - key: The key associated with the configuration.
  - value: The value of the configuration.
  - localeid: The locale identifier for the configuration.
  - storeid: The store identifier associated with the configuration.

- created_at: The timestamp when the configuration was created.

- **total**: The total number of configurations returned. This will indicate how many configurations are available in the response.
- **filters**: This field may contain any applied filters, but in this case, it is null.

Notes

- Ensure that the key parameter is provided to filter the configurations effectively.
- The localeid and storeid parameters are optional and can be used to further refine the search.
- The response will include an array of configurations that match the provided filter criteria, along with metadata about the total number of configurations found.

Instructions for Integrating an EDS / Headless Storefront with app

Below are steps to integrate this app with an EDS Storefront or Headless App:

1. **Setting up EDS storefront –** Please follow the steps mentioned in the link https://experienceleague.adobe.com/developer/commerce/storefront/get-started/create-storefront/
2. **API Exposure from the App** – The Configuration Manager exposes configuration data (labels, translations, etc.) via APIs. – This is already there in App Builder App. Refer to section API Endpoints in this document.
3. **Consumption in the EDS / Headless Storefront** – An EDS / Headless storefront consumes these APIs and could render the updated labels dynamically.
    a. Consume the API
    b. Call API in the block like commerce-cart.js
        a. const configuration = await callToConfigurationManager();
    c. Get the desired configuration / label
        a. const cartheading = configuration.data.find(item=> item.key === "cart.page.title");
    d. Render the value in desired place

```
Heading: (ctx) => {
    const headingEl = document.createElement('h2');
    headingEl.className = 'fruition__cart__heading';
```

```
        headingEl.textContent = cartheading?.value || 'Cart';
        ctx.appendChild(headingEl);
      }
```

4. **Real-time Updates** – Storefront labels and translations can be updated on the fly from the admin, without needing a redeploy.
5. **Extension Beyond Placeholders –** While EDS already supports managing out-of-the-box labels through placeholder files, our app extends this by enabling merchants to define and manage any new labels introduced in slots, containers, or new drop-ins, keeping them consistent and centrally controlled.
6. **Unified Workflow** – Since the EDS-based demo covers both use cases, the same approach ensures compatibility across SaaS and PaaS storefronts.

Link to Video - https://fruitionconsultancies.com/wp-content/uploads/2025/09/Configuration-Manager-SaaS-with-EDS-Demo.mp4

Note:

- The steps under "Consumption in the EDS / Headless Storefront" are just one example of how to use the app's configurations.
- The App is compatible with Luma storefront also, simply integrate the Luma storefront with API exposed by App and then you can use the configuration / label accordingly on the page.

Configuring a Mesh

The getConfigurations endpoint can be added as a source to a mesh, the mesh can securely store the auth credentials needed for requests to the getConfigurations endpoint. Please refer to the link for more information Command Reference

Sample mesh configuration snippet for the endpoint:

```
{
 "meshConfig": {
  "sources": [
   {
    "name": "getConfiguration",
    "handler": {
     "JsonSchema": {
      "baseUrl": <app_builder_endpoint_prefix>,
```

```
        "operations": [
         {
           "type": "Query",
           "field": "getConfigurationByFilters",
           "path": "/getConfigurations",
           "method": "POST",
           "requestSchema": "./requestSchema/get-configuration.json",
           "responseSchema": "./responseSchema/get-configuration.json"
         },
         {
           "type": "Query",
           "field": "getConfiguration",
           "path": "/getConfigurations",
           "method": "GET",
           "responseSchema": "./responseSchema/get-configuration.json"
         }
        ],
        "operationHeaders": {
          "Authorization": "Bearer {context.secrets.authorization}",
          "x-gw-ims-org-id": "{context.secrets.x-gw-ims-org-id}",
          "Content-Type": "application/json"
        }
      }
     }
    }
  ]
 }
}
```

<app_builder_endpoint_prefix> - Replace with your URL

requestSchema/get-configuration.json

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "title": "Get Configuration Request",
 "type": "object",
 "properties": {
  "filter": {
   "type": "object",
   "properties": {
    "key": {
     "type": "string"
    },
    "localeid": {
     "type": "string"
    },
    "storeid": {
     "type": "string"
    }
   },
   "additionalProperties": false
  }
 },
 "additionalProperties": false
}
```

responseSchema/get-configuration.json

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "title": "Get Configuration Response",
```

```
"type": "object",
"properties": {
 "data": {
  "type": "array",
  "items": {
   "type": "object",
   "properties": {
    "id": {
     "type": "integer"
    },
    "key": {
     "type": "string"
    },
    "value": {
     "type": "string"
    },
    "localeid": {
     "type": "string"
    },
    "storeid": {
     "type": "string"
    },
    "created_at": {
     "type": "string",
     "format": "date-time"
    },
    "updated_at": {
     "type": "string",
     "format": "date-time"
    }
   },
```

```
      "required": ["id", "key", "value", "localeid", "storeid", "created_at"],
      "additionalProperties": false
    }
   },
   "total": {
    "type": "integer"
   },
   "filters": {
    "type": ["object", "null"]
   }
  },
  "required": ["data", "total"],
  "additionalProperties": false
}
```


Examples of Creating Secret File

Create a YAML file, such as secrets.yaml, to define your secrets. The file name must end with the yaml or yml file extension. Each line in the files defines a different secret.

Example:

authorization: ${authorization_token}

x-gw-ims-org-id: ${x-gw-ims-org-id}

Create or update your mesh secrets

When you create or update a mesh that you want to include secrets in, add the --secrets flag followed by the path to your secrets file. If you do not provide the secrets file when updating a mesh that has secrets, the secrets` values are overwritten by their literal references.

Create:

aio api-mesh create mesh.json --secrets secrets.yaml

Update:

aio api-mesh update mesh.json --secrets secrets.yaml

For more information about mesh secret, please visit Secrets management

Support

In case of any questions, please contact support@fruitionconsultancies.com