MULESOFT PORTFOLIO

Emiliano Benavides Díaz

Tabla de contenido

Introducción

Problemática

Desarrollar la interface para una aplicación móvil que muestre la información básica del universo de Star Wars, como películas existentes, personajes principales o naves espaciales. En esta ocasión, se espera tener la información de todos los personajes de las películas en un formato CSV para que la aplicación muestre estos datos en una tabla tipo Excel, y el usuario final pueda comparar las características de los personajes.

Se sabe que el origen de los datos es una API (SWAPI) que devuelve la información en un formato JSON. Se debe diseñar (con RAML o Swagger) una especificación de API y desarrollar un proyecto Mule que permita recuperar la información de TODOS los personajes de Star Wars que la SWAPI provee y entregarlos en un formato CSV, ejemplo:

```
name,height,mass,hair_color,skin_color,eye_color,birth_year,gender
Luke Skywalker,172,77,blond,fair,blue,19BBY,male
C-3PO,167,75,n/a,gold,yellow,112BBY,n/a
```

Como segundo requisito, se debe probar la funcionalidad de que a través de un Query Parameter se pueda filtrar el resultado recibido en el CSV basándose en el género del personaje, ejemplo:

```
Gender=female

name,height,mass,hair_color,skin_color,eye_color,birth_year,gender
Leia Organa,150,49,brown,light,brown,19BBY,female
Beru Whitesun lars,165,75,brown,light,blue,47BBY,female
```

Acercamiento

La manera de abordar el problema era sencilla. Necesitaba crear una especificación en API Designer para poder filtrar las búsquedas, es decir, que únicamente aceptara las request con los parámetros que yo definía. De igual manera, creé esta API de manera que llevara manejo de errores, ejemplos y los parámetros aceptados. Después, desplegué la API en Anypoint Studio y desarrollé la implementación. Esta implementación contenía la búsqueda de la información, la transformación al formato que quería (JSON y CSV) y la limpieza de la información para ajustarla a los requerimientos que necesitaba: name, height, mass, hair_color, skin_color, eye_color, birth_year, gender, eliminando información no relevante como homeworld, species, etc.

Lo siguiente fue desplegar la aplicación en Anypoint Exchange para que se pudiera usar públicamente. Posteriormente, agregué la API a API Manager y la desplegué en CloudHub.

Nota:
La información no está necesariamente en orden de creación, sino que está escrita de manera que sea más sencillo de leer.

Desarrollo de la solución

Especificación de API

SWAPI Request

Descripción

El propósito de la API es obtener información de SWAPI (Star Wars API) tanto en formato JSON como en formato CSV, así como permitir query parameters y enumerar qué parámetros acepta el request. También se definen diferentes mensajes tanto de ejemplo como de manejo de errores, y se especifican los tipos y detalles de estos mensajes.

Versiones

- 1.0.0

Métodos

- GET

Parámetros

- Query Parameters
- Body Parameters

Ejemplos de Solicitudes y Respuestas

- Ejemplos de payloads de salida

Códigos de Estado HTTP

- 200 • 400 • 404 • 405 • 415 • 500

Design Center

El API se dividió en tres formatos:

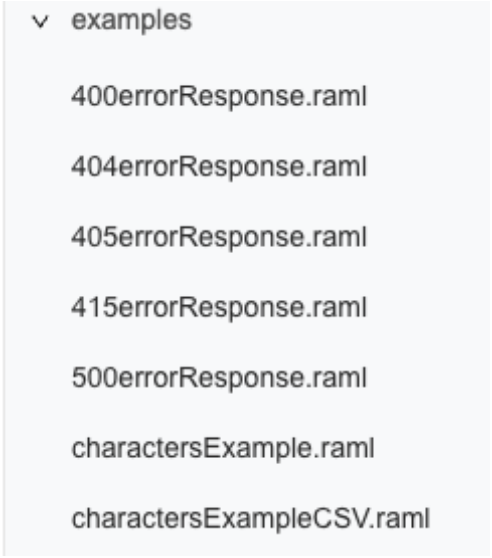
Root File: En este archivo, especifico todo lo necesario para el funcionamiento de la API. Este archivo es el swapi-request.raml.
Carpeta de Ejemplos: Contiene archivos RAML para diferentes ejemplos, como 400errorResponse.raml y charactersExampleCSV.raml, para facilitar el uso de la API desde el Mocking Service.
Carpeta de Data Types: Contiene archivos de datatypes para poder especificar los requerimientos de las respuestas de los archivos de ejemplos.

En el Root File se especificó el método disponible para la aplicación, en este caso un GET, junto con la opción para aceptar un query parameter en el que irá el filtro por género. Analizando la información de SWAPI, podemos ver los diferentes géneros que hay y los enumeramos para que, si el género no está dentro del query parameter, no acepte la request. Los géneros enumerados son: male, female, n/a, hermaphrodite y none.

```
#%RAML 1.0
title: Star Wars Characters API
mediaType: application/json

/characters:
  displayName: Characters
  description: Resource to manage characters
  get:
    description: Retrieve a list of Star Wars characters.
    queryParameters:
      gender:
        description: Filter characters by gender.
        type: string
        required: false
        enum:
          - male
          - female
          - n/a
          - hermaphrodite
          - none
    responses:
      200:
        body:
          application/csv:
            example: !include examples/charactersExampleCSV.raml
          application/json:
            type: !include datatypes/characterType.raml
            example: !include examples/charactersExample.raml

      400:
        body:
          application/json:
            type: !include datatypes/errorType.raml
            example: !include examples/400errorResponse.raml

      404:
        body:
          application/json:
            type: !include datatypes/errorType.raml
            example: !include examples/404errorResponse.raml
```

Después, creamos los parámetros de las respuestas que podría contestar la API, siendo 200, 400, 404, 415 y 500. Para cada respuesta se separaron las respuestas disponibles que son application/json y application/csv. Para cada respuesta se creó un archivo de ejemplos que está ligado a un archivo de datatypes.

Dentro de la carpeta de ejemplos, se encuentran los archivos de cada respuesta, cada uno conteniendo el ejemplo de cómo se vería la respuesta para cada response:



Para la respuesta 200, se configuró un RAML tanto para la respuesta en CSV como para la respuesta en JSON, conteniendo los arreglos conformados por tres objetos para que fuera más claro a la hora de usarse.

```
#%RAML 1.0 NamedExample

name,height(cm),weight(kg),hairColor,skinColor,eyesColor,birthYear,gender
Luke Skywalker,172,77,blond,fair,blue,19BBY,male
C-3PO,167,75,n/a,gold,yellow,112BBY,n/a
R2-D2,96,32,n/a,white\, blue,red,33BBY,n/a
```

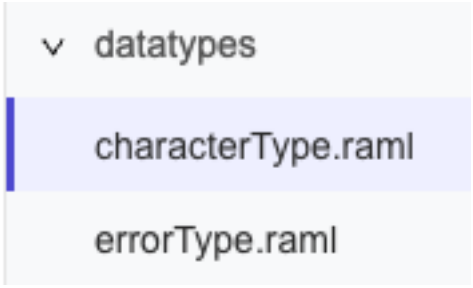
```
#%RAML 1.0 NamedExample
[
  {
  "name": "Luke Skywalker",
  "height(cm)": 172,
  "weight(kg)": 77,
  "hairColor": "blond",
  "skinColor": "fair",
  "eyesColor": "blue",
  "birthYear": "19BBY",
  "gender": "male"
},
{
  "name": "C-3PO",
  "height(cm)": 172,
  "weight(kg)": 77,
  "hairColor": "blond",
  "skinColor": "fair",
  "eyesColor": "blue",
  "birthYear": "19BBY",
  "gender": "none"
},
{
  "name": "Darth Vader",
  "height(cm)": 172,
  "weight(kg)": 77,
  "hairColor": "blond",
  "skinColor": "fair",
  "eyesColor": "blue",
  "birthYear": "19BBY",
  "gender": "male"
}
]
```

Estos objetos estaban definidos en base a los requerimientos de información sobre los personajes. Es decir, se definió que cada objeto dentro del arreglo debe contener: name, height (cm), weight (kg), hairColor, skinColor, eyesColor, birthYear y gender.

Mientras que, para las respuestas de errores, se definió el código del error, el mensaje, el detalle y un timestamp para facilitar la búsqueda de un error.


```
{
    "code": "400",
    "message": "Bad request",
    "detail": "Missing required property [name]",
    "timestamp": "2024-07-16T01:32:29.540039Z"
}
```


Y, por último, para la sección de data types, creamos dos carpetas: una que contenía las especificaciones de los objetos de characters y otra para las especificaciones de los objetos de error.



Para el characterType.raml, se especificó qué propiedades debía contener cada parte del objeto dentro del arreglo de los personajes. En este caso, para cada atributo como el nombre, la altura, etc., se definió tanto el tipo que iba a ser (si iba a ser string por ser texto o si iba a ser integer por ser un valor numérico), como si era un campo obligatorio o no. Para esto, también se añadió una breve descripción para que su entendimiento fuera más sencillo.

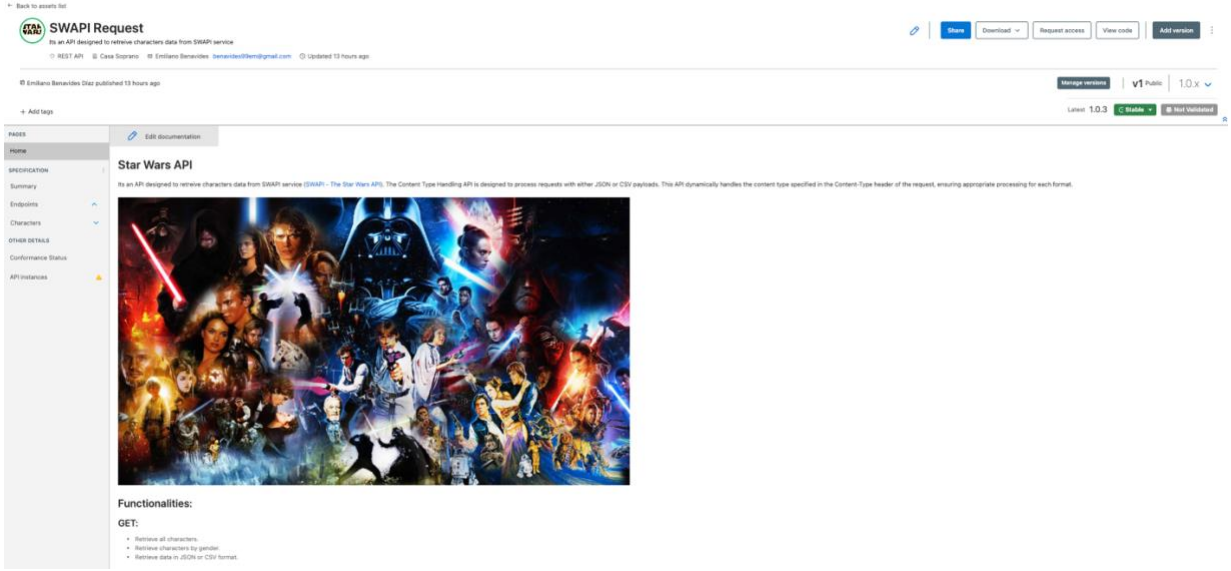
```
#%RAML 1.0 DataType

type: array
items:
  type: object
  properties:
    name:
      type: string
      required: true
      description: "Character name."
    height(cm):
      type: integer
      required: true
      description: "Character height in cm."
    weight(kg):
      type: integer
      required: true
      description: "Character weight in kg."
    hairColor:
      type: string
      required: true
      description: "Character hair color."
    skinColor:
      type: string
      required: true
      description: "Character skin color."
    eyesColor:
      type: string
      required: true
      description: "Character eyes color."
    birthYear:
      type: string
      required: true
      description: "Character birth year."
    gender:
      type: string
      required: true
      description: "Character gender."
```

Para la sección de errores, utilizamos errorType.raml, en donde se definió que el mensaje de error sería un objeto y los requerimientos del mensaje, que en este caso serían: code, message, detail y timestamp; especificando igualmente que cada uno era string, que era necesario y añadiendo una breve descripción.

El estructurar la API de esta manera no solo garantizó el funcionamiento correcto de la API, sino que también nos facilitó la organización y manejo de diferentes errores, como un error que no estaba permitiendo que yo agregara un request a la API en formato CSV ya que no estaba definido dentro del body de la respuesta 200.

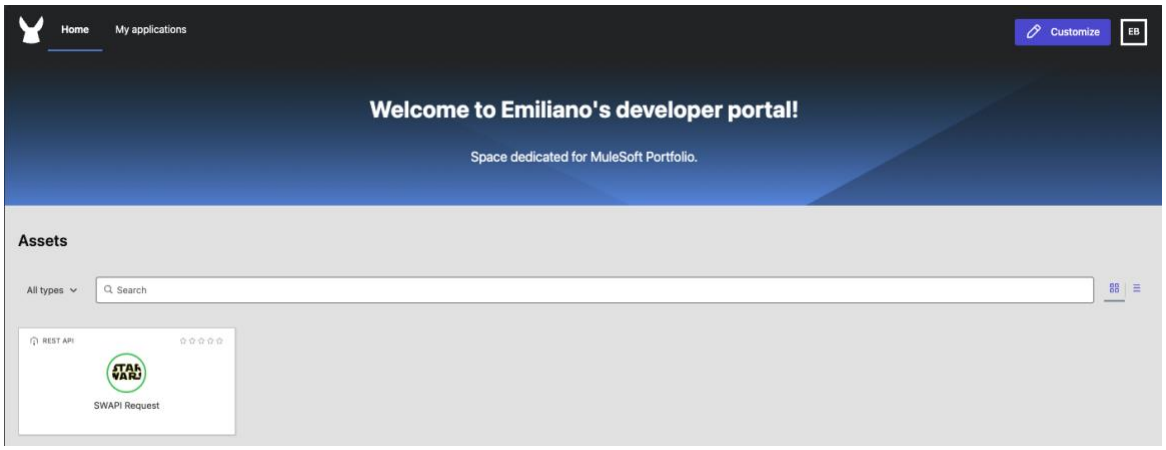
Publicación en Exchange

Para la publicación de la API se utilizó Anypoint Exchange. Una vez publicada esta versión, se agregó una breve descripción que incluye tanto una descripción de lo que hace la API, como las funcionalidades, la información de contacto y un logo para facilitar la búsqueda de la misma.



Portal Público

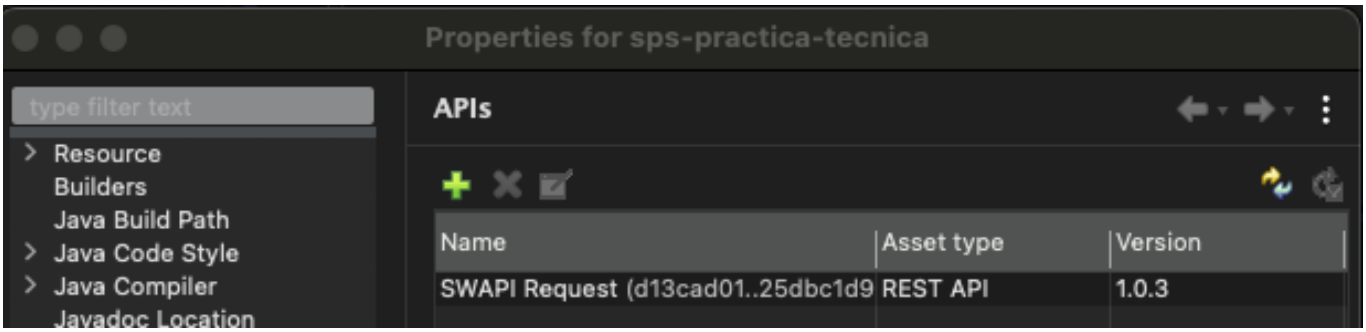
Para el portal público en donde se encuentra la API se customizo con diferente información para resaltar la página, agregando tanto el logo, como la paleta de colores y una breve descripción del portal que contiene la API de “SWAPI Request”.



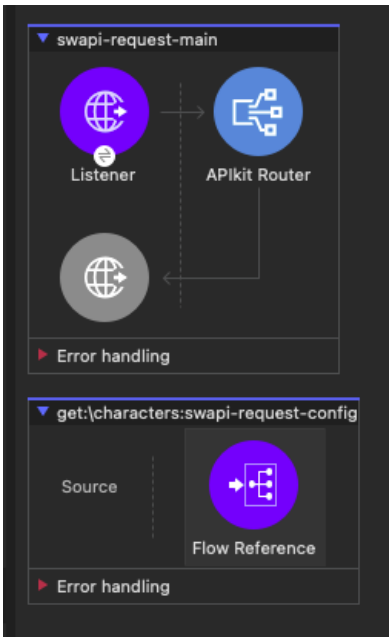
Implementación de API

Interfaz

Una vez con la API creada y desplegada en Anypoint Exchange, se creó un nuevo proyecto en Anypoint Studio llamado mule-portfolio-emiliano-benavides. Dentro de este proyecto, en la sección de manage dependencies, se implementó esta API a través de Anypoint Exchange.



Esa implementación sirvió para generar el flujo swapi-request-main, que conformaría parte del flujo de la interfaz. Este se encarga de trackear todas las solicitudes a sus diferentes métodos con un solo listener, en este caso, el único método es un GET.

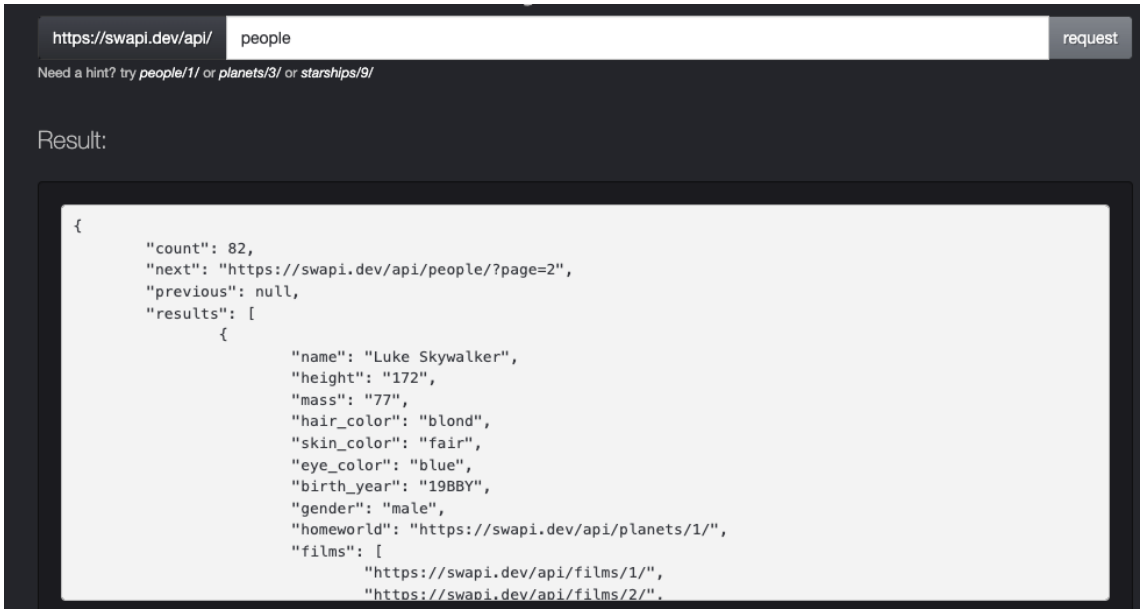


Esto facilita entender la lógica de la aplicación, donde lo único que se encuentra dentro de este Mule Configuration File es el flujo principal que trackea los requests a su método, se encarga de separar los flujos por métodos y facilita el manejo de errores.

Implementación

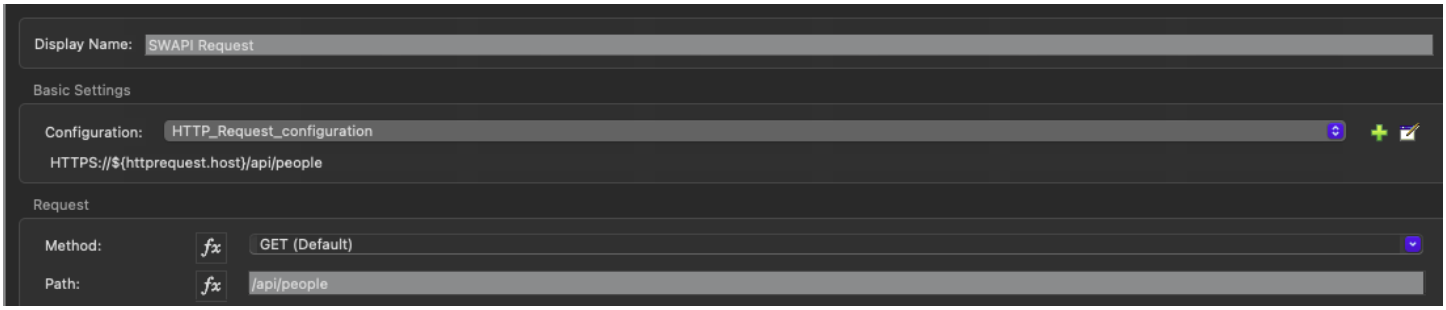
Para la implementación se creó un nuevo “Mule Configuration File” para separar la interfaz de la lógica de la aplicación, creando un nuevo mainFlow que se encargará de esto.

Lo primero que se debía hacer era consultar SWAPI (SWAPI - The Star Wars API) para poder entender cómo es que se daba la información y así poder seguir la solución. Esta página daba la información dividida en Starships, Homeworld, Vehicles, Species, Films y la que interesaba que era “People”, y dentro de ese People se arrojaba un objeto que contenía Count (con el número de personajes), next (con la URL a la siguiente página), previous (con la URL a la página anterior) y results que contiene un arreglo con diferentes objetos, cada uno con las especificaciones de un personaje.



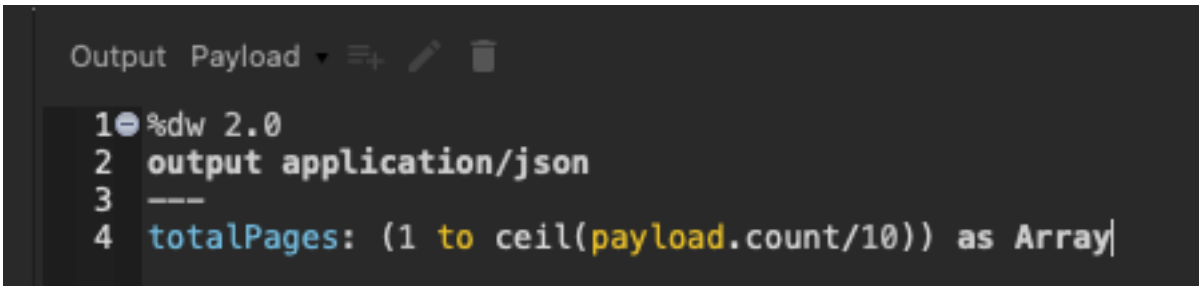
Con esta información se obtuvo la URL a la que se tenía que hacer el request, que es “https://swapi.dev/api/people”, y de qué parte del payload se tenía que sacar la información de los personajes.

Ya teniendo conocimiento sobre cómo se debía abordar el problema. Entonces, en el mainFlow dentro de la implementación, se comenzó agregando un Request. Este request se configuró en base a lo que tenía que extraer, que en este caso es la sección de people.



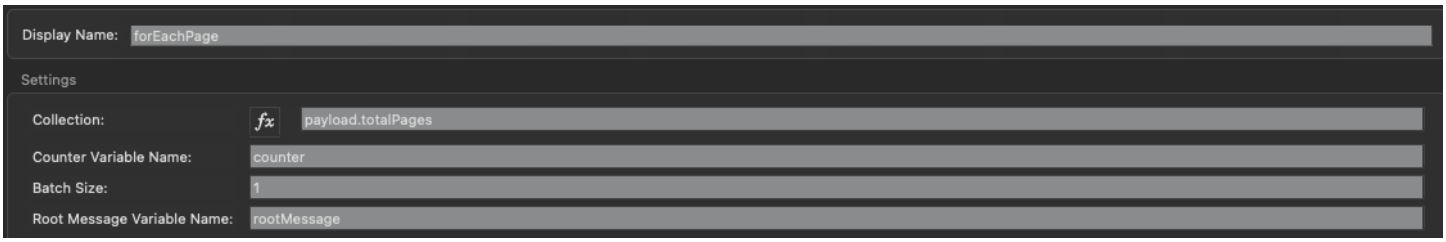
Una vez que se obtuvo la información, se notó que no se daba la lista de personajes completos, sino que se daba por páginas. Al principio se intentó extraer la URL de la sección de next ya que contenía la URL a la siguiente página, pero se estaba volviendo muy complicado, entonces se abordó de diferente manera.

Con count se podía saber cuántos personajes se iban a extraer y, viendo los datos y pasando todas las páginas dentro de SWAPI, se vio que el patrón de cada página y es que cada página contiene 10 personajes. Teniendo esta información, el siguiente paso fue calcular cuántas páginas hay y se implementó un transform message para calcular las mismas.

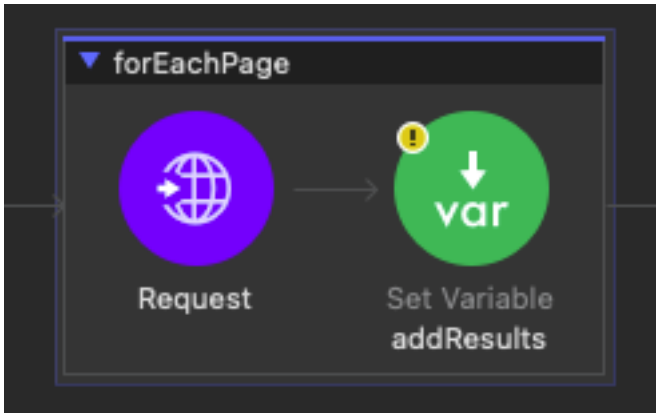


La fórmula lo que hace es extraer del payload la información de count que contiene el total de los personajes. Este número se divide entre 10 ya que cada página tenía 10 personajes. Para afrontar el problema de que no son números enteros, se utilizó la fórmula para redondear este número hacia arriba y así obtener el total de páginas necesarias. Todo esto se extrajo en un arreglo para poder utilizar un for each más adelante.

Para el for each se agregó la configuración para que utilizara el arreglo que anteriormente se sacó, que es payload.totalPages, ya que este contiene un arreglo con el número total de páginas, ocasionando que el for each se repita las veces necesarias.

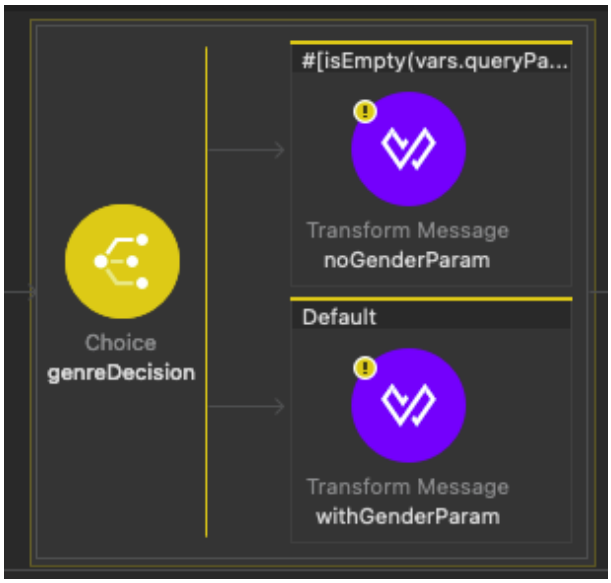


Dentro del for each hay dos módulos: el primero es un request que funciona exactamente igual que el request anterior con la diferencia de que no se extrae la información total de los personajes, sino que se mantiene toda la información del payload. Luego, se creó una variable en donde se almacenará la información relevante, agregando un Set Variable.

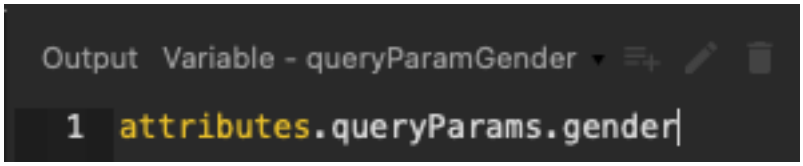


Por medio de DataWeave, se implementó una fórmula para que añadiera los personajes. Esta fórmula se conforma de dos partes: vars.characters default [] que se encarga de crear la variable y ponerla como arreglo, y la segunda parte que es ++ payload.results, que extrae la información de results del payload, donde se encuentra la información de los personajes, y la suma como arreglo.

Gracias a estas dos funciones, ya se puede extraer la información completa. Lo siguiente sería añadir la función con la cual se pueda hacer la filtración por género con un choice.



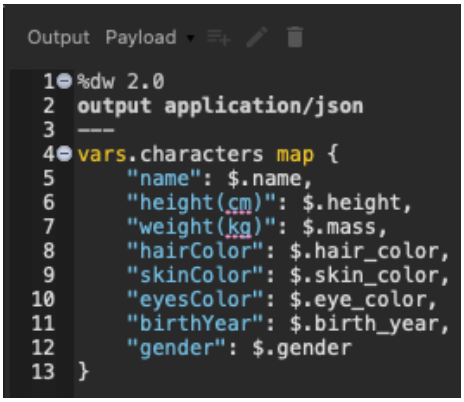
Para el choice, se necesitó una nueva variable que sería el query parameter, por lo que se creó al inicio del flujo otro transform message con nombre setVariables, que se encarga de extraer de los atributos el query parameter gender y crear una variable llamada QueryParamGender.



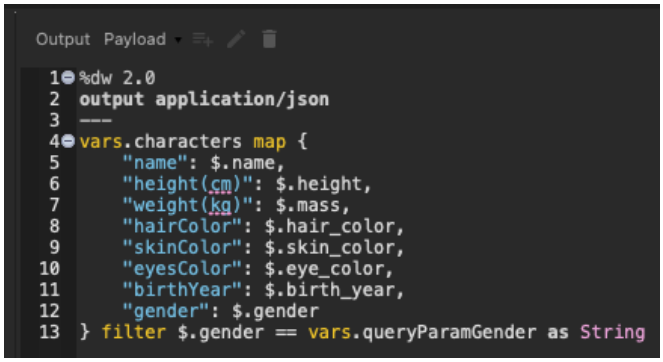
Ya con esta nueva variable, se pudo filtrar en el choice. La función escrita dentro del choice es #[isEmpty(vars.queryParamGender)], esto para diferenciar si se otorgó esa variable o no, ya que si no tiene variable, aun así se necesita extraer toda la información.

Ahora, el problema que se debe abordar es que ya se diferenció entre si tiene queryParam o no, pero ahora se necesita limpiar la información acorde a las necesidades. Cada opción del choice tiene un transform message que se encarga de esta función.

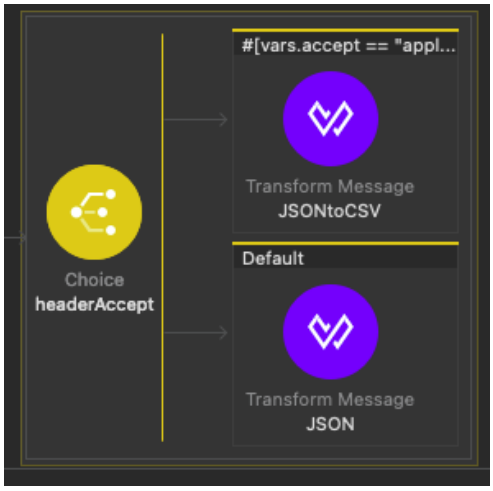
Para el caso en el que el request no tenga queryParam, se dirige a un transform message que se encarga de extraer la información necesaria, es decir, filtrar la información que ya se tiene guardada en la variable para extraer únicamente los datos necesarios del personaje utilizando un map, gracias a que anteriormente se había puesto la condición de que la variable que los contiene es un arreglo.



Para el caso en donde sí tenga queryParam, se utilizó otro transform message igual que el anterior, con la diferencia de que en este transform message se utilizó la fórmula de filter $.gender == vars.queryParamGender as String, esto para que de la información que ya se obtuvo se limpie, comparando el género con la variable que previamente se agregó de género.



Ya con la información completa, tanto si tiene queryParam o no, el siguiente paso sería otorgar la opción de que se extraiga tanto en formato CSV o en formato JSON, por lo que se agregó otro choice.

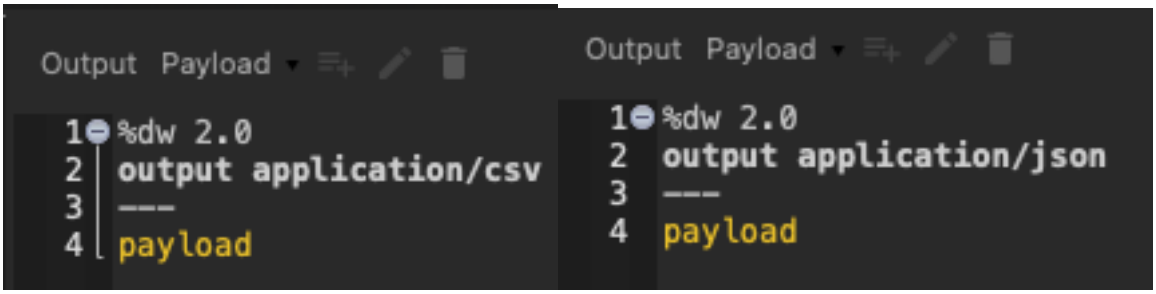


Para este choice se necesitó otra variable que ayudara a diferenciar cuál es el formato requerido. Entonces, al primer transform message que se encargaba de extraer la variable de género se le agregó una nueva variable que se puso en los headers. Esta variable se llama accept, por lo que en los headers dentro de la búsqueda se puede seleccionar accept – application/json o application/csv.

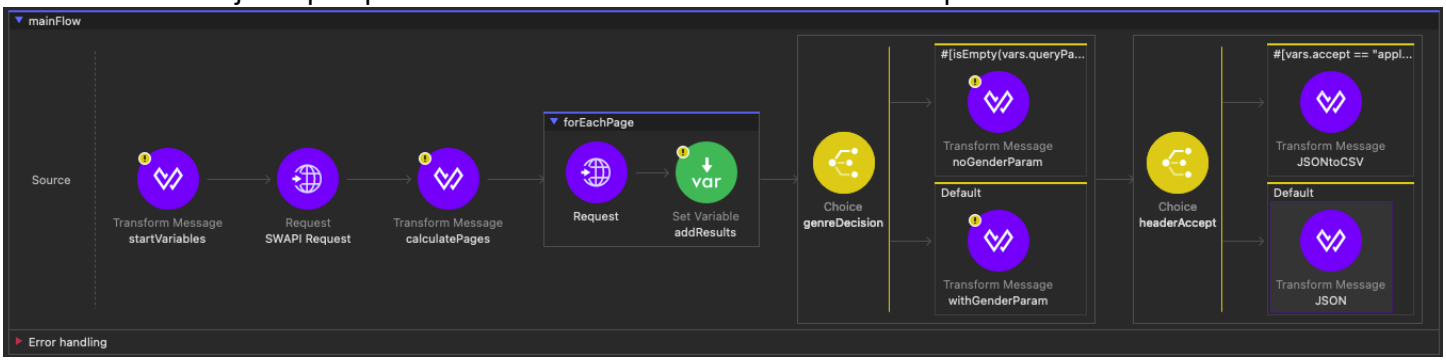
Ya con la nueva variable en el choice, se puede distinguir con la fórmula #[vars.accept == “application/csv”].

Es decir, si el header contiene application/csv se va a esa opción y, si no, lo extrae en formato JSON.

Ya con la información, cada transform message convierte la información al formato deseado.

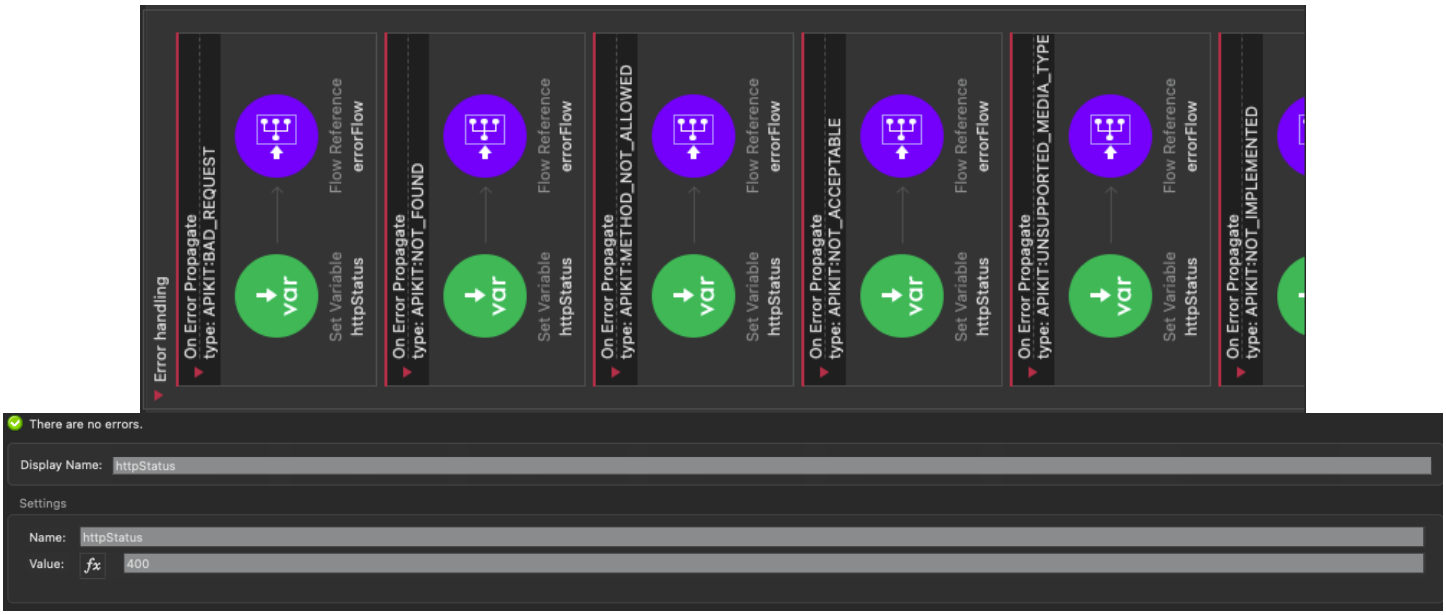


Garantizando que la búsqueda pueda filtrar tanto por género como por formato. Mientras que, si no se especifica nada, aun así regresa un resultado tanto en JSON como la lista de todos los personajes sin importar el género. y obteniendo un flujo limpio que es referenciado desde la interface a la implementación con un flow reference.

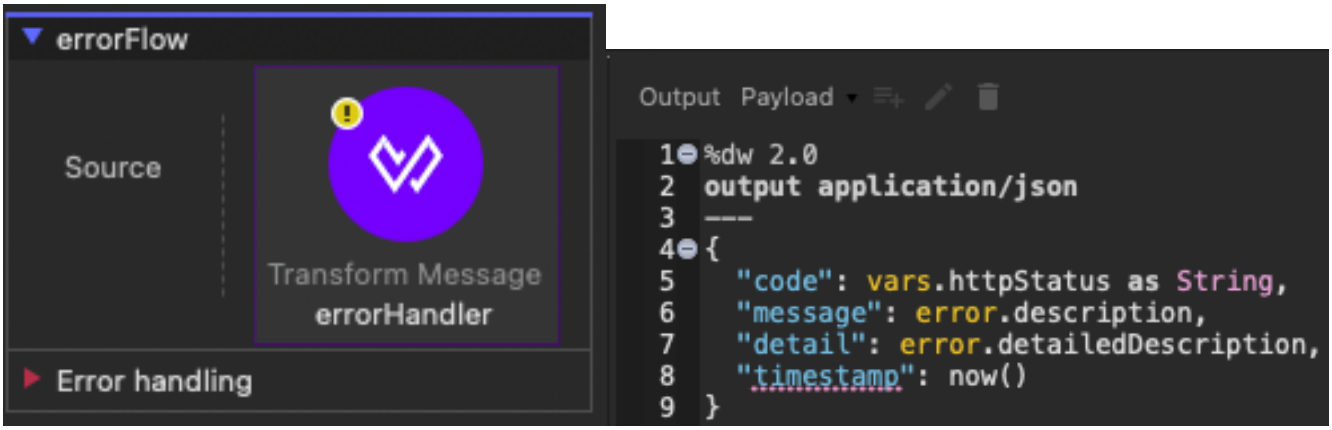


Manejo de errores

El manejo de errores se llevó a cabo en la interfaz, bajo el flujo principal que contiene el apiKit. Cada error contiene un set variable que está configurado para crear una variable llamada httpStatus, y el valor es el valor de cada error.



Todo esto está referenciado a un subflujo que únicamente contiene un transform message que extrae los valores en el orden que se había definido en la API.



De esta manera, se logró que cada código de error se diera como mensaje extraído de la variable previamente creada con el mensaje del error y la descripción, añadiendo también un timestamp para que sea más evidente cuándo y cómo ocurrió el error.

Configuraciones globales / config.yml

Ya que la aplicación corría bien con todas las especificaciones necesarias, se crearon dos nuevos archivos: el primero, un config.yaml que contiene la información relevante para las configuraciones de algunos conectores, y otro archivo de “Mule Configuration File” llamado Global, que contiene todas las configuraciones de los elementos globales de los conectores.

Para el config.yaml, se agregaron tres secciones: el http, el httprequest y el autodiscovery (que se verá más adelante). Estas configuraciones facilitan el orden y el entendimiento de las configuraciones globales.

```yaml
http:
  port: "8081"
  path: "/api/characters"

httprequest:
  host: "swapi.dev"

autodiscovery:
  id: "19720620"
```

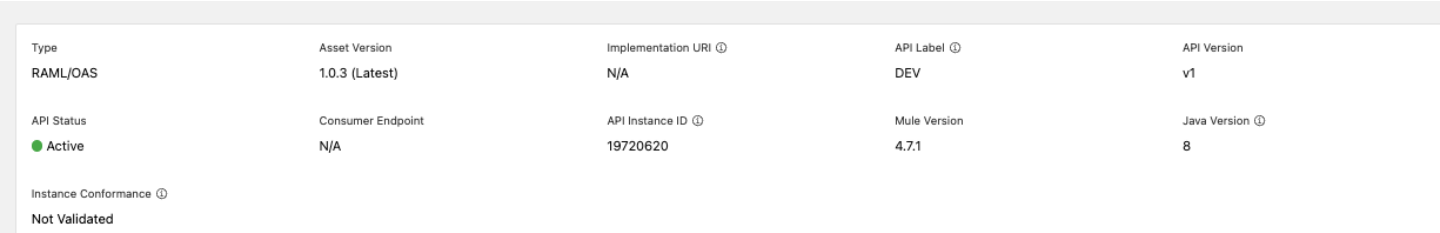
Para el archivo de las configuraciones globales, se movieron todas las configuraciones a esta sección, cada una configurada con respecto al config.yaml. Por ejemplo, la configuración del HTTP Listener: en lugar de escribir el puerto dentro de la configuración, se escribió ${http.port}, que hace referencia al config.yaml.

Utilizar estos archivos de configuración facilita que, si en el futuro se tuviera que cambiar el puerto o se tuvieran que hacer algunos ajustes dentro de las configuraciones globales, se pueda hacer todo desde un solo config.yaml.

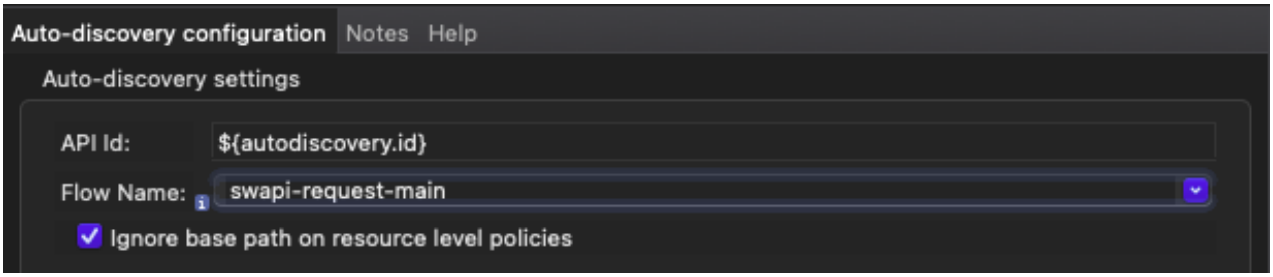
Despliegue de API

Configuración Autodiscovery

Una vez con todo configurado, se creó la instancia de API Manager para desplegarla como Mule 4. Una vez con esta instancia creada, se extrajo el API Instance ID que posteriormente se utilizaría para el despliegue en autodiscovery.

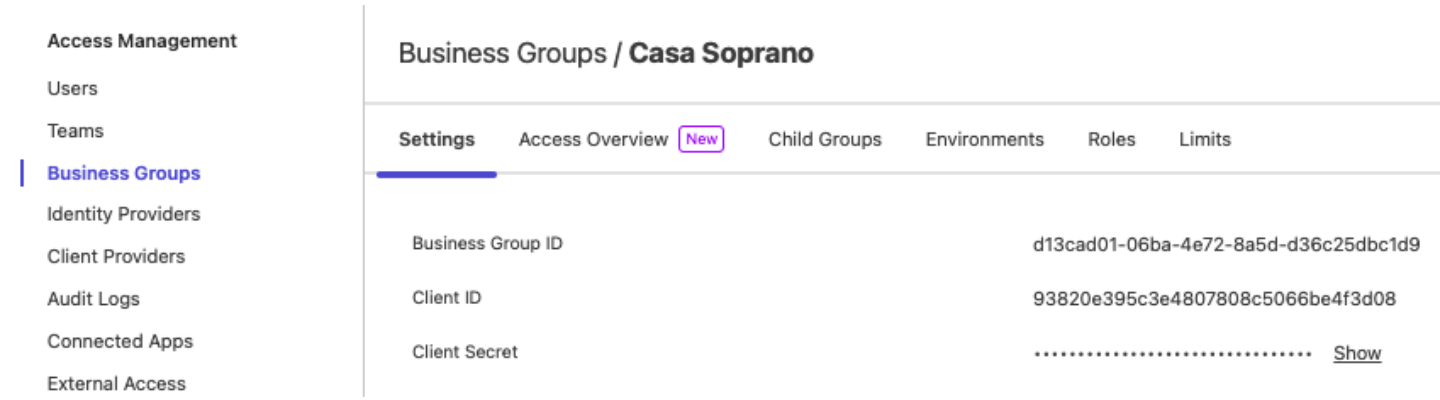


Con el ID en el archivo de config.yaml, se agregó esa sección y se configuró el autodiscovery dentro de las configuraciones globales.



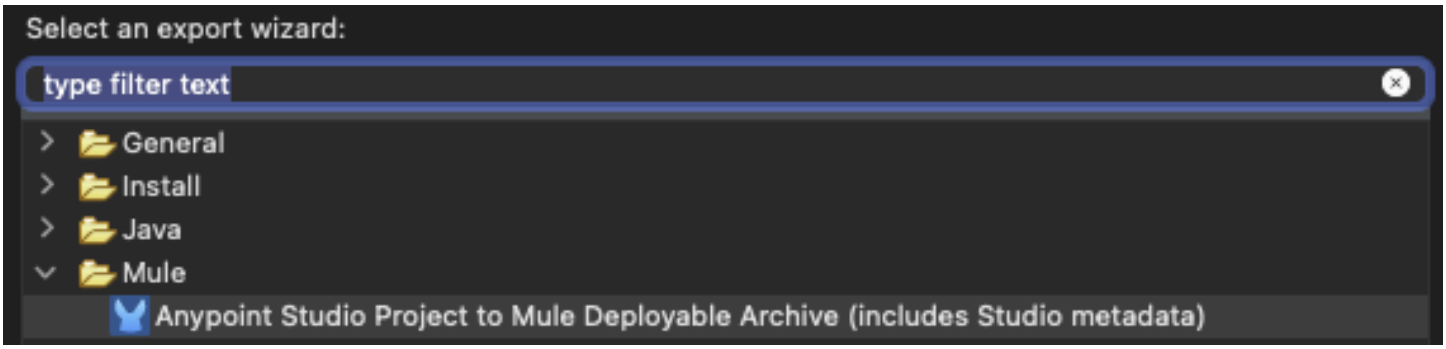
Esto sirvió para la integración automática de la API para la gestión y aplicación de políticas de seguridad (que se verán más adelante) sin la necesidad de una intervención manual una vez que la API fue desplegada.

Otro de los problemas que se abordó fue la necesidad de un secret y un client ID, los cuales se extrajeron de Access Management.

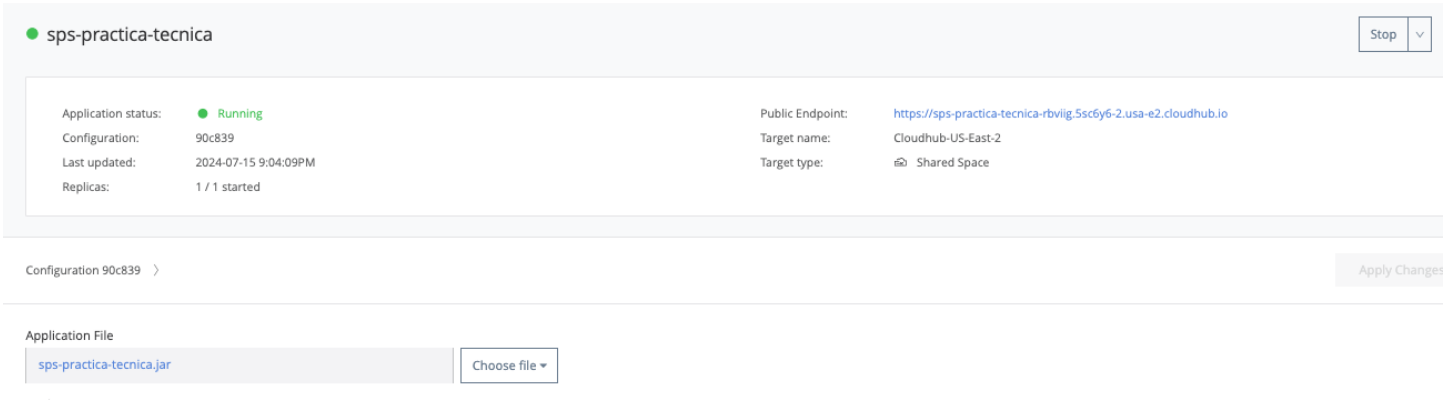


Despliegue Cloud Hub

Una vez con la aplicación funcionando y la API desplegada en Exchange, el siguiente paso sería desplegar la aplicación en CloudHub de manera que corriera para cualquier persona y no únicamente con mi host local. Para esto, fue necesario extraer el archivo que corre la aplicación.



Con el archivo en formato correcto, fue posible subir la aplicación a CloudHub para realizar un buen despliegue.

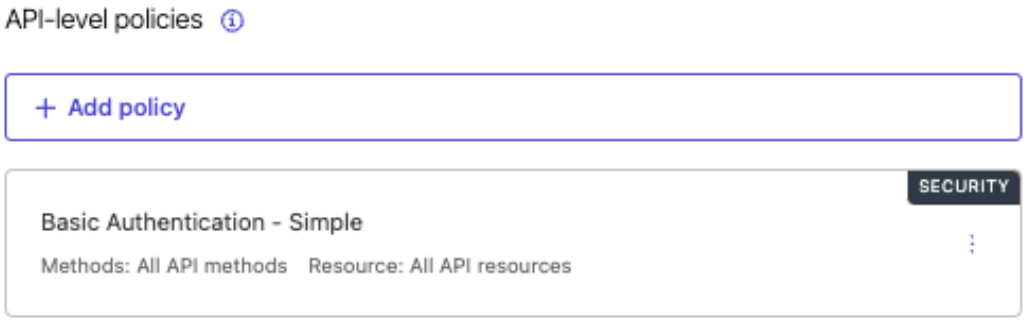


Este despliegue consta de una política de seguridad que se abordara más adelante.

Administración de API

Política de seguridad

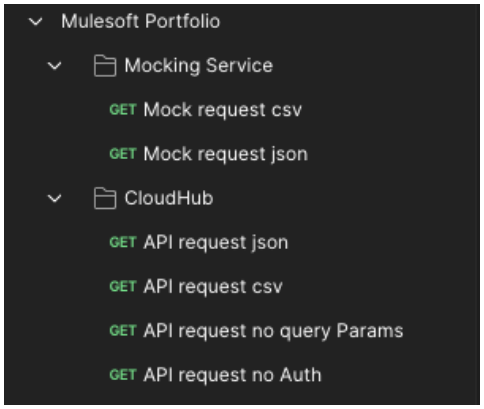
Ya con la aplicación desplegada en CloudHub, se agregó una política de seguridad de autenticación básica dentro del API Manager. Esta política consta de un usuario y una contraseña para poder hacer el request a la aplicación.



Para poder otorgar el acceso a la aplicación, se debe proporcionar un usuario y contraseña que en este caso sería MulePortfolio y Password. Sin estas credenciales, no se puede hacer el request a la aplicación que fue desplegada en CloudHub.

Colección Postman

Para facilitar la visualización de los requests, se creó una colección en Postman, ya que esta permite guardar y mandar los requests ya creados. Se creó un proyecto llamado Mule Portfolio, con dos subcarpetas: una con los requests al Mocking Service para probar los resultados de los ejemplos de la API desplegada en Anypoint Exchange, y otra para probar los resultados de los requests a la aplicación corriendo en CloudHub con su factor de autenticación.



Esto garantiza que tanto la API como la aplicación desplegada en CloudHub funcionan de manera correcta y dan los resultados tanto con queryParam como con los headers que contienen si se requiere en formato JSON o CSV.

Conclusiones

El desarrollo de esta aplicación ha sido una experiencia valiosa para demostrar mis capacidades en la creación, implementación y despliegue de APIs. Durante este proyecto, he desarrollado soporte para formatos CSV y JSON, ajustando la lógica de la aplicación para manejar ambos formatos de manera eficiente. Además, he agregado funcionalidades adicionales, como la publicación en el portal público y la implementación de políticas de seguridad, para demostrar un manejo completo del ciclo de vida de una API.

Este proyecto no solo muestra mi habilidad para desarrollar soluciones técnicas, sino también mi capacidad para mejorar y optimizar la funcionalidad y seguridad de las aplicaciones. Las soluciones implementadas enriquecieron significativamente mi comprensión y dominio del desarrollo y despliegue de APIs, posicionándome como un profesional competente y preparado para enfrentar desafíos en el campo de la integración de sistemas.