

# MULESOFT PORTFOLIO



Emiliano Benavides Díaz

## Content Table

<b>Introduction.....</b>	<b>2</b>
Problem Statement .....	2
Approach .....	2
<b>Solution Development.....</b>	<b>3</b>
API Specification .....	3
Design Center .....	4
Publishing on Exchange .....	7
Public Portal .....	7
<b>API Implementation.....</b>	<b>8</b>
Interface .....	8
Implementation .....	9
Error Handling.....	14
Global Configurations / config.yml .....	15
<b>API Deployment.....</b>	<b>16</b>
Autodiscovery Configuration .....	16
Cloud Hub Deployment .....	17
<b>API Management.....</b>	<b>18</b>
Security Policy .....	18
Postman Collection .....	18
<b>Conclusions.....</b>	<b>19</b>

## Introduction

### Problem Statement

Develop an interface for a mobile application that displays basic information from the Star Wars universe, including existing movies, main characters, and new releases. On this occasion, the goal is to have information about all the characters from the movies in a CSV format so that the application can show this data in an Excel-like table, and the end user can compare the characteristics of the characters.

It is known that the data source is an API (SWAPI) that returns information in JSON format. The task is to design an API specification (with RAML or Swagger) and develop a Mule project that retrieves the information of ALL the characters from Star Wars provided by the SWAPI and delivers it in a CSV format, for example:

```
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
```

As a second requirement, the functionality must be tested to filter the results received in the CSV by the character's gender using a Query Parameter, for example:

```
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
```

### Approach

The way to approach the problem was straightforward. I needed to create an API specification in API Designer to filter the searches, meaning it would only accept requests with the defined parameters. Likewise, I created this API to handle error management, examples, and accepted parameters. Then, I deployed the API in Anypoint Studio and developed the implementation. This implementation contained data retrieval, transformation to the required format (JSON to CSV), and data cleaning to adjust it to the necessary requirements: name, height, mass, hair, color, skin color, eye color, birth year, gender, omitting irrelevant information like homework, species, etc.

Next, I deployed the application in Anypoint Exchange to be publicly available. Subsequently, I added the API to API Manager and deployed it on CloudHub.

Note:

The information is not necessarily in chronological order but is written to be easier to read.

# Solution Development

## API Specification

### SWAPI Request

#### Description

The purpose of the API is to retrieve information from SWAPI (Star Wars API) both in JSON format and CSV format, allowing query parameters and specifying which parameters the request accepts. Different messages are also defined, including examples and error handling, specifying the types and details of these messages.

#### Versions

- 1.0.0

#### Methods

- GET

#### Parameters

- Query Parameters
- Body Parameters

#### Examples of Requests and Responses

- Examples of output payloads

#### HTTP Status Codes

- 200 • 400 • 404 • 405 • 415 • 500

## Design Center

The API is divided into three formats:

**Root File:** In this file, everything necessary for the API's operation is specified. This file is swapi-request.raml.

**Examples Folder:** Contains RAML files for different examples, such as 400errorResponse.raml and charactersExampleCSV.raml, to facilitate the use of the API from the Mocking Service.

**Data Types Folder:** Contains datatype files to specify the requirements of the responses in the example files.

In the Root File, the method available for the application is specified, in this case, GET, along with the option to accept a query parameter for filtering by gender. By analyzing the information from SWAPI, we can see the different genders that exist, and we enumerate them so that if the gender is not within the query parameter, the request is not accepted. The enumerated genders are: male, female, n/a, hermaphrodite, and none..

```
1  #%RAML 1.0
2  title: Star Wars Characters API
3  mediaType: application/json
4
5  /characters:
6    displayName: Characters
7    description: Resource to manage characters
8    get:
9      description: Retrieve a list of Star Wars characters.
10     queryParameters:
11       gender:
12         description: Filter characters by gender.
13         type: string
14         required: false
15         enum:
16           - male
17           - female
18           - n/a
19           - hermaphrodite
20           - none
21     responses:
22       200:
23         body:
24           application/csv:
25             example: !include examples/charactersExampleCSV.raml
26           application/json:
27             type: !include datatypes/characterType.raml
28             example: !include examples/charactersExample.raml
29       400:
30         body:
31           application/json:
32             type: !include datatypes/errorType.raml
33             example: !include examples/400errorResponse.raml
34       404:
35         body:
36           application/json:
37             type: !include datatypes/errorType.raml
38             example: !include examples/404errorResponse.raml
39
40
```

Next, we create the parameters for the responses the API could provide, being 200, 400, 404, 415, and 500. For each response, the available responses are separated, which are application/json and application/csv. For each response, an example file was created, linked to a datatype file.

Within the examples folder, the files for each response are found, each containing an example of how the response would look for each response:

▼ examples

400errorResponse.raml

404errorResponse.raml

405errorResponse.raml

415errorResponse.raml

500errorResponse.raml

charactersExample.raml

charactersExampleCSV.raml

For the 200 response, a RAML was configured for both the CSV response and the JSON response, containing arrays made up of three objects to make it clearer when using them..

```
1  #%RAML 1.0 NamedExample
2
3  name,height(cm),weight(kg),hairColor,skinColor,eyesColor,birthYear,gender
4  Luke Skywalker,172,77,blond,fair,blue,19BBY,male
5  C-3P0,167,75,n/a,gold,yellow,112BBY,n/a
6  R2-D2,96,32,n/a,white\, blue,red,33BBY,n/a
```

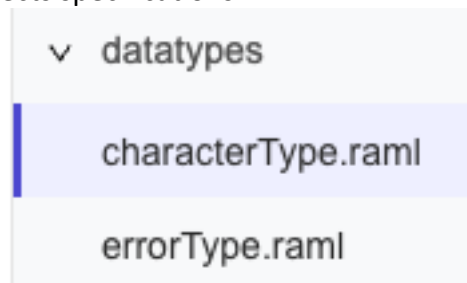
```
#%RAML 1.0 NamedExample
[
  {
    "name": "Luke Skywalker",
    "height(cm)": 172,
    "weight(kg)": 77,
    "hairColor": "blond",
    "skinColor": "fair",
    "eyesColor": "blue",
    "birthYear": "19BBY",
    "gender": "male"
  },
  {
    "name": "C-3P0",
    "height(cm)": 172,
    "weight(kg)": 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"
  }
]
```

These objects were defined based on the requirements for information about the characters. That is, it was defined that each object within the array should contain: name, height (cm), weight (kg), hairColor, skinColor, eyesColor, birthYear, and gender.

While for the error responses, the error code, message, detail, and a timestamp were defined to facilitate error tracking.

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

Finally, for the data types section, we created two folders: one that contains the specifications of the character objects and another for the error objects specifications.



For characterType.raml, it was specified which properties each part of the object within the character array should contain. In this case, for each attribute like name, height, etc., both the type was defined (if it was a string it would be text, or if it was an integer, it would be a numeric value), and if it was a required field or not. A brief description was also added to make it easier to understand.

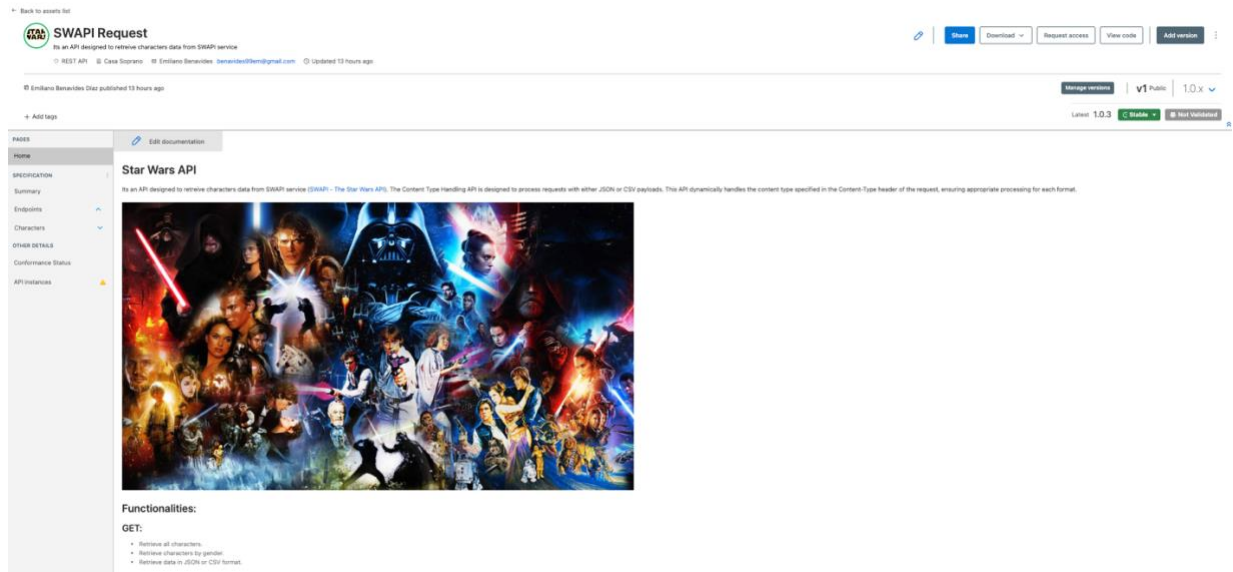
```
1  #%RAML 1.0 DataType
2
3  type: array
4  items:
5    type: object
6    properties:
7      name:
8        type: string
9        required: true
10       description: "Character name."
11      height(cm):
12        type: integer
13        required: true
14        description: "Character height in cm."
15      weight(kg):
16        type: integer
17        required: true
18        description: "Character weight in kg."
19      hairColor:
20        type: string
21        required: true
22        description: "Character hair color."
23      skinColor:
24        type: string
25        required: true
26        description: "Character skin color."
27      eyesColor:
28        type: string
29        required: true
30        description: "Character eyes color."
31      birthYear:
32        type: string
33        required: true
34        description: "Character birth year."
35      gender:
36        type: string
37        required: true
38        description: "Character gender."
```

For the error section, we used errorType.raml, where it was defined that the error message would be an object and the required fields of the message, which in this case would be: code, message, detail, and timestamp; specifying that each one was a string, which was necessary, and adding a brief description.

By structuring the API this way, it not only ensured the correct functioning of the API, but it also facilitated the organization and management of different errors, such as an error that did not allow a request to be added to the API in CSV format since it was not defined within the body of the 200 response.

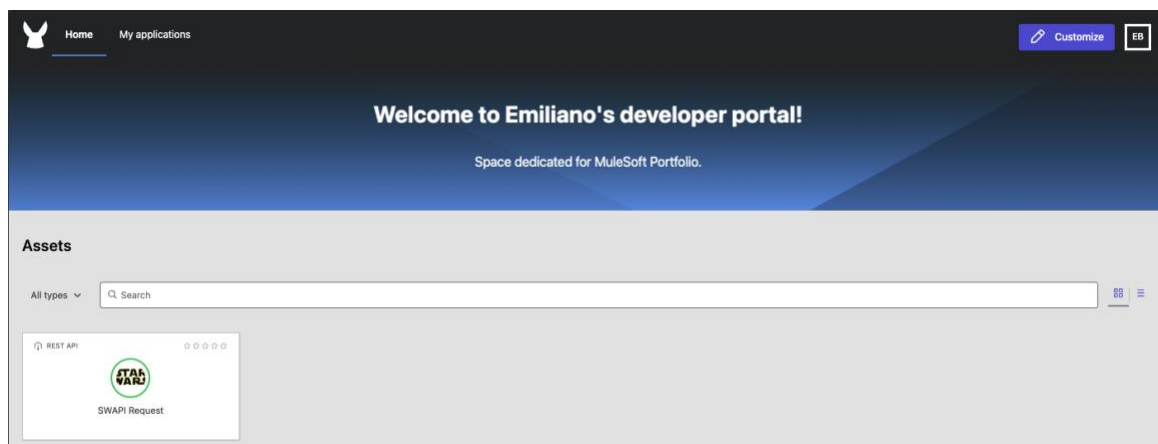
## Publishing on Exchange

To publish the API, Anypoint Exchange was used. Once this version was published, a brief description was added that includes both a description of what the API does, as well as its functionalities, contact information, and a logo to facilitate its search.



## Public Portal

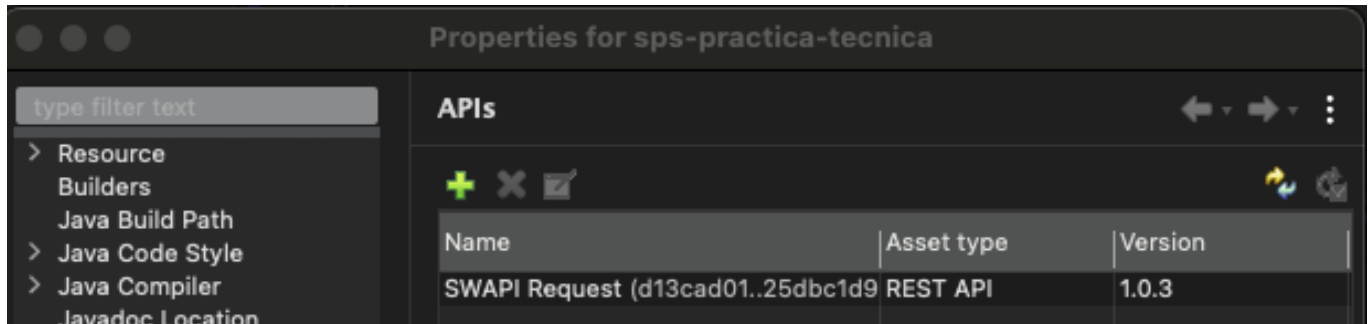
For the public portal where the API is located, it was customized with different information to highlight the page, adding both the logo, the color palette, and a brief description of the portal containing the “SWAPI Request” API.



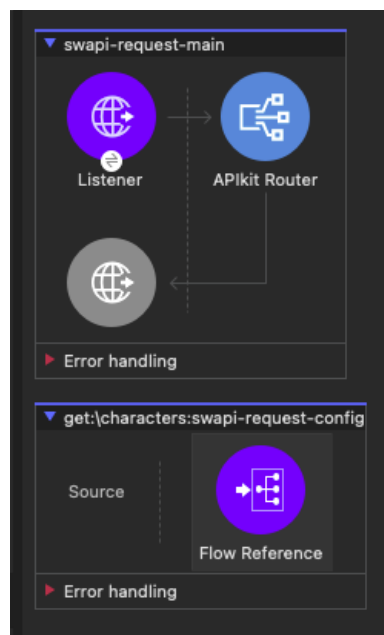
## API Implementation

### Interface

Once the API was created and deployed in Anypoint Exchange, a new project was created in Anypoint Studio named mule-portfolio-emiliano-benavides. Within this project, in the manage dependencies section, this API was implemented through Anypoint Exchange.



This implementation was used to generate the swapi-request-main flow, which will be part of the interface flow. This is responsible for tracking all requests to its different methods with a single listener; in this case, the only method is a GET.



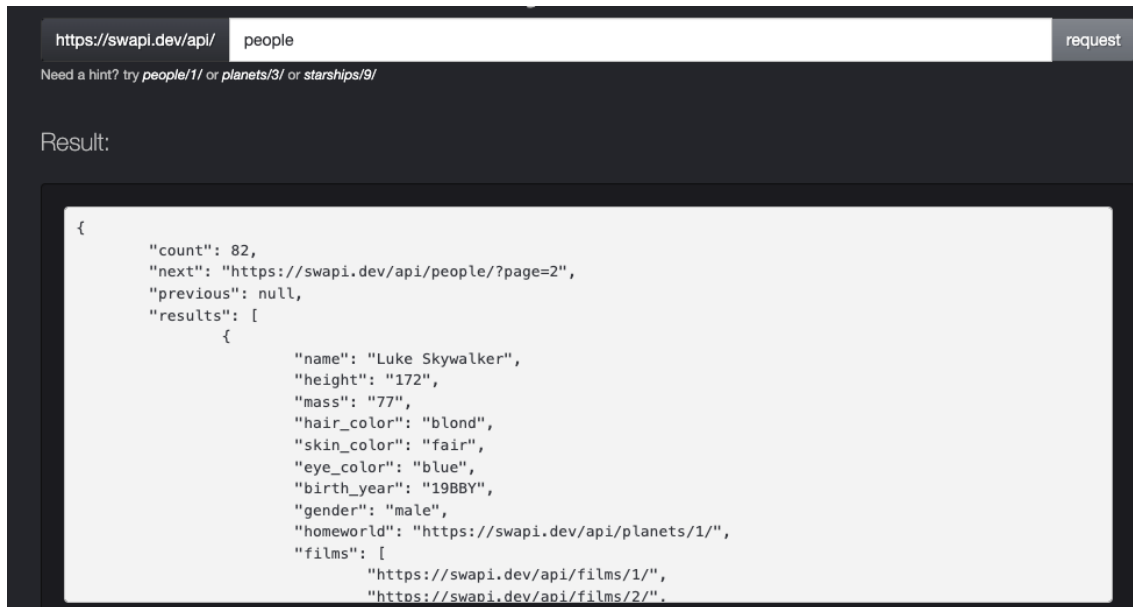
This facilitates extending the application logic, where the only thing within this Mule Configuration File is the main flow that tracks requests to their method, separates flows by methods, and facilitates error handling.



## Implementation

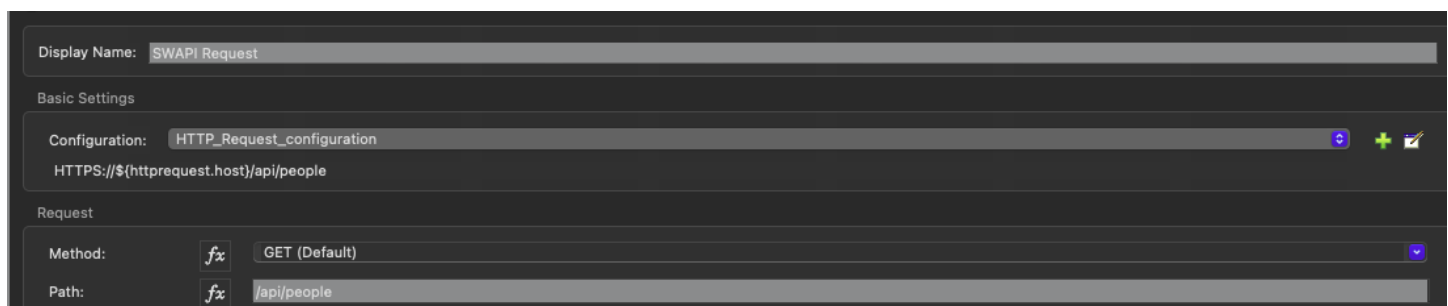
For the implementation, a new “Mule Configuration File” was created to separate the interface from the logic of the application, creating a new mainFlow that would handle this.

The first thing that needed to be done was to query SWAPI (SWAPI - The Star Wars API) to extend how the information was given and thus follow the solution. This page provides information divided into Starships, Homeworld, Vehicles, Species, Films, and what we were interested in, which was “People.” Within this “People” section, an object containing count (the number of characters), next (with the URL to the next page), previous (with the URL to the previous page), and results, which contains an array of different objects, each with the character’s specifications.



With this information, the URL from which to make the request was obtained, which is “https://swapi.dev/api/people/”, and from which part of the payload the character information had to be extracted.

Having knowledge of how the problem should be approached, in the mainFlow within the implementation, a Request was added. This request was configured based on what needed to be extracted, which in this case was the people section.



Once the information was obtained, it was noted that the complete list of characters was not given, but it was given by pages. Initially, it was attempted to extract the URL from the next section to get the URL to the next page, but this was becoming very complicated, so it was approached differently.

With count, it was possible to know how many characters were to be extracted, and by viewing the data and passing all the pages within SWAPI, it was noted that each page pattern has 10 characters, and each page contains 10 characters. With this information, the next step was to calculate how many pages there are, and a transform message was implemented to calculate the same.

```
Output Payload ▾ ⌵ ✎ 🗑️
1 1= %dw 2.0
2 2 output application/json
3 3 ---
4 4 totalPages: (1 to ceil(payload.count/10)) as Array|
```

The formula extracts from the payload the count information, which contains the total number of characters. This number is divided by 10 since each page had 10 characters. To address the problem of non-whole numbers, the formula was used to round this number up to get the total number of pages. Everything was extracted into an array to use it for each further down.

For the for each, the configuration was added to use the array previously extracted, which is payload.totalPages, as this contains an array with the total number of pages, causing the for each to repeat the necessary times.

Display Name:

Settings

Collection:

f<sub>x</sub>

payload.totalPages

Counter Variable Name:

counter

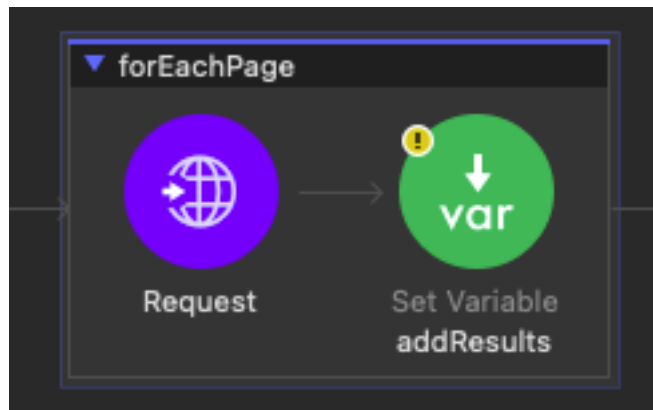
Batch Size:

1

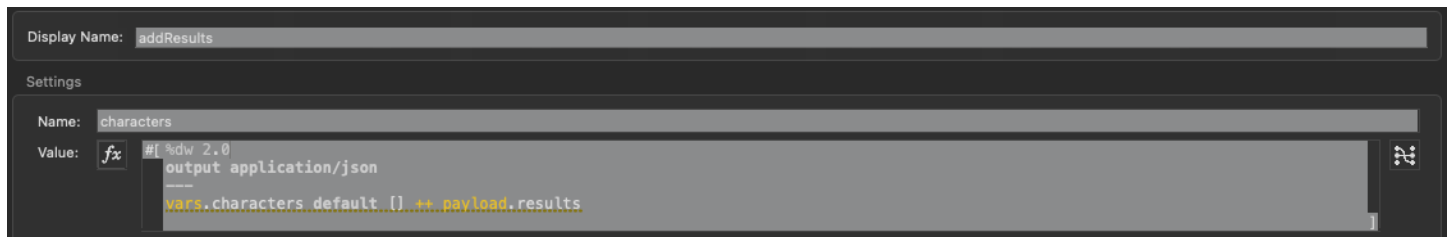
Root Message Variable Name:

rootMessage

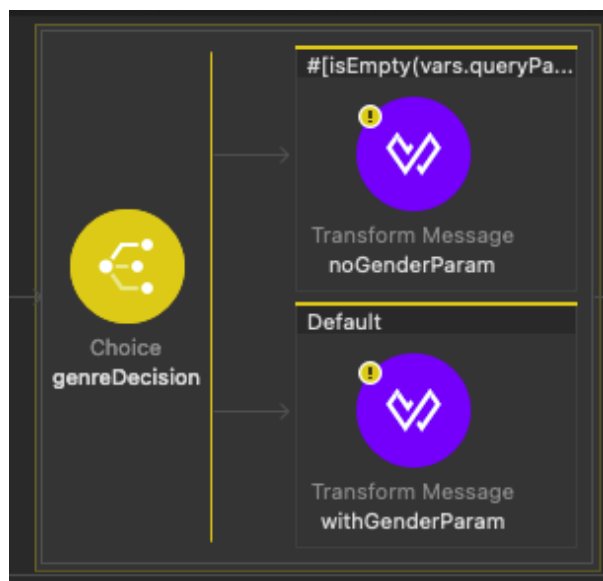
Within the for each, there are two modules: the first is a request that works exactly like the previous request, with the difference being that it does not extract the total character information but keeps all the payload information. Then, a variable was created to store the relevant information by adding a Set Variable.



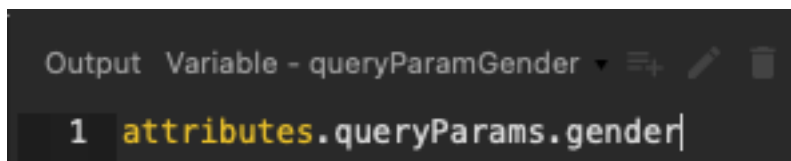
By means of DataWeave, a formula was implemented to add the characters. This formula consists of two parts: `vars.characters default []` which is responsible for creating the variable and putting it into an array, and the second part, which is `++ payload.results`, which extracts the information from the results of the payload, where the character information is found, and sums it as an array.



Thanks to these two functions, the complete information can be extracted. The next step was to add the function with which the filtering by gender could be done using a choice.



For the choice, a new variable was needed that would be the query parameter, so a new transform message named `setVariables` was created at the beginning of the flow, which extracts the attributes from the query parameter `gender` and creates a variable called `QueryParamGender`.

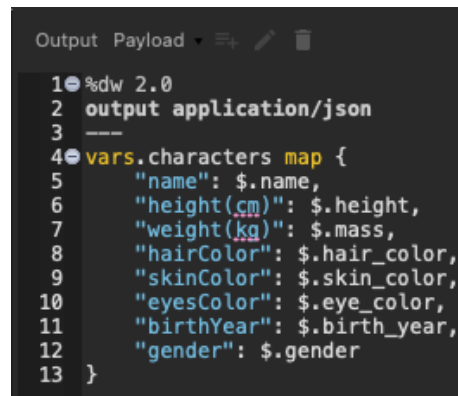


With this new variable, filtering could be done in the choice. The function written within the choice is `#[isEmpty(vars.queryParamGender)]`, this to differentiate if that variable was given or not, since if it does not have the variable, all information still needs to be extracted.



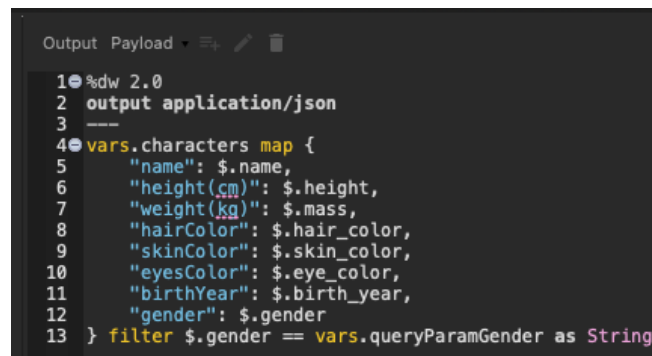
Now, the problem to be addressed is that it is already differentiated between whether it has a queryParam or not, but now it is necessary to clean the information according to the needs. Each option of the choice has a transform message that handles this function.

For the case where the request does not have a queryParam, it goes to a transform message that is responsible for extracting the necessary information, that is, filtering the information stored in the variable to extract only the necessary character data using a map, thanks to the condition previously set that the variable contains them in an array.



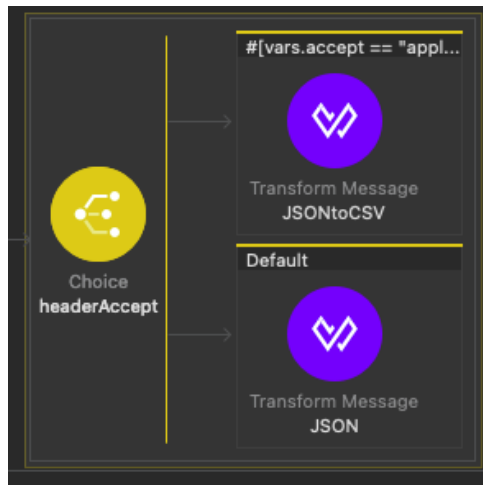
```
1 %dw 2.0
2 output application/json
3 ---
4 vars.characters map {
5     "name": $.name,
6     "height(cm)": $.height,
7     "weight(kg)": $.mass,
8     "hairColor": $.hair_color,
9     "skinColor": $.skin_color,
10    "eyesColor": $.eye_color,
11    "birthYear": $.birth_year,
12    "gender": $.gender
13 }
```

For the case where it has a queryParam, another transform message is used just like the previous one, with the difference that in this transform message the filter formula is used `gender == vars.queryParamGender as String`, this to ensure that the obtained information is cleaned, comparing the gender with the variable previously set as gender.



```
1 %dw 2.0
2 output application/json
3 ---
4 vars.characters map {
5     "name": $.name,
6     "height(cm)": $.height,
7     "weight(kg)": $.mass,
8     "hairColor": $.hair_color,
9     "skinColor": $.skin_color,
10    "eyesColor": $.eye_color,
11    "birthYear": $.birth_year,
12    "gender": $.gender
13 } filter $.gender == vars.queryParamGender as String
```

Now, with the complete information, whether it has a queryParam or not, the next step would be to provide the option to extract both in CSV format or in JSON format, so another choice was added.



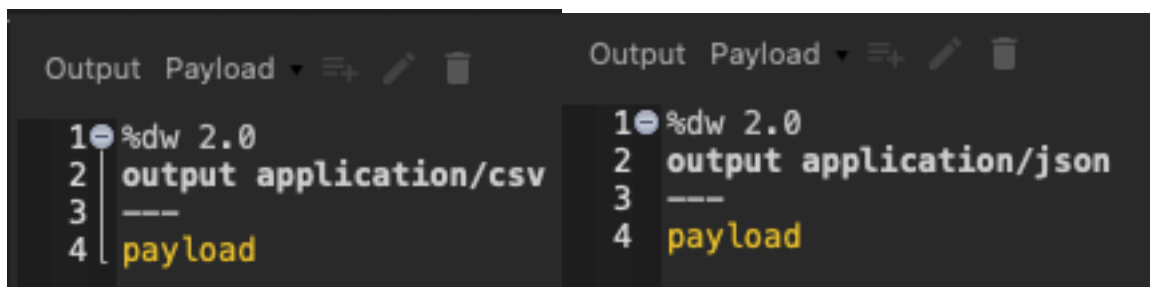
For this choice, another variable was needed to help differentiate which format was required. So, in the first transform message responsible for extracting the gender variable, a new variable was added to the headers. This variable is called accept, so in the headers within the query, accept can select application/json or application/csv.

Now, with the new variable in the choice, it can be distinguished with the formula `#[vars.accept == "application/csv"]`.

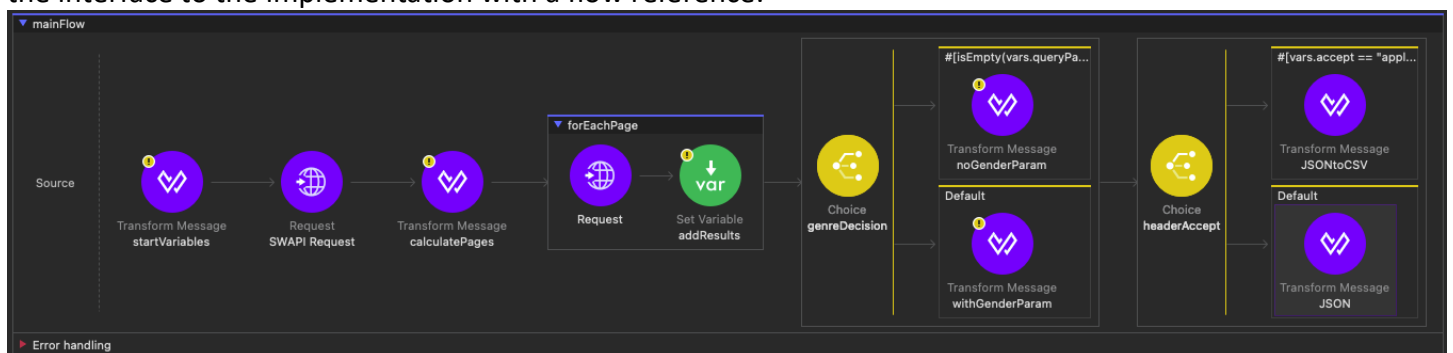
```
Expression: f: #[vars.accept == "application/csv"]
```

That is, if the header contains application/csv, it goes to that option, and if not, it extracts it in JSON format.

Now, with the information, each transform message converts the information to the desired format.

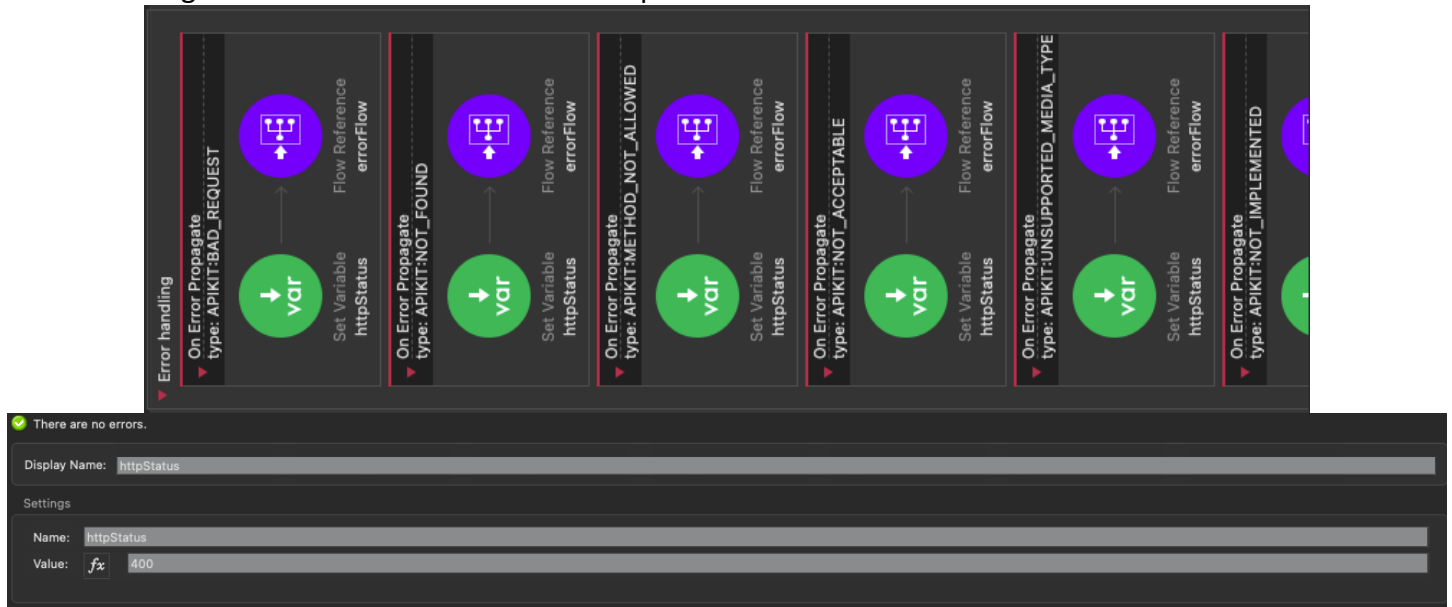


Ensuring that the search can filter both by gender and by format. Meanwhile, if nothing is specified, it still returns a result in JSON as the list of all characters regardless of gender, obtaining a clean flow that is referenced from the interface to the implementation with a flow reference.

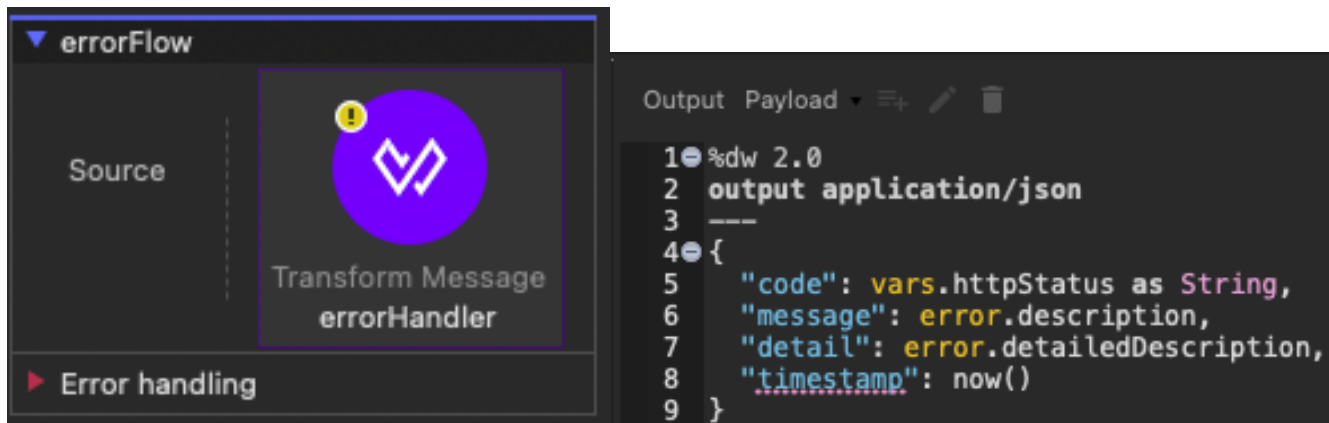


## Error Handling

Error handling was done in the interface, under the main flow containing the apiKit. Each error contains a set variable configured to create a variable called `httpStatus` with the value of each error.



All this is referenced to a subflow that only contains a transform message that extracts the values in the order defined in the API.



In this way, each error code was managed to be given as an extracted message from the previously created variable, with the error message and description, also adding a timestamp to make it clearer when and how the error occurred.

Since the application runs well with all the necessary specifications, two files were created: the first, a config.yml containing relevant information for the configurations of some connectors, and another file called Global, which contains all the configurations of the global elements of the connectors.

For the config.yml, three sections were added: http, httprequest, and autodiscovery (which will be seen later). These configurations facilitate the order and understanding of the global configurations.

```
1 http:
2   port: "8081"
3   path: "/api/characters"
4
5 httprequest:
6   host: "swapi.dev"
7
8 autodiscovery:
9   id: "19720620"
```

For the global configurations file, all configurations were moved to this section, each configured with respect to config.yml. For example, the HTTP Listener configuration: instead of writing the port within the configuration, \${http.port} was written, referencing config.yml.

The screenshot shows a configuration interface with the following fields:

- Protocol: HTTP (Default)
- Host: All Interfaces [0.0.0.0] (default)
- Port: \${http.port}
- Read timeout: 30000

Below the fields is a table with the following columns: Type, Name, Description, and actions (Create, Edit, Delete).

Type	Name	Description	
HTTP Listener config (Configuration)	HTTP_Listener_config		Create
HTTP Request configuration (Configuration)	HTTP_Request_configuration		Edit
Configuration properties (Configuration)	Configuration_properties		Delete
API Autodiscovery (Configuration)	API_Autodiscovery		

Using these configuration files makes it easier to change the port or adjust some settings within the global configurations in the future, as everything can be done from a single config.yml.

## API Deployment

### Autodiscovery Configuration

Once everything was configured, an instance of API Manager was created to deploy it as Mule 4. Once this instance was created, the API Instance ID was extracted, which would later be used for the deployment in autodiscovery.

Type	Asset Version	Implementation URI ⓘ	API Label ⓘ	API Version
RAML/OAS	1.0.3 (Latest)	N/A	DEV	v1
API Status	Consumer Endpoint	API Instance ID ⓘ	Mule Version	Java Version ⓘ
● Active	N/A	19720620	4.7.1	8
Instance Conformance ⓘ	Not Validated			

With the ID in the config.yml file, this section was added and autodiscovery was configured within the global configurations.

Auto-discovery configuration Notes Help

Auto-discovery settings

API Id:

Flow Name:

☒ Ignore base path on resource level policies

This allowed for the automatic integration of the API for the management and application of security policies (which will be seen later) without the need for manual intervention once the API was deployed.

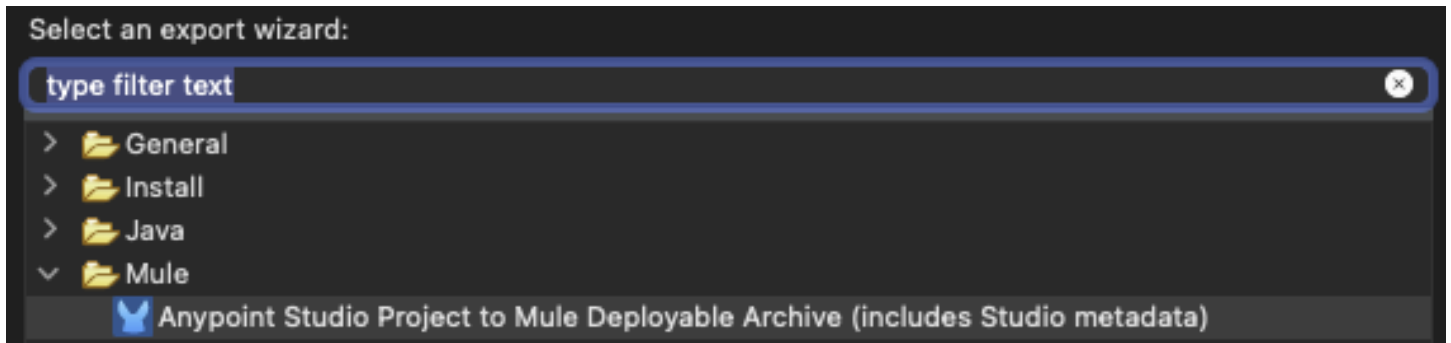
Another problem that was addressed was the need for a secret and a client ID, which were extracted from Access Management.

Access Management	Business Groups / Casa Soprano	
	Users	
	Teams	
	Business Groups	
	Identity Providers	
	Client Providers	
	Audit Logs	
Connected Apps	Settings	Access Overview <span>New</span> Child Groups Environments Roles Limits
	Business Group ID	d13cad01-06ba-4e72-8a5d-d36c25dbc1d9
	Client ID	93820e395c3e4807808c5066be4f3d08
External Access	Client Secret	..... <a href="#">Show</a>

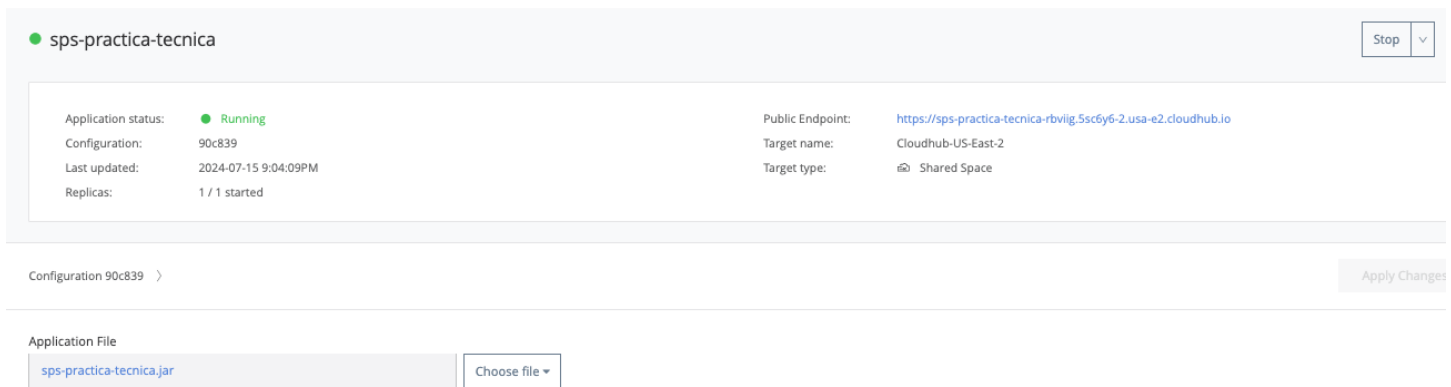


## Cloud Hub Deployment

Once the application was running and the API was deployed in Exchange, the next step was to deploy the application in CloudHub in a way that it runs for anyone and not just with my local host. For this, it was necessary to export the file that runs the application.



With the file in the correct format, it was possible to upload the application to CloudHub to perform a proper deployment.

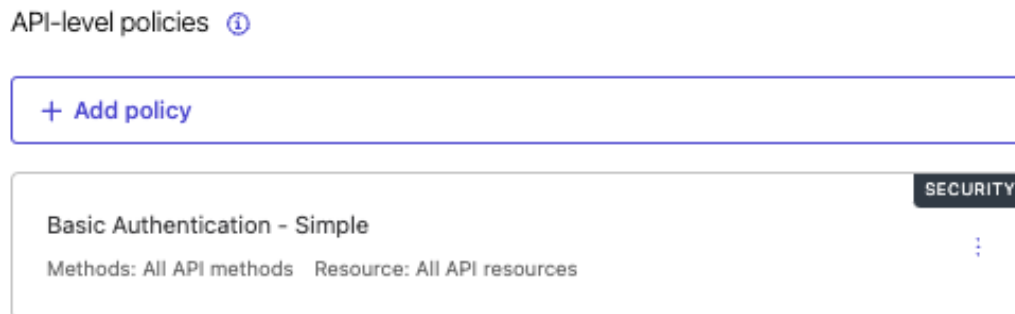


This deployment includes a security policy that will be addressed later.

## API Management

### Security Policy

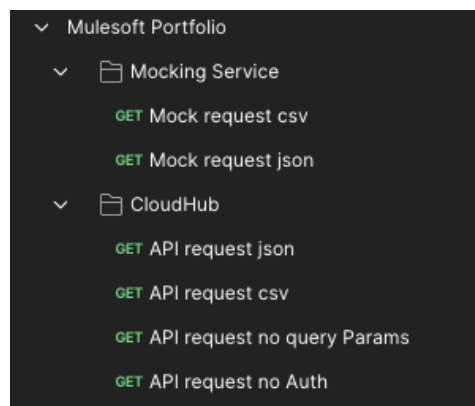
Once the application was deployed in CloudHub, a basic authentication security policy was added within API Manager. This policy consists of a username and a password to be able to make the request.



To grant access to the application, a username and password must be provided, which in this case would be MulePortfolio and Password. Without these credentials, the request cannot be made to the application that was deployed in CloudHub.

### Postman Collection

To facilitate the visualization of the requests, a collection was created in Postman, as it allows saving and sending the created requests. A project named Mule Portfolio was created with two subcategories: one with the requests to the Mocking Service to test the results of the examples of the API deployed in Anypoint Exchange, and another to test the results of the requests to the application running in CloudHub with its authentication factor.



This ensures that both the API and the application deployed in CloudHub function correctly and provide the results both with queryParam and with the headers that contain whether JSON or CSV format is required.

## Conclusions

The development of this application has been a valuable experience to demonstrate my capabilities in the creation, implementation, and deployment of APIs. During this project, I have developed support for CSV and JSON formats, adjusting the application's logic to handle both formats efficiently. Additionally, I have added extra functionalities, such as publishing on the public portal and implementing security policies, to demonstrate comprehensive management of the API lifecycle.

This project not only shows my ability to develop technical solutions but also my capacity to improve and optimize the functionality and security of applications. The implemented solutions significantly enhanced my understanding and mastery of API development and deployment, positioning me as a competent and prepared professional to face challenges in the field of systems integration.