Day 1

- Inference Hands-on
- Vectors Hands-on
- Webcrawler Hands-on

Inference Hands-on

Inference Connector Hands-on

As the Inference Connector is being certified as we deliver this training, it is not available on the Anypoint Exchange Portal.
Will will be using the latest open source release 0.5.7

OpenAI API KEY

As part of the exercise, the MAC team will sponsor an **API Key from OpenAI** to be used during the exercise.

```
sk-proj-IhCPniMttidFmGGZHrptPqQBxD8OgmdF0ocVMESXrsUCgmSyGI8o1ifMg9jlaUY
9fUDOXp62NpT3BlbkFJmavxgM7aSJj782OzF-J-9R54DwgkWgCHQPa6N7j2nTrr7kb6YtFX
ZwLWO5-om4nfCGFMEeEYEA
```

This API Key will expire on the 30th of June.
Please consider provisioning an Azure Open AI APi Key through

ACB (Inference)

ACB (Inference)

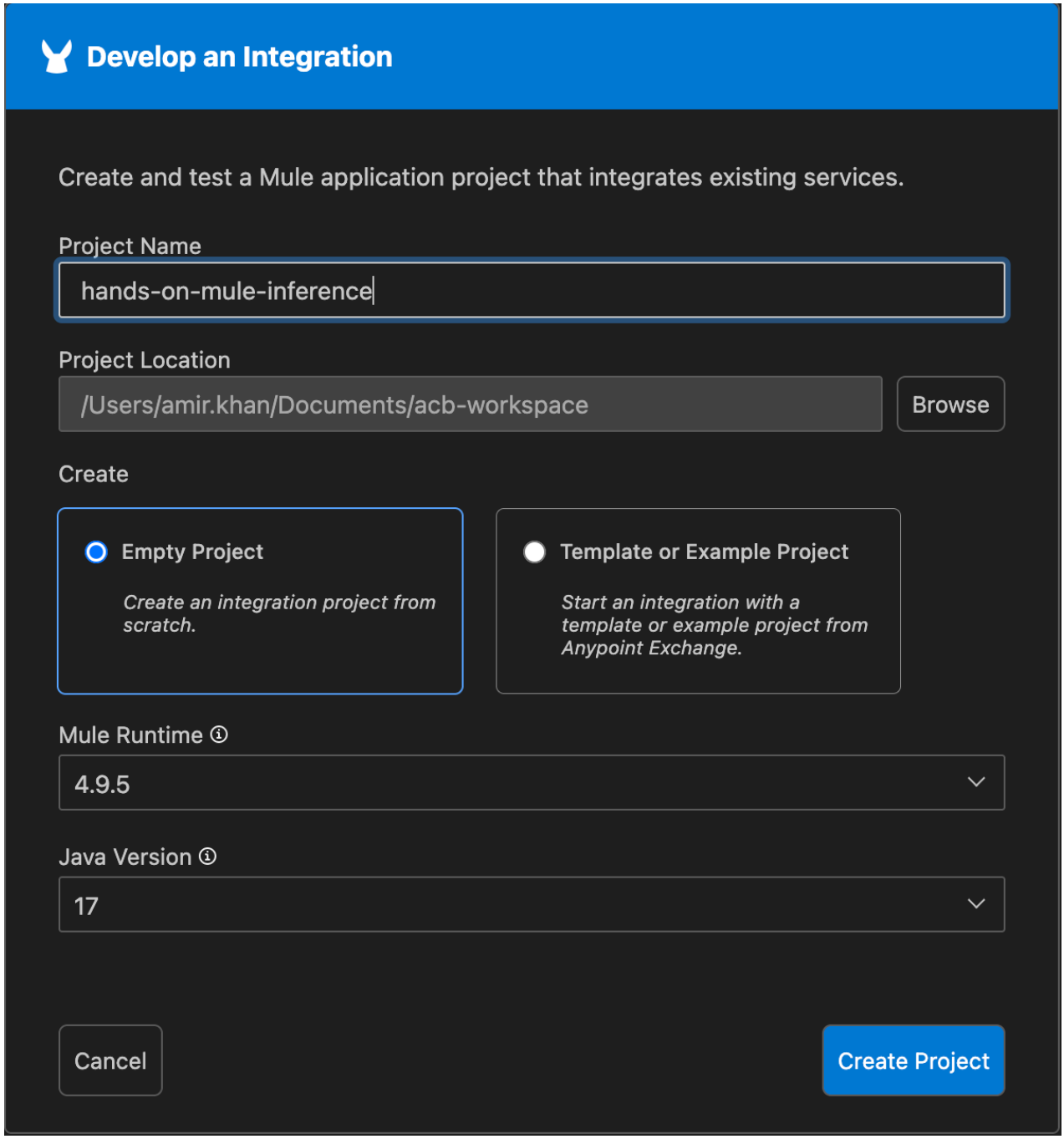
Inference Connector Hands-on

Exercise 1 - Simple Chat App

The purpose of this exercise is to demonstrate, how you can use the chat completions operation of LLM to implement a workflow in MuleSoft. In this exercise we are going to build a simple headless chat app with an LLM.

Step 1

Create a Mule Project `hands-on-mule-inference`

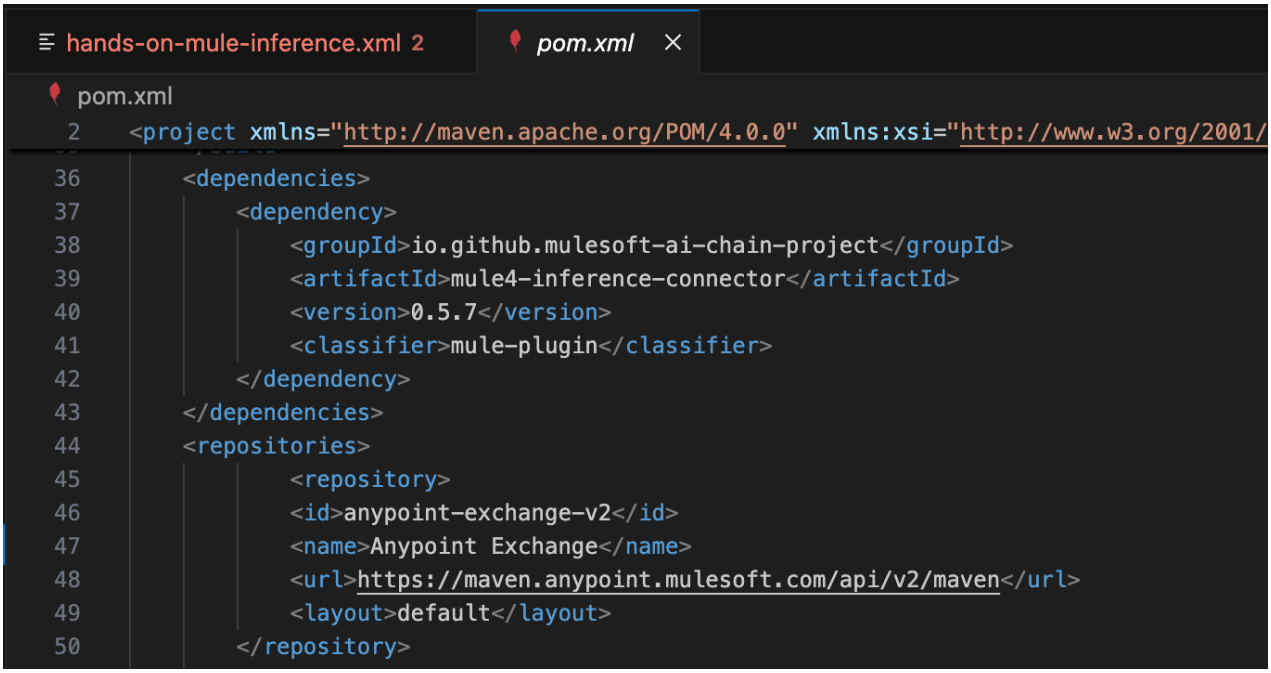


Step 2

Add the following dependency in the pom.xml
hands-on-mule-inference

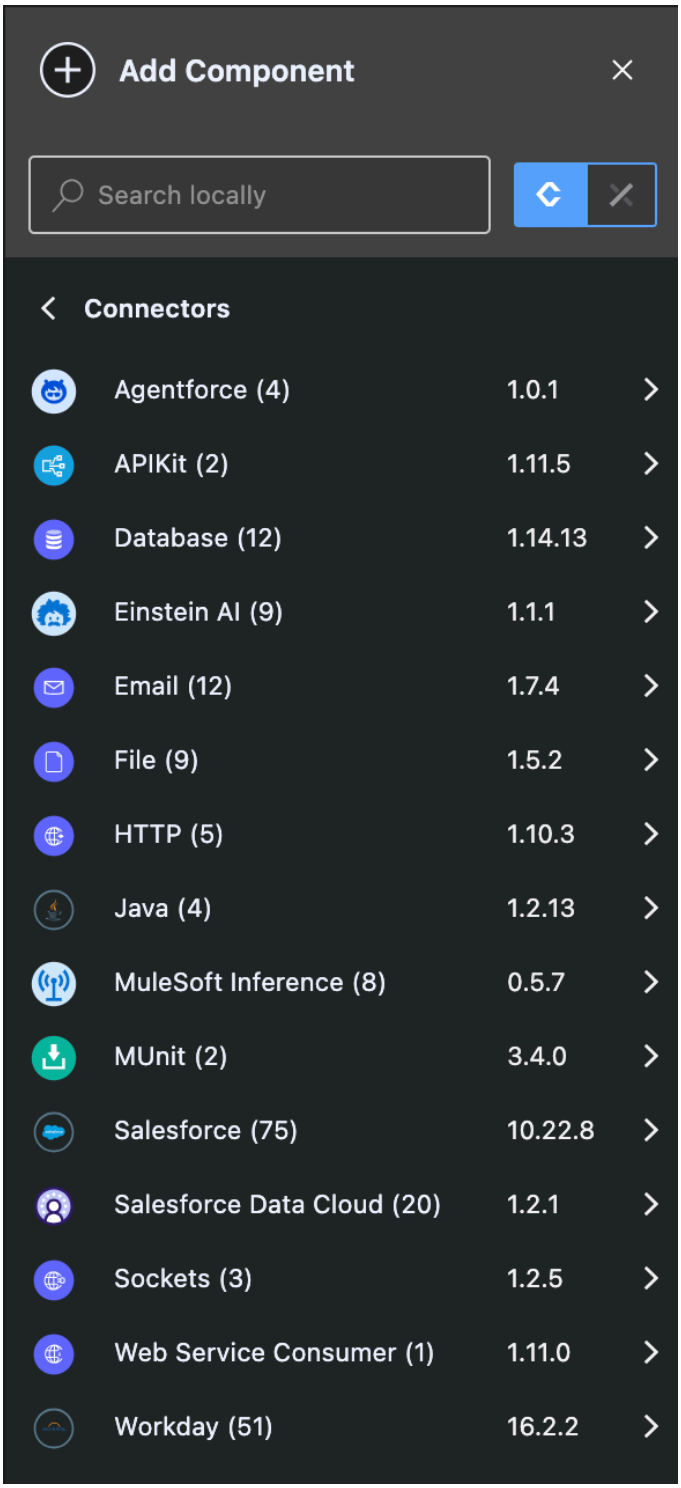
```
<dependency>
<groupId>io.github.mulesoft-ai-chain-project</groupId>
<artifactId>mule4-inference-connector</artifactId>
<version>0.5.7</version>
<classifier>mule-plugin</classifier>
</dependency>
```

Save the project.



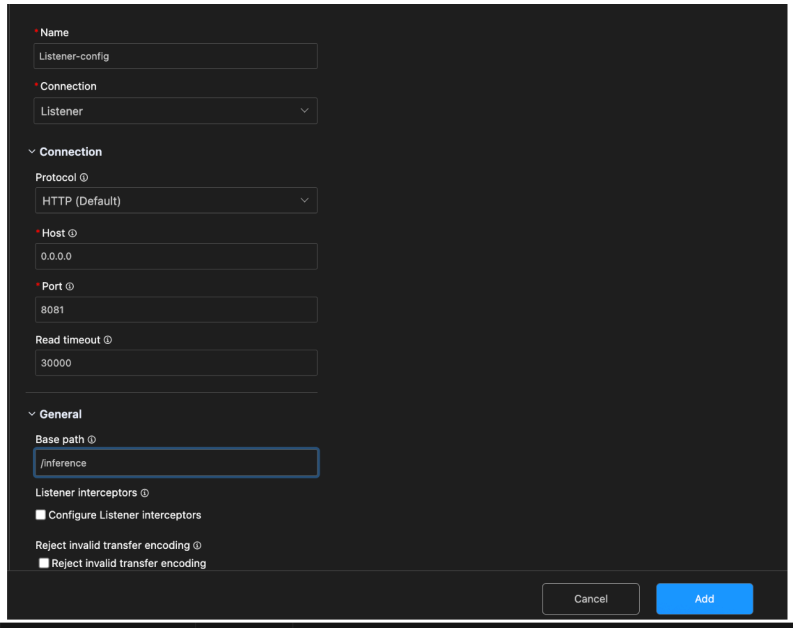
Step 3

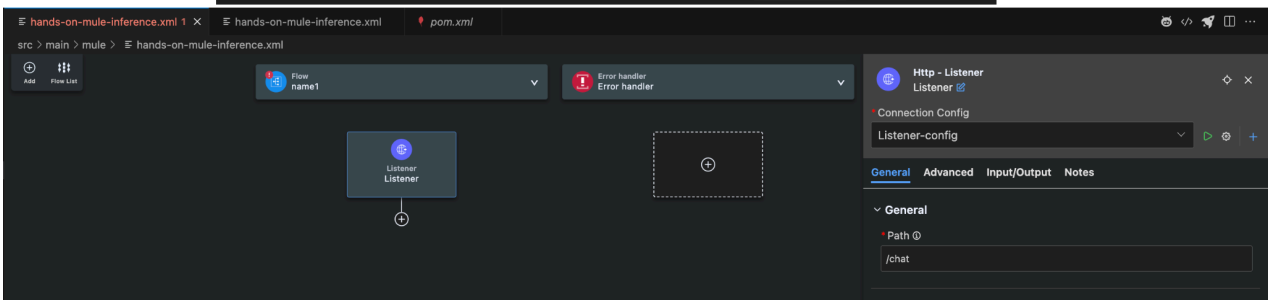
Check that MuleSoft Inference Connector is available in the Mule Palette



Step 4

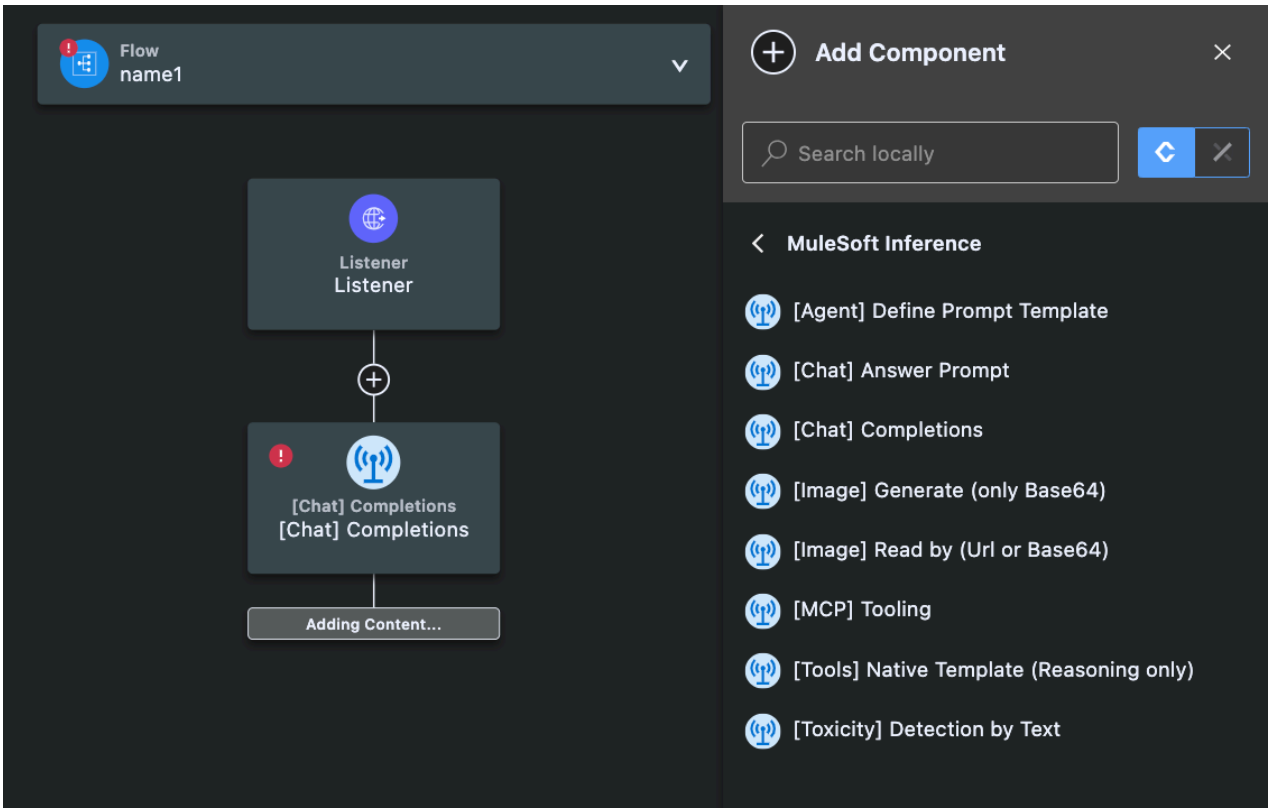
Drop a http listener (/inference/chat) to build a simple chat interaction using postman. Note: For ACB, please create a flow before dropping a processor.





Step 5

Drop a [Chat] Completions operation from Inference into the canvas.

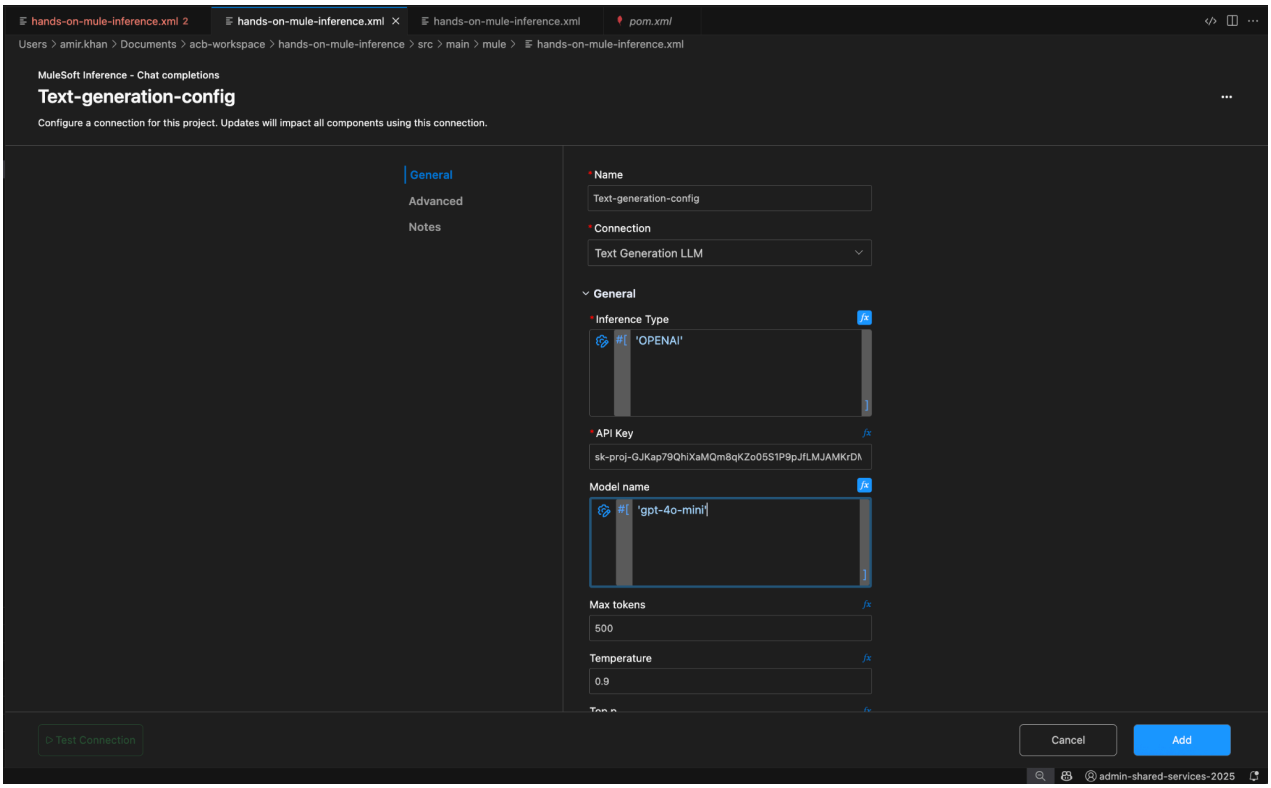


Step 6

Add a new configuration (Text-Generation) for Inference Connector for OpenAI.

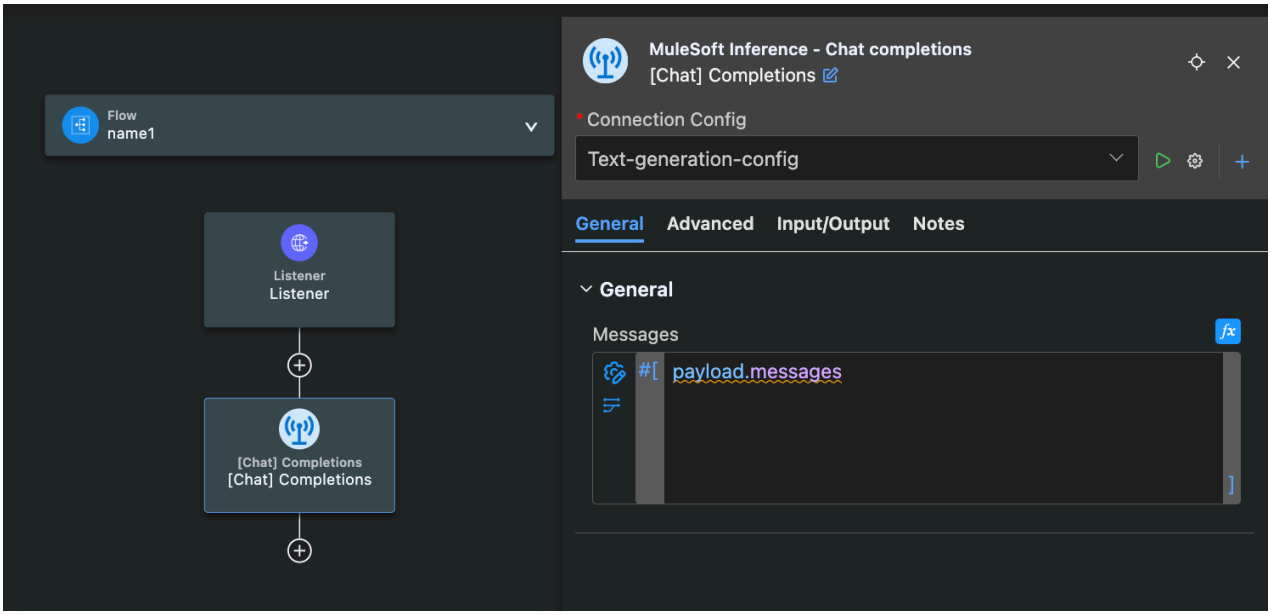
Inference Type: OPENAI
API Key: [copy from here]
Model Name: 'gpt-4o-mini'

leave the rest as is.
Click on the Test Connection and we should get 'Test connection successful' or 'Connection is valid' on ACB.
Note: If in ACB the drop down list doesn't populate, please use the expression mode to fill-in the data. For Anypoint Studio, after specifying the Inference Type, sometimes the list is not displayed (Model name parameter). And u are getting the below while trying to refresh the list:
"
Metadata resolution is taking longer than expected.
Click Studio services indicator to resolve.
"
The quickest fix is to restart Studio to restart the metadata resolution service.



Step 7

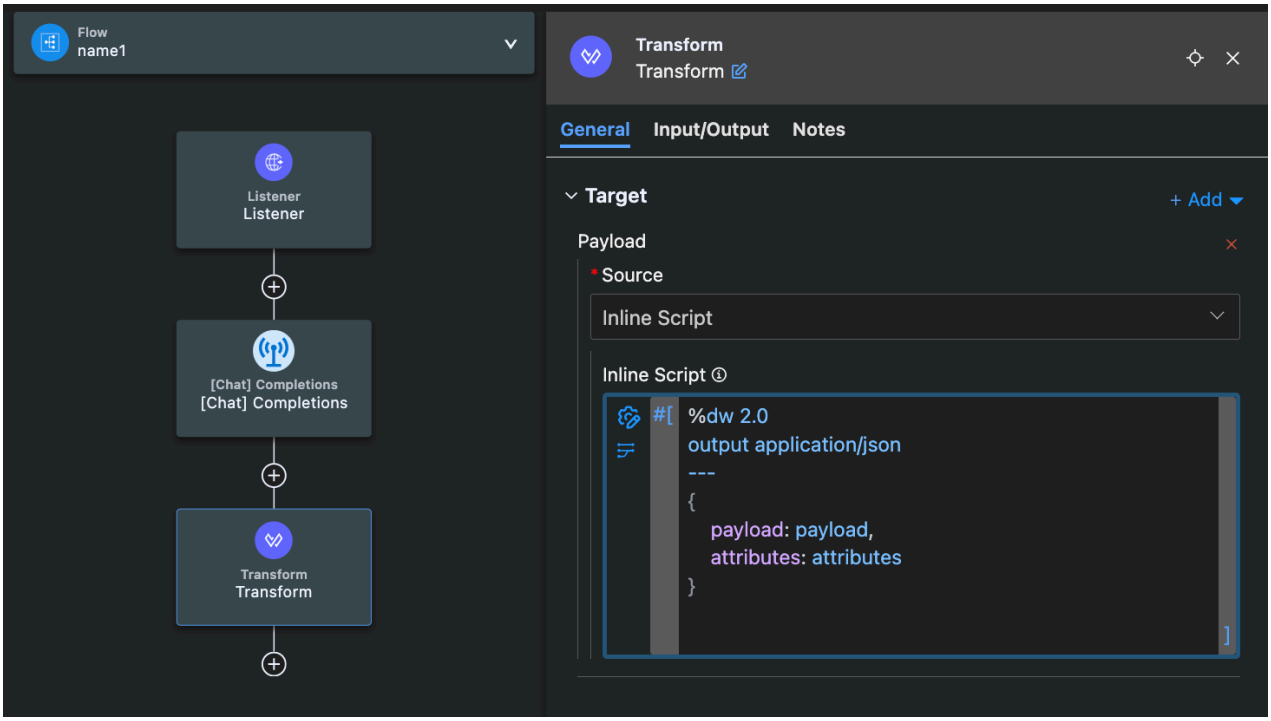
Select the [Chat] Completions operations and change the messages field to `payload.messages`



Step 8

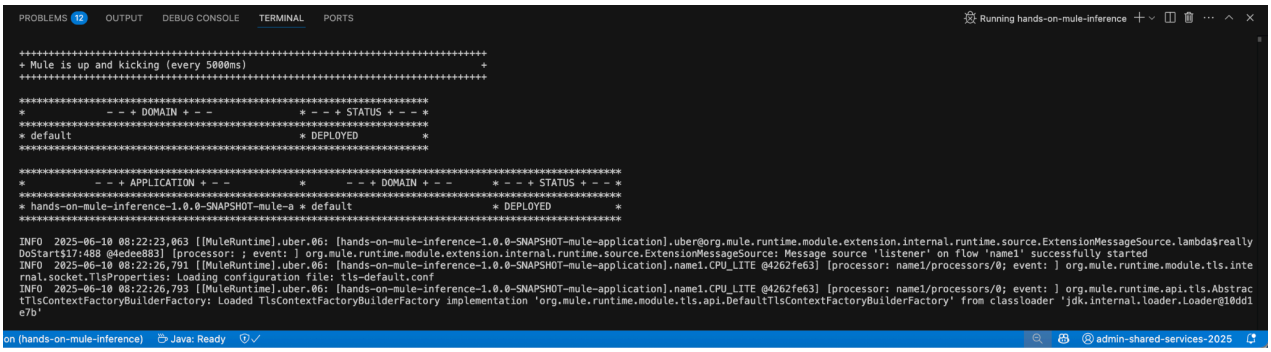
Add a Transform Message processor after [Chat] Completions with the following dataweave expression.

```
%dw 2.0
output application/json
---
{
payload: payload,
attributes: attributes
}
```



Step 9

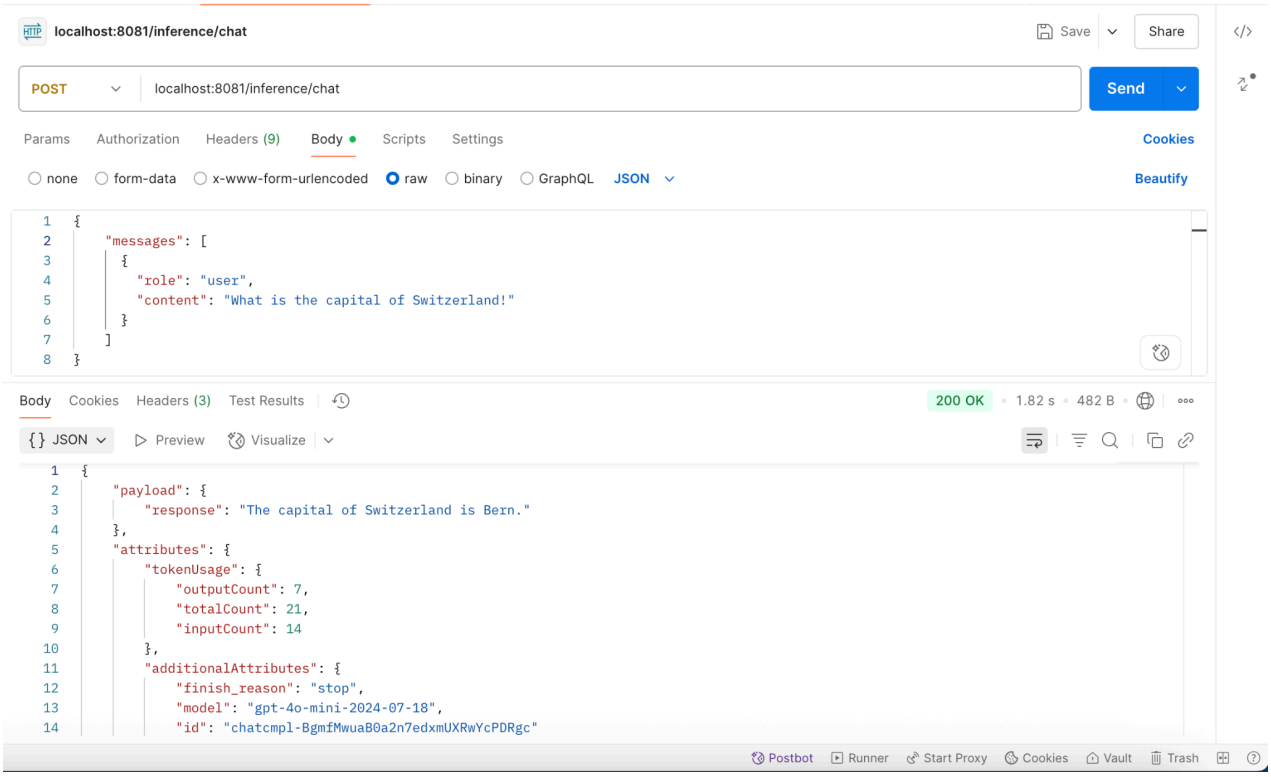
Deploy the application locally using the IDE (Studio or ACB)



Step 10

Use postman or any other rest api client to fire the following request:

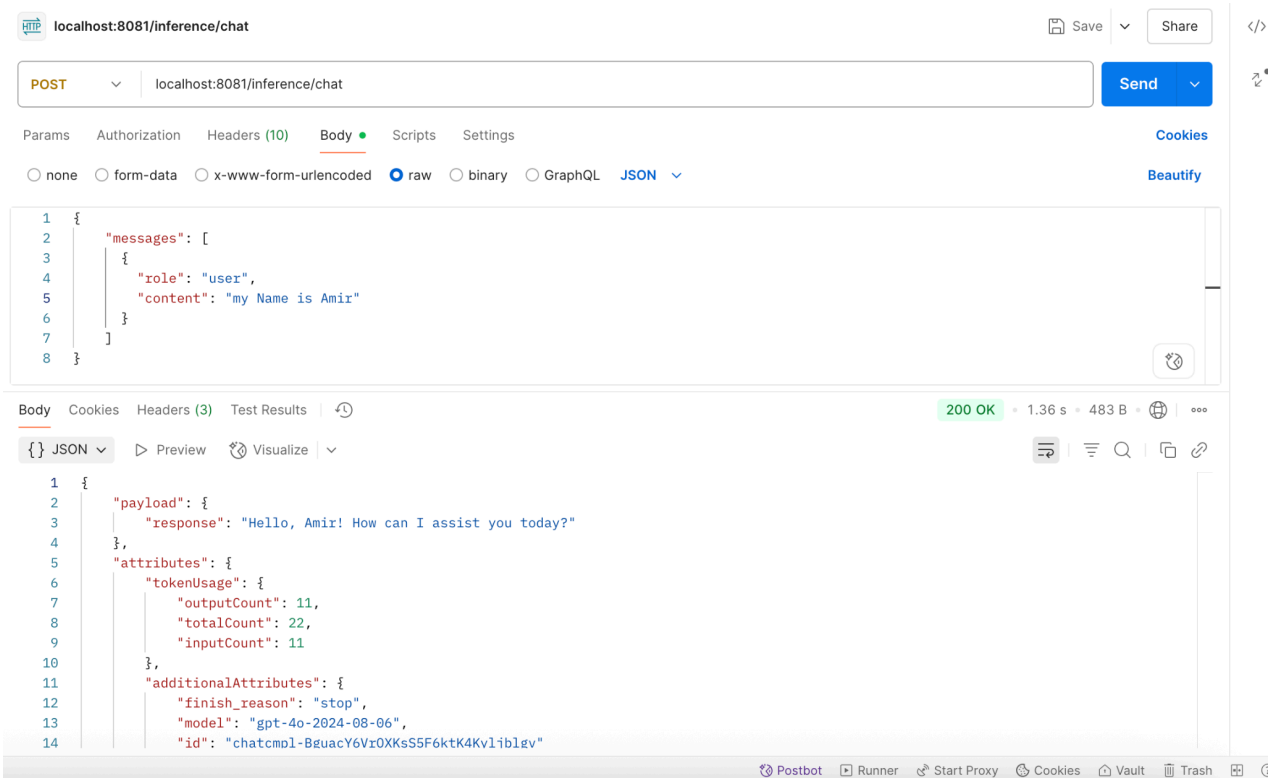
```
curl --location 'localhost:8081/inference/chat' \
--header 'Content-Type: application/json' \
--data '{
"messages": [
{
"role": "user",
"content": "What is the capital of Switzerland!"
}
]
}'
```



Step 11

Use postman or any other rest api client to fire the following request:

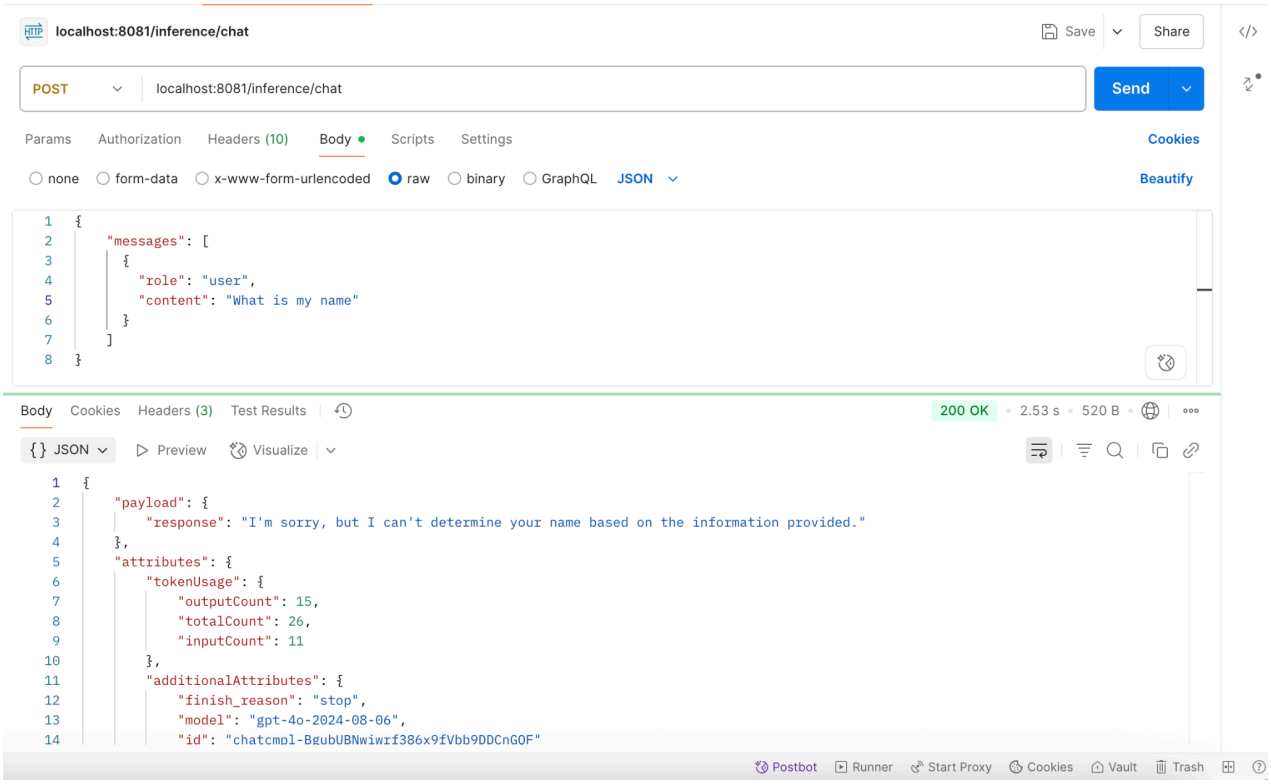
```
curl --location 'localhost:8081/inference/chat' \
--header 'Content-Type: application/json' \
--data '{
"messages": [
{
"role": "user",
"content": "My Name Is Amir"
}
]
}'
```



Step 12

Use postman or any other rest api client to fire the following request:

```
curl --location 'localhost:8081/inference/chat' \
--header 'Content-Type: application/json' \
--data '{
"messages": [
{
"role": "user",
"content": "What is my name"
}
]
}'
```



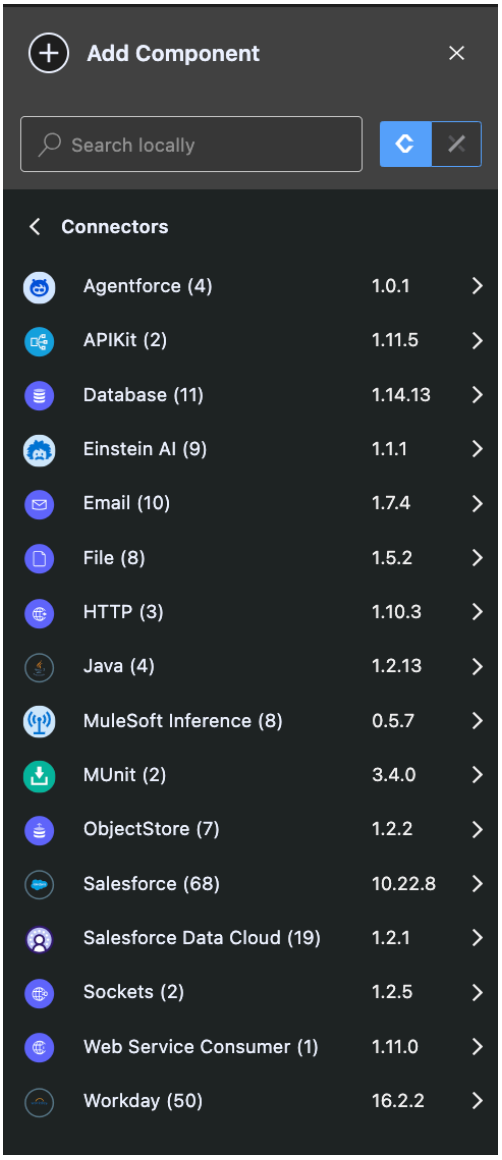
Exercise 2 - Enabling Chat History

As we are using the Chat Completions operation of an LLM, we can also provide history about the chat with the user. You can decide which data to be considered in the chat history when extending the chat app.

Step 1

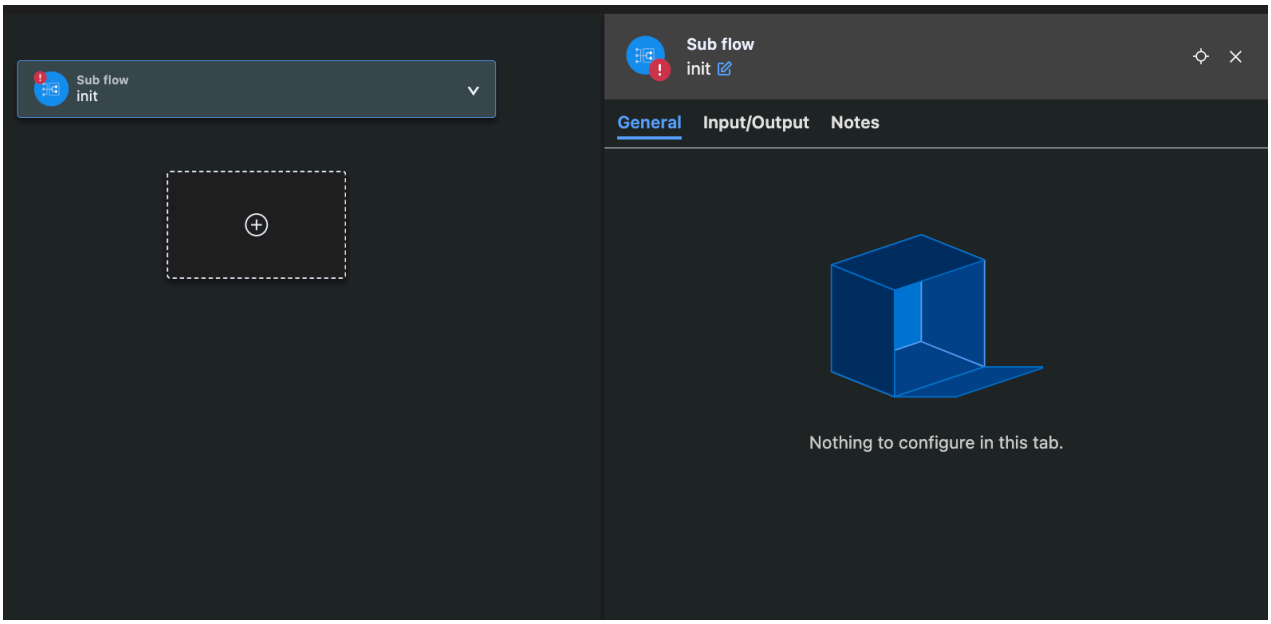
Open the `hands-on-mule-inference` app from exercise 1.
Drop the Object Store Connector into the Mule Palette or make sure it is available.
Once available, create a object store config either using UI or using the following expression in the xml.

```
<os:object-store name="Object_store" doc:name="Object store"
doc:id="75bf104d-4263-4eb8-a2ac-7be3e619c4b8" persistent="false"/>
```



Step 2

Create a new subflow called memory-retriever

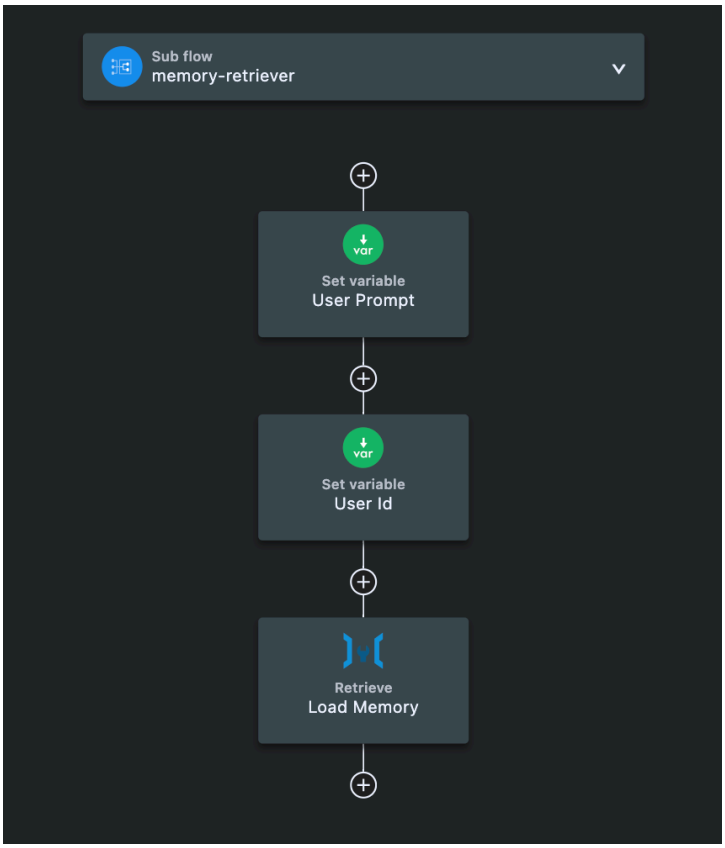


Step 3

Now we need to create a simple memory retriever for the chat history. The purpose of the simple memory retriever will be to evaluate the chat history by userId, and maxMessages, which are passed as Http Attributes to the app.

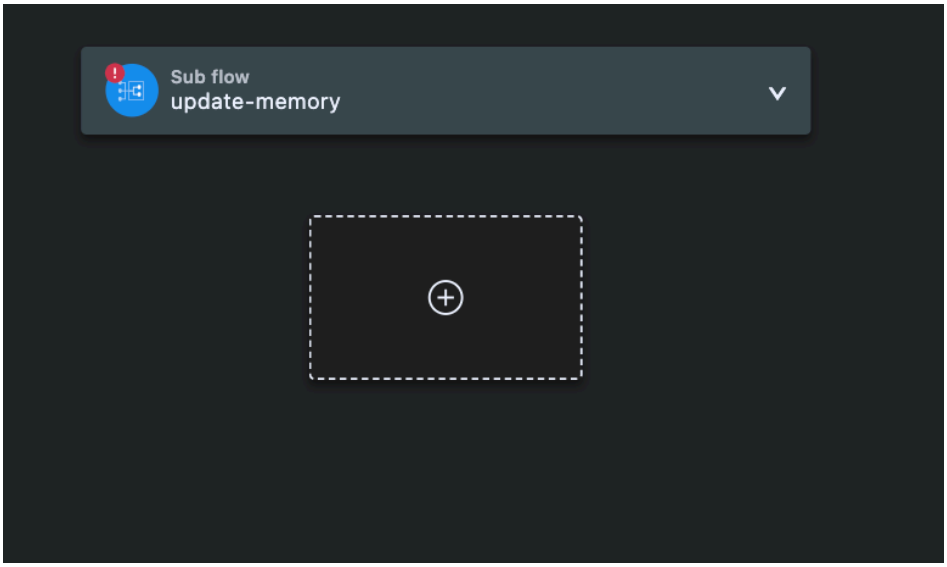
In order to achieve this, add the following variables and retriever (Object store) into the flow:

```
<set-variable value="#[payload.messages]" doc:name="User Prompt"
doc:id="76e70ed2-1b17-46fa-8e0a-327ae728158f"
variableName="userPrompt"/>
<set-variable value='#[attributes.headers."userId"]' doc:name="User Id"
doc:id="28824951-7f99-4313-b74f-50fd66fbd94b" variableName="userId"/>
<os:retrieve doc:name="Load Memory"
doc:id="4e5a4fb2-ca70-44dc-84f7-f5de36676a44" key="#[vars.userId]"
objectStore="Object_store" target="memory">
<os:default-value ><![CDATA[#[output application/json
---
{
chat_history: []
}]]]></os:default-value>
</os:retrieve>
```



Step 4

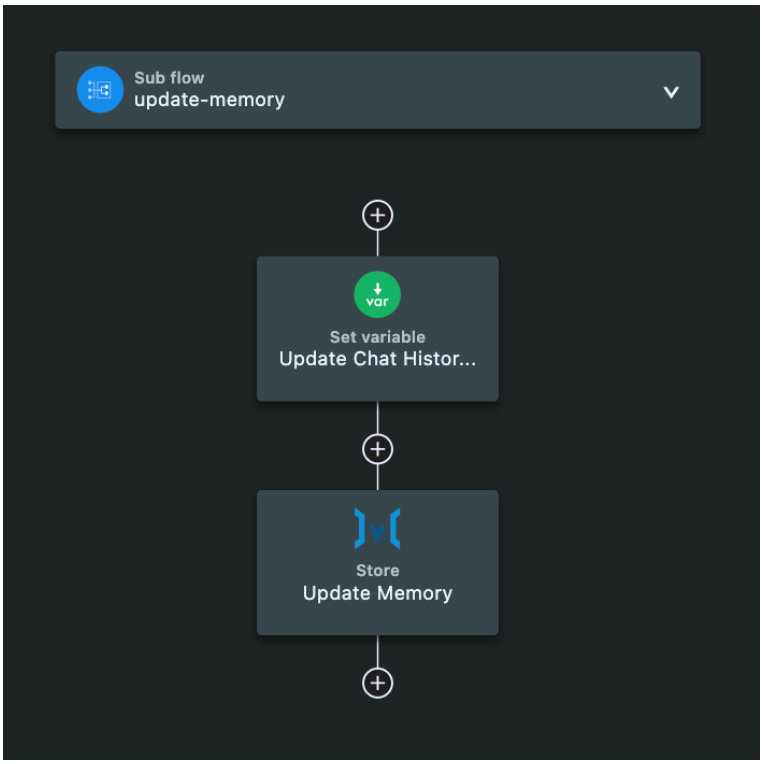
We need to create a new sub flow called "update-memory".



Step 5

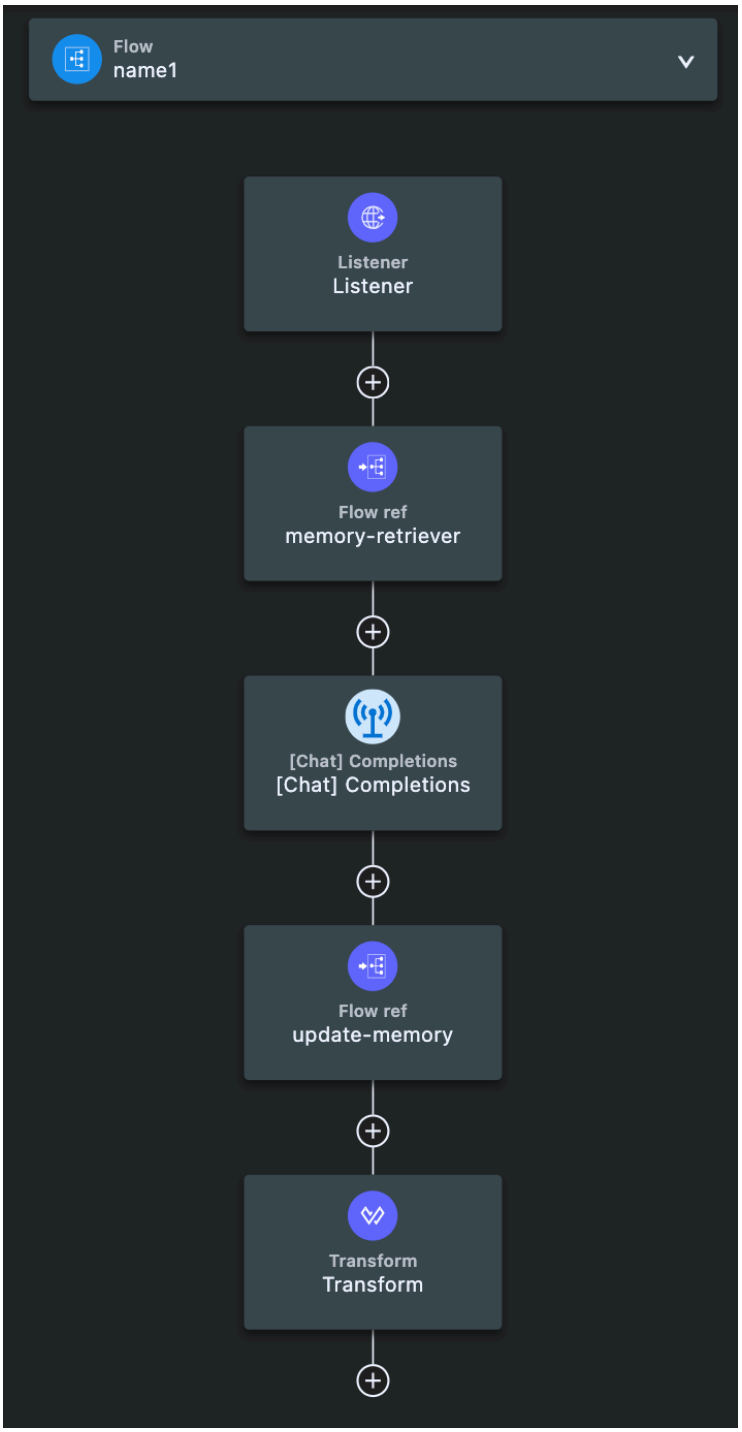
Now we need to update the memory retriever after the communication with the agent. This is done by copy / pasting the following dataweave code into the sub-flow "update-memory".

```
<set-variable value='#[output application/json&#10;---&#10;vars.memory
update {&#10; case history at .chat_history -&gt; (history default []) ++
[&#10; {&#10; role: "user",&#10; content: vars.userPrompt.content[0]&#10;
},&#10; {&#10; role: "assistant",&#10; content: payload.response default
""&#10; }&#10; ]&#10;}]' doc:name="Update Chat History"
doc:id="e7e9a10b-363a-4464-85f3-496f5116e52b" variableName="memory" />
<os:store doc:name="Update Memory"
doc:id="bf81989c-f94d-4e3c-84b4-fc86d1c0d6ad" key='#[vars.userId]'
objectStore="Object_store" >
<os:value ><![CDATA[#[vars.memory]]]></os:value>
</os:store>
```



Step 6

Add the flow references before (for memory-retriever) and after (for update-memory) the [Chat] Completions operation.



Step 7

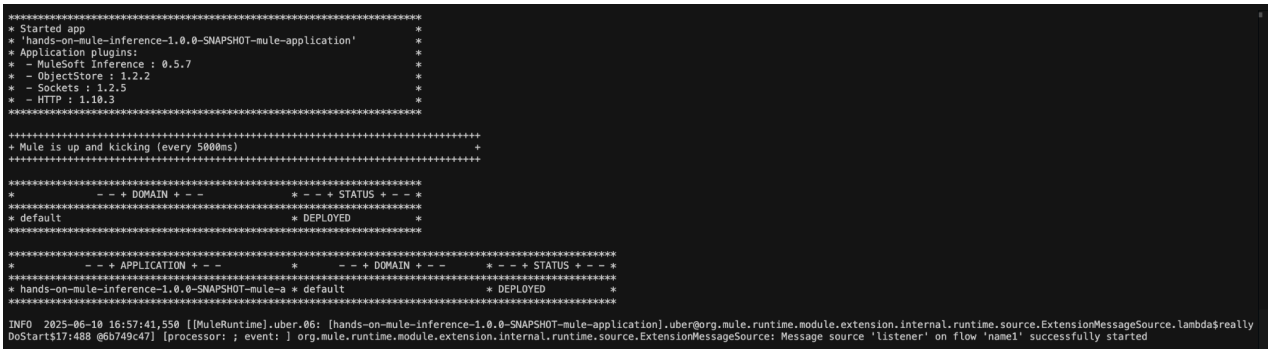
Update the [Chat] Completions operation with the following expression.

```
output application/json
---
vars.memory.chat_history ++ vars.userPrompt
```

IMAGE MISSING

Step 8

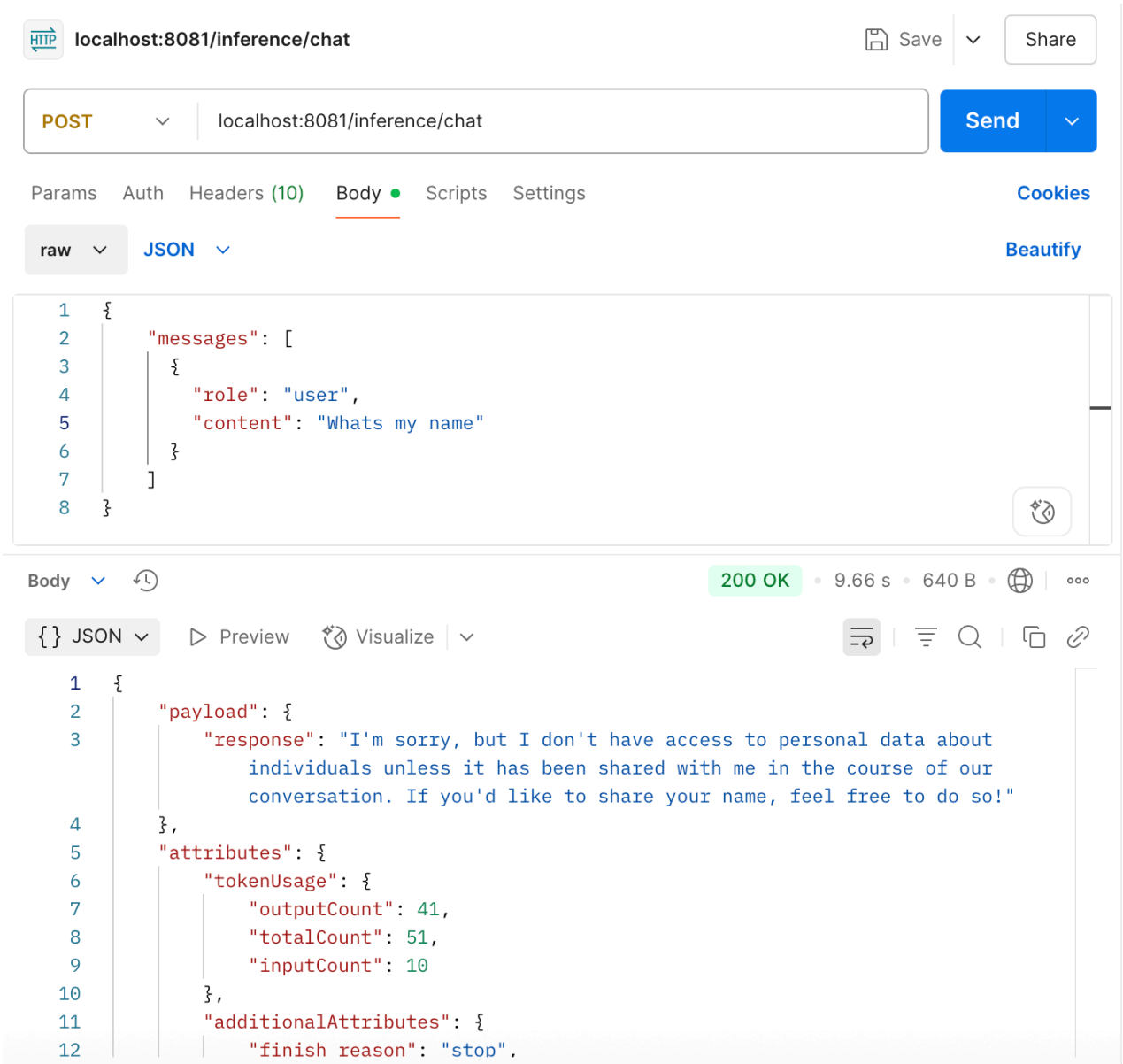
Start the mule app locally.



Step 9

Use postman or any other REST API Client to fire the following request.

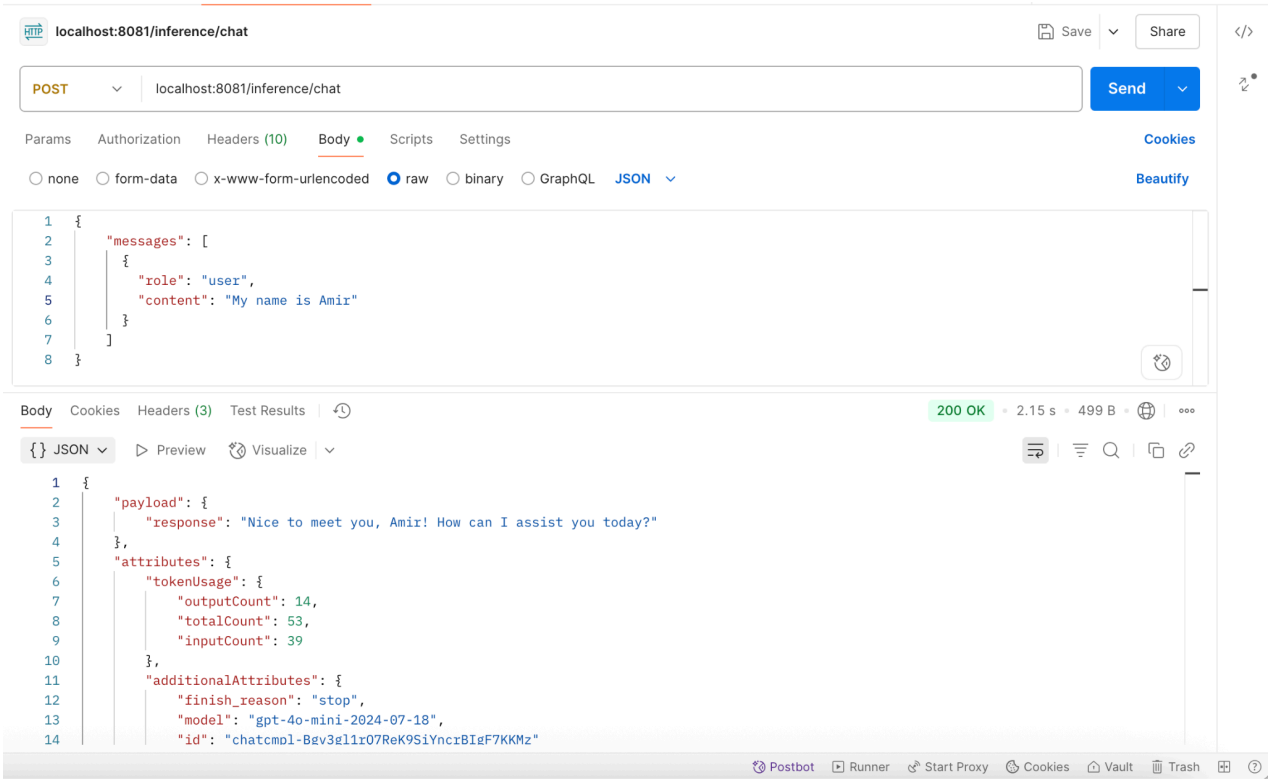
```
curl --location 'localhost:8081/inference/chat' \
--header 'userId: amir-khan' \
--header 'Content-Type: application/json' \
--data '{
"messages": [{
"role": "user",
"content": "Whats my name"
}]}'
```



Step 10

Use postman or any other REST API Client to fire the following request.

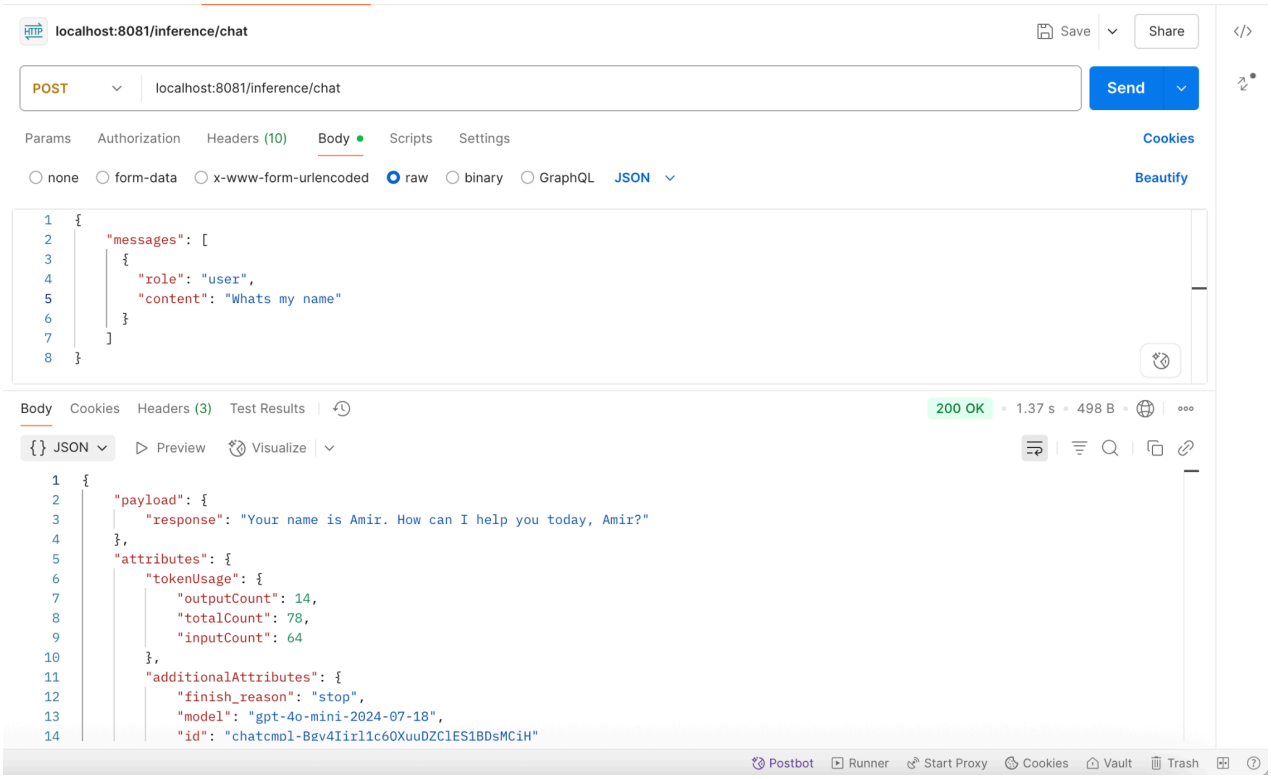
```
curl --location 'localhost:8081/inference/chat' \
--header 'userId: amir-khan' \
--header 'Content-Type: application/json' \
--data '{
"messages": [
{
"role": "user",
"content": "My name is Amir"
}
]
}'
```



Step 11

Use postman or any other REST API Client to fire the following request.

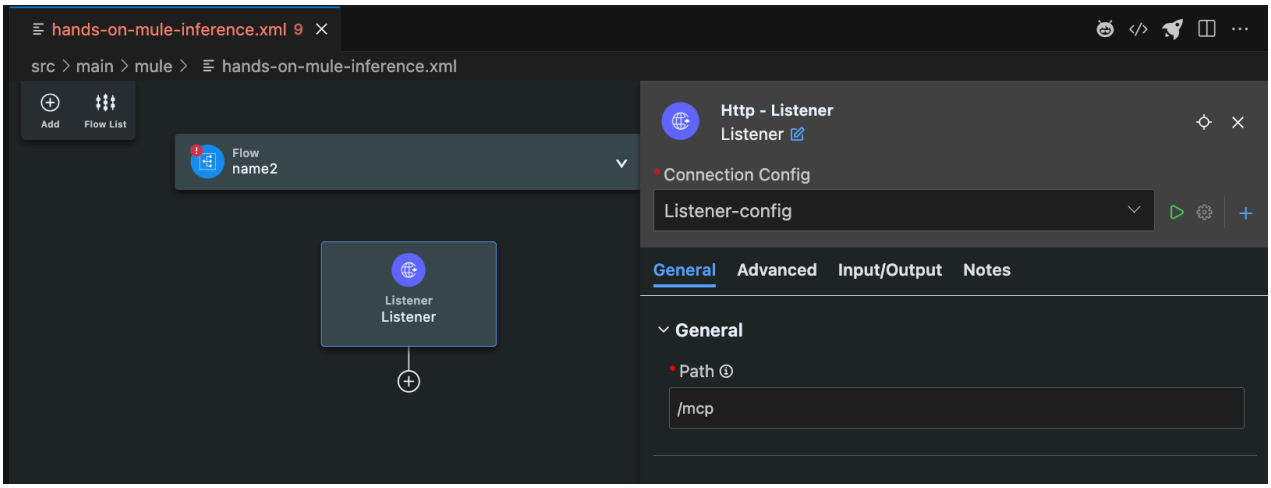
```
curl --location 'localhost:8081/inference/chat' \
--header 'userId: amir-khan' \
--header 'Content-Type: application/json' \
--data '{
"messages": [
{
"role": "user",
"content": "Whats my name"
}
]
}'
```



Exercise 3 - Add Native MCP Support

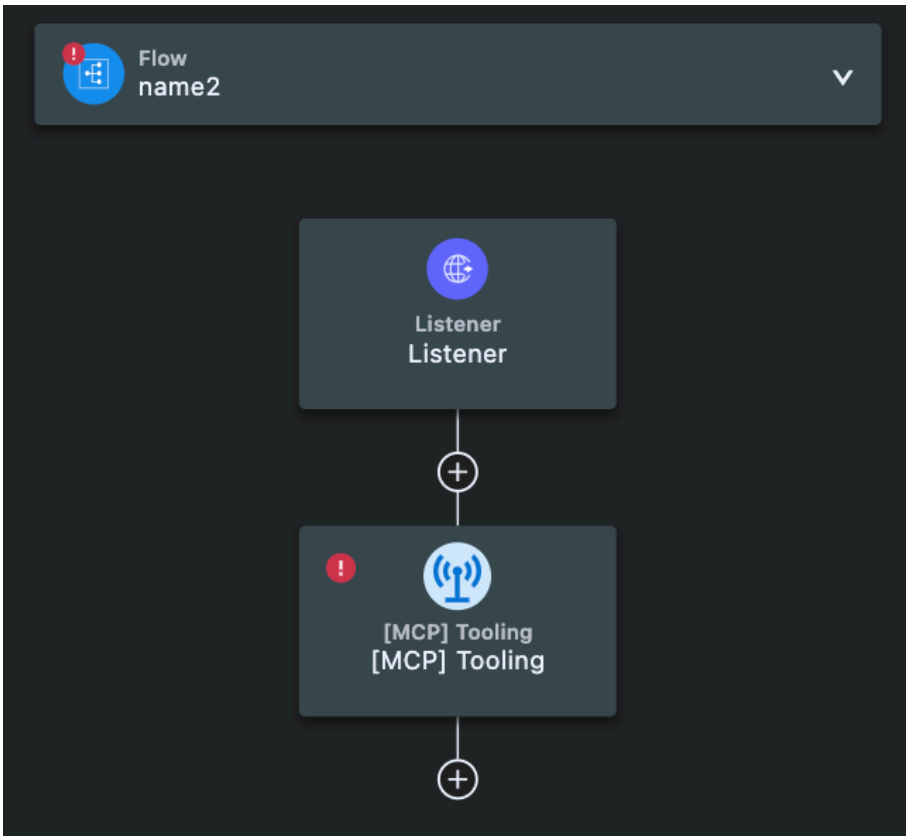
Step 1

Drop a http listener with (/inference/mcp) into a new flow.



Step 2

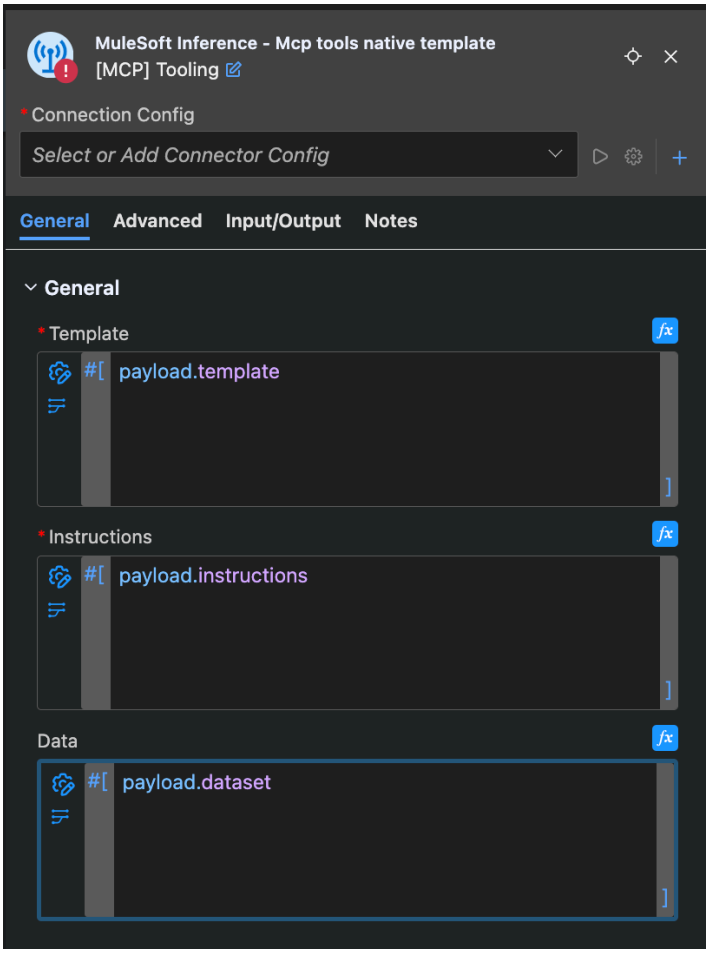
Add the [MCP] Tooling operations into the flow



Step 3

Add the following expression to the fields of the processor:

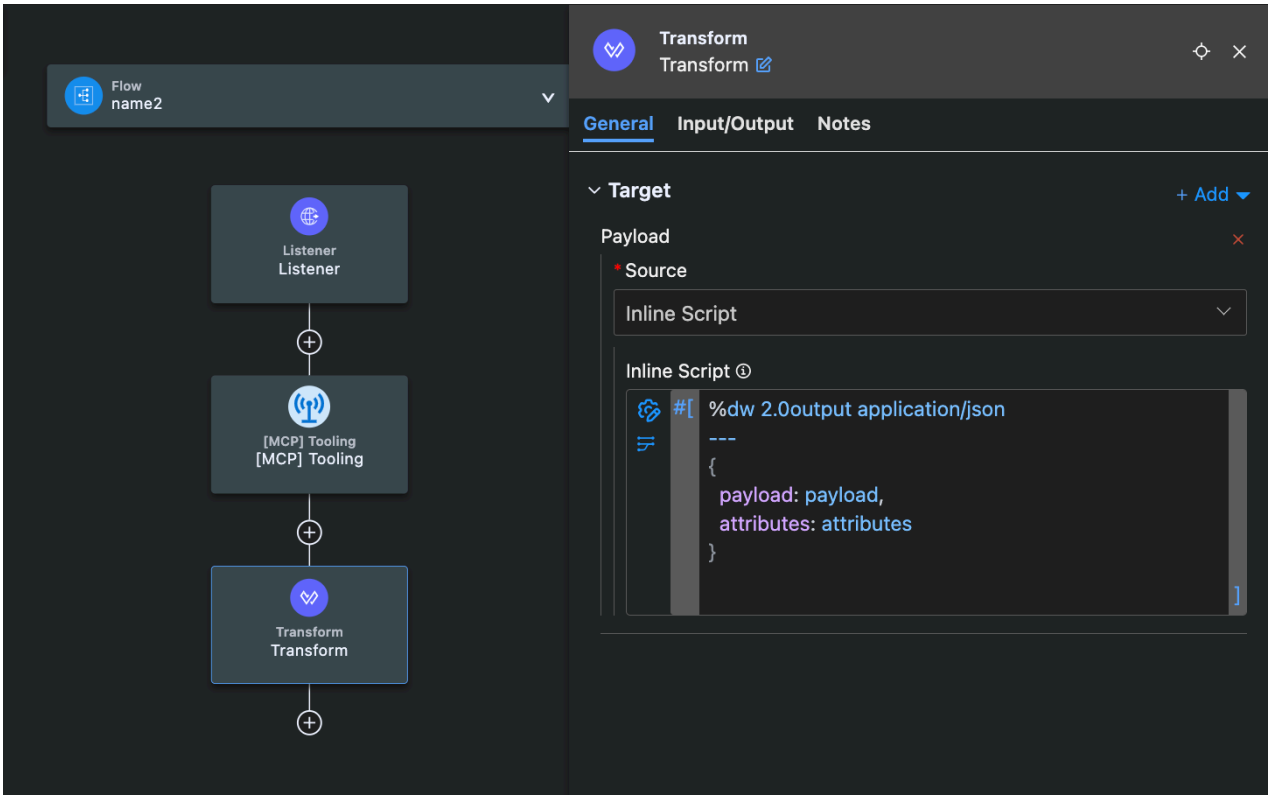
Template: `payload.template`
Instructions: `payload.instructions`
Dataset: `payload.dataset`



Step 4

Drop a **Transform Message** after the MCP Tooling operation and copy the following expression.

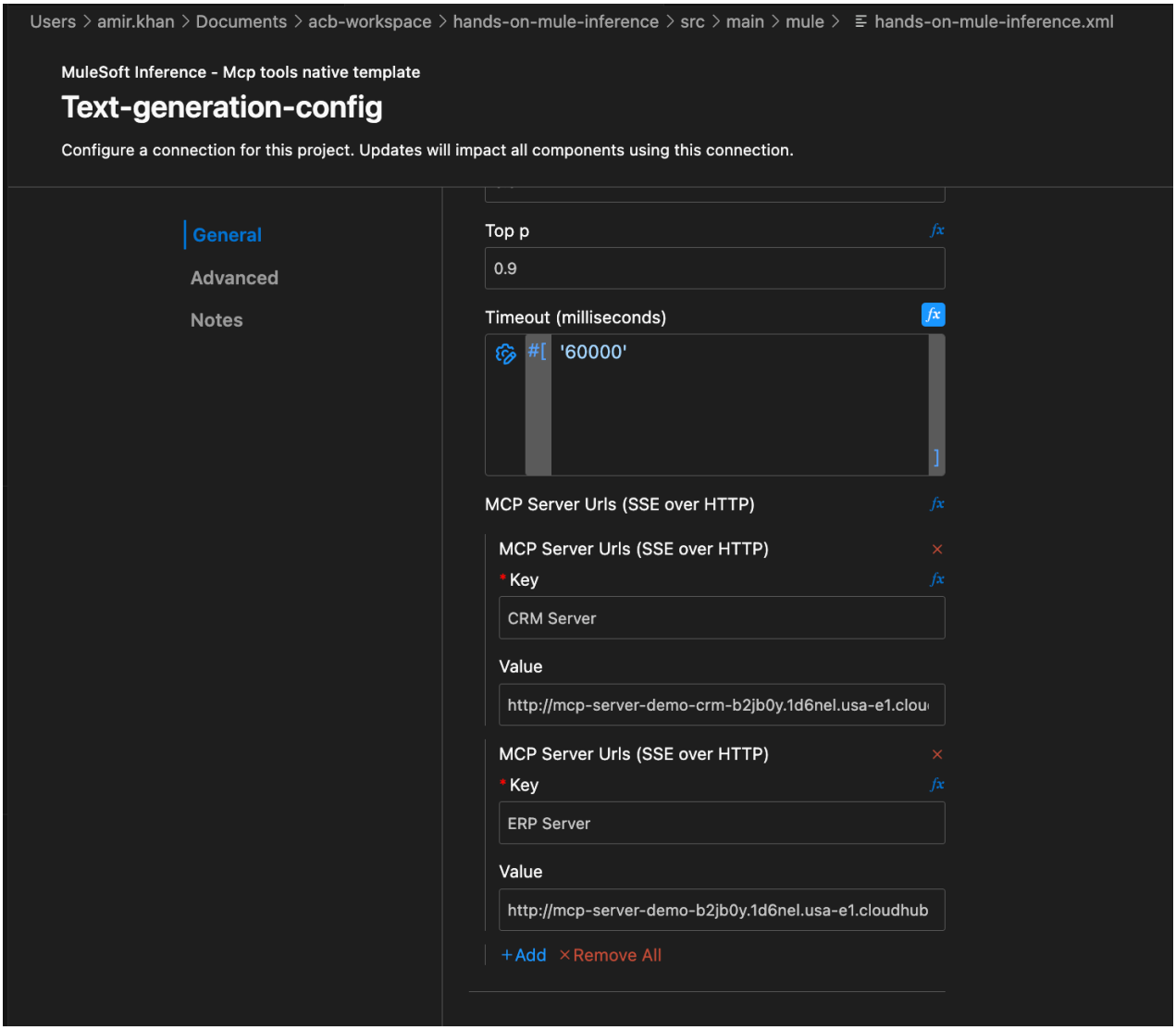
```
%dw 2.0
output application/json
---
{
payload: payload,
attributes: attributes
}
```



Step 5

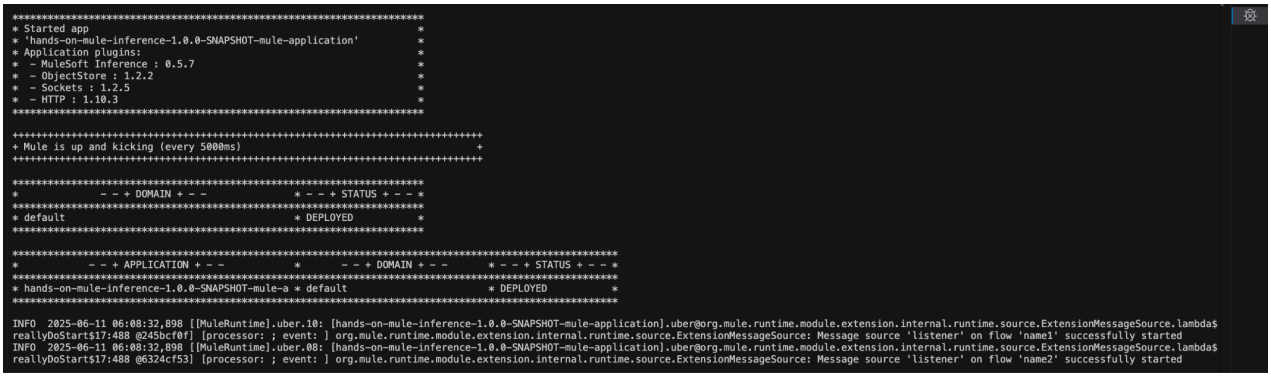
Now configure the MCP Servers provided above in the Inference Text Generation configuration.

ERP Server: http://mcp-server-demo-erp-ch1-application.us-e2.cloudhub.io
~~CRM Server: http://mcp-server-demo-crm-ch1-application.us-e2.cloudhub.io~~ DO NOT USE (you may face a timeout error)



Step 6

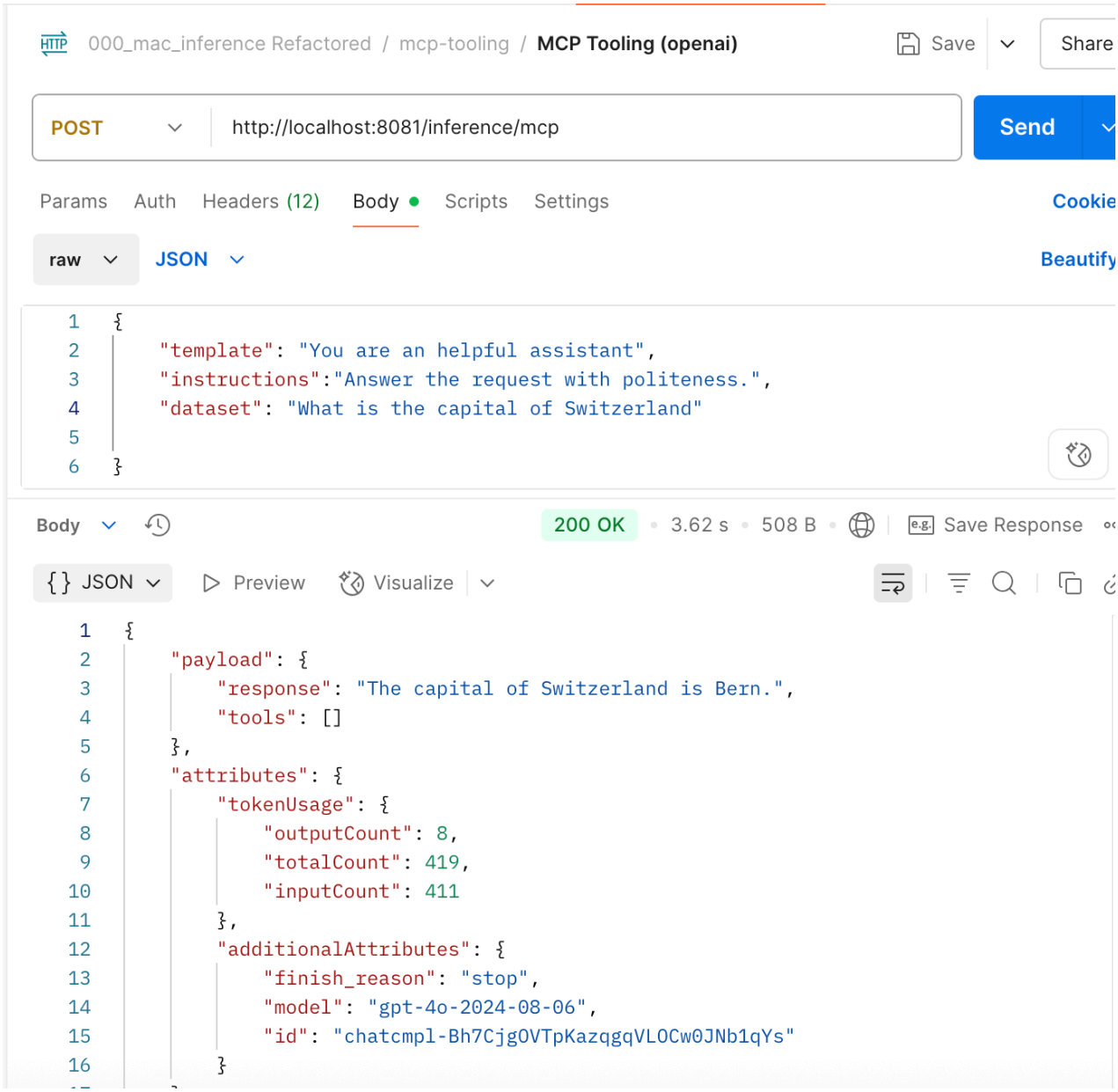
Deploy the application locally.



Step 7

Use Postman or any other API Client to perform the following request:

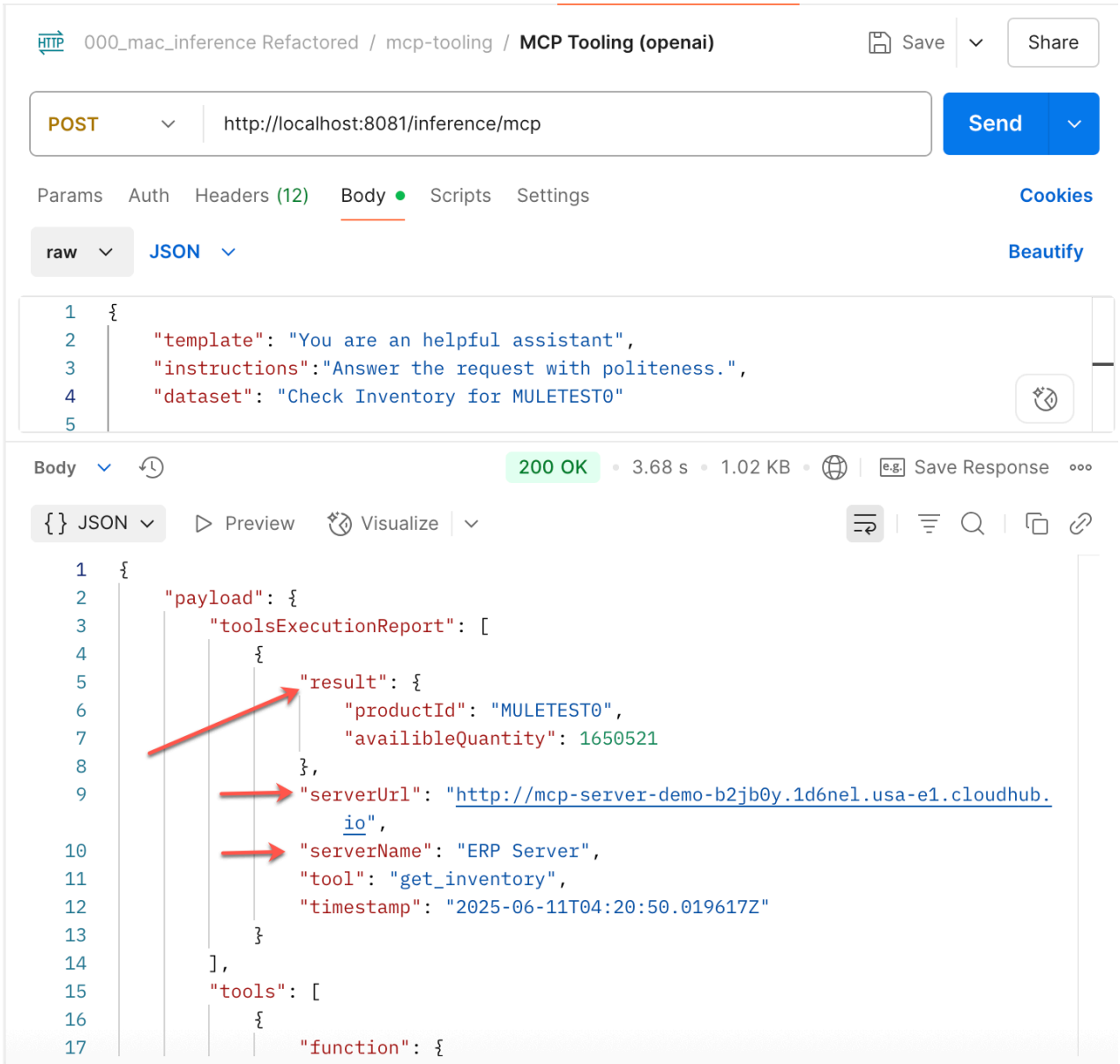
```
curl --location 'http://localhost:8081/inference/mcp' \
--header 'Content-Type: application/json' \
--data '{
"template": "You are an helpful assistant",
"instructions":"Answer the request with politeness.",
"dataset": "What is the capital of Switzerland"
}'
```



Step 8

Use Postman or any other API Client to perform the following request:

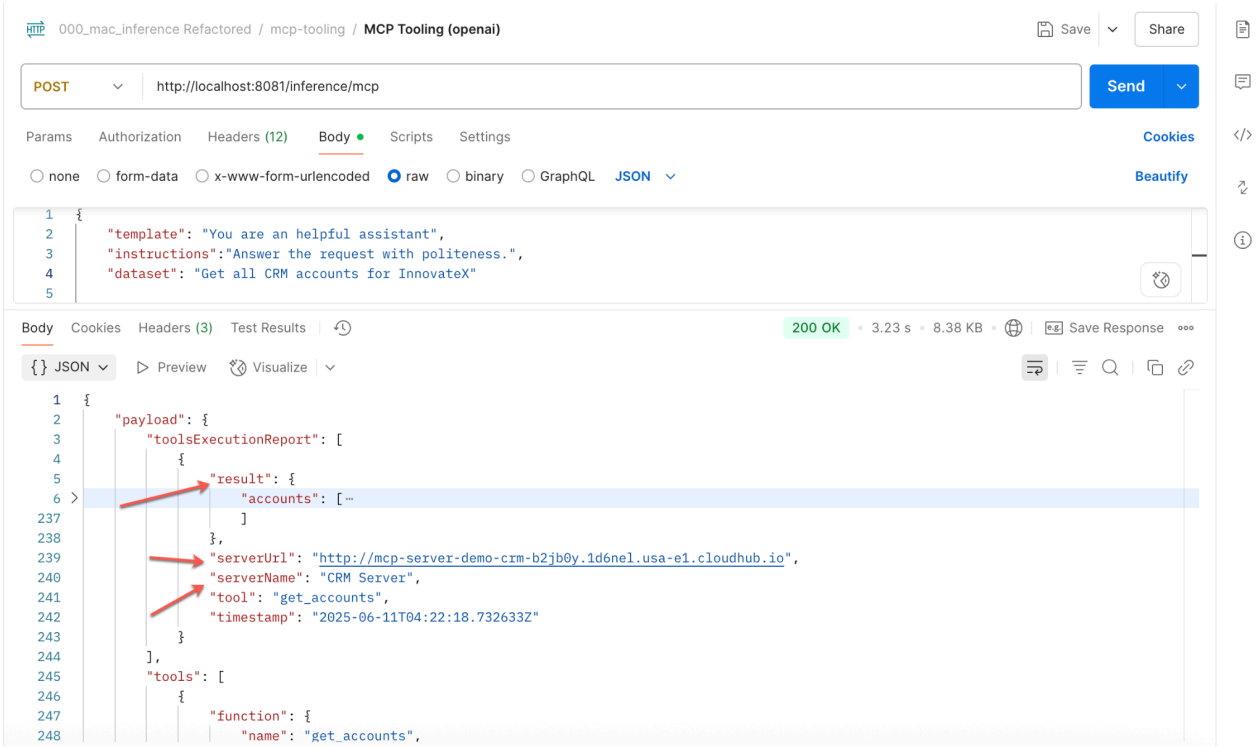
```
curl --location 'http://localhost:8081/inference/mcp' \
--header 'Content-Type: application/json' \
--data '{
"template": "You are an helpful assistant",
"instructions":"Answer the request with politeness.",
"dataset": "Check Inventory for MULETEST0"
}'
```



Step 9

Use Postman or any other API Client to perform the following request:

```
curl --location 'http://localhost:8081/inference/mcp' \
--data '{
"template": "You are an helpful assistant",
"instructions":"Answer the request with politeness.",
"dataset": "Get all CRM accounts for InnovateX"
}'
```

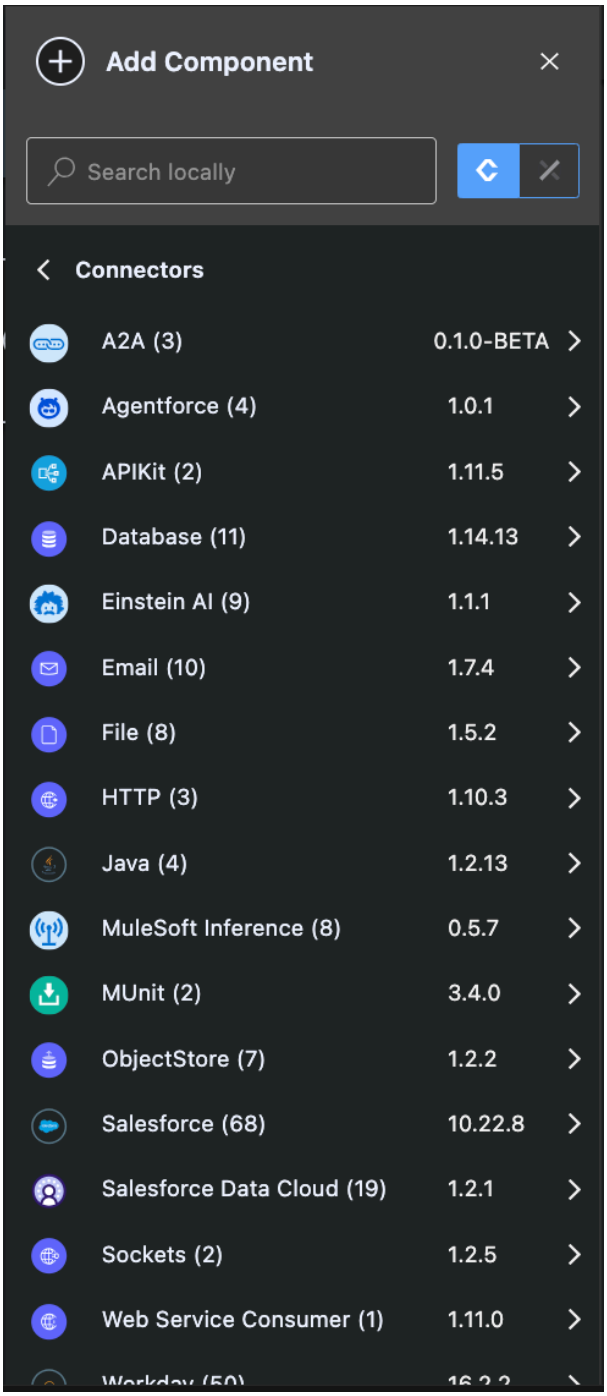


Exercise 4 - Implement A2A Standard with your Agent

A2A Connector 0.1.0-BETA can be added through Anypoint Exchange.

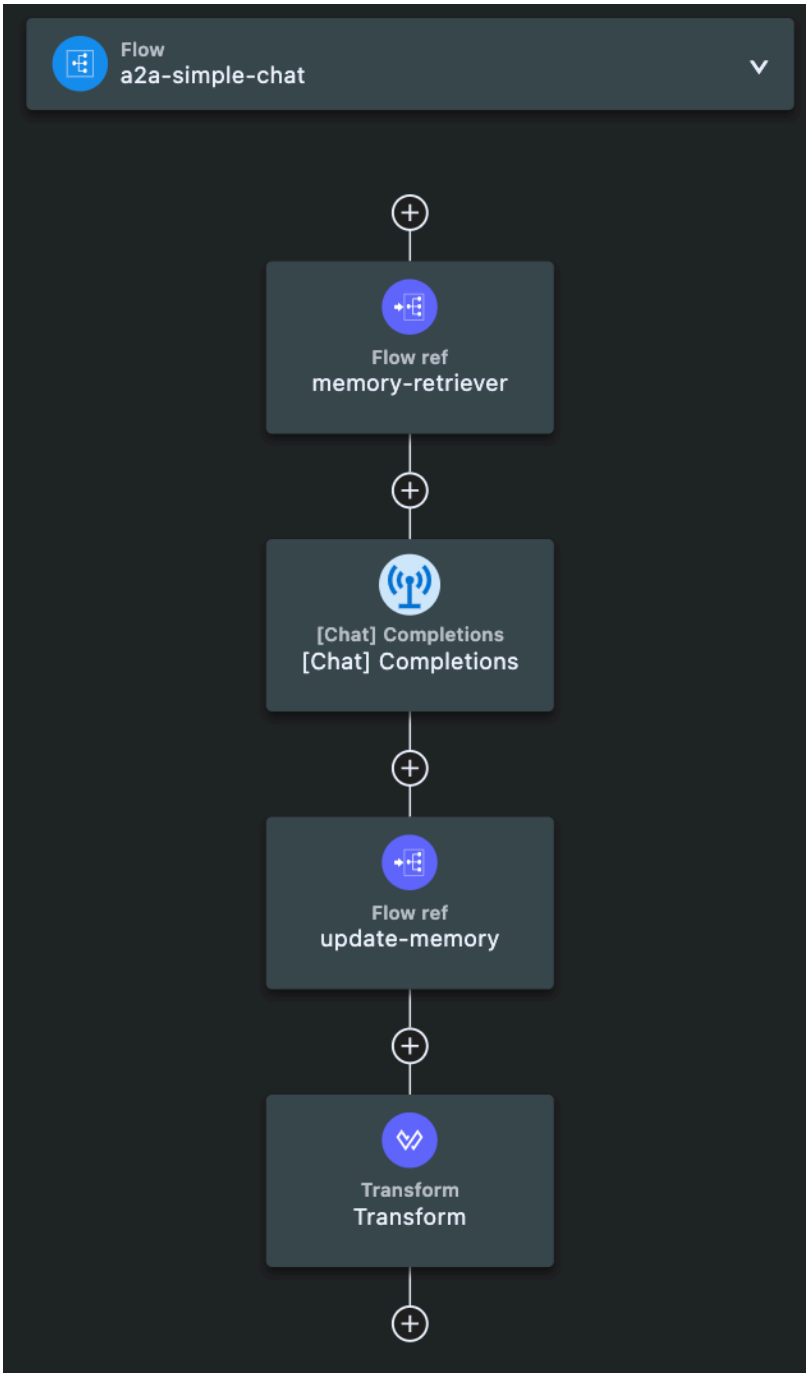
Step 1

Delete the Http Listener from the flow with the listener (/inference/chat) and rename it to a2a-simple-chat.



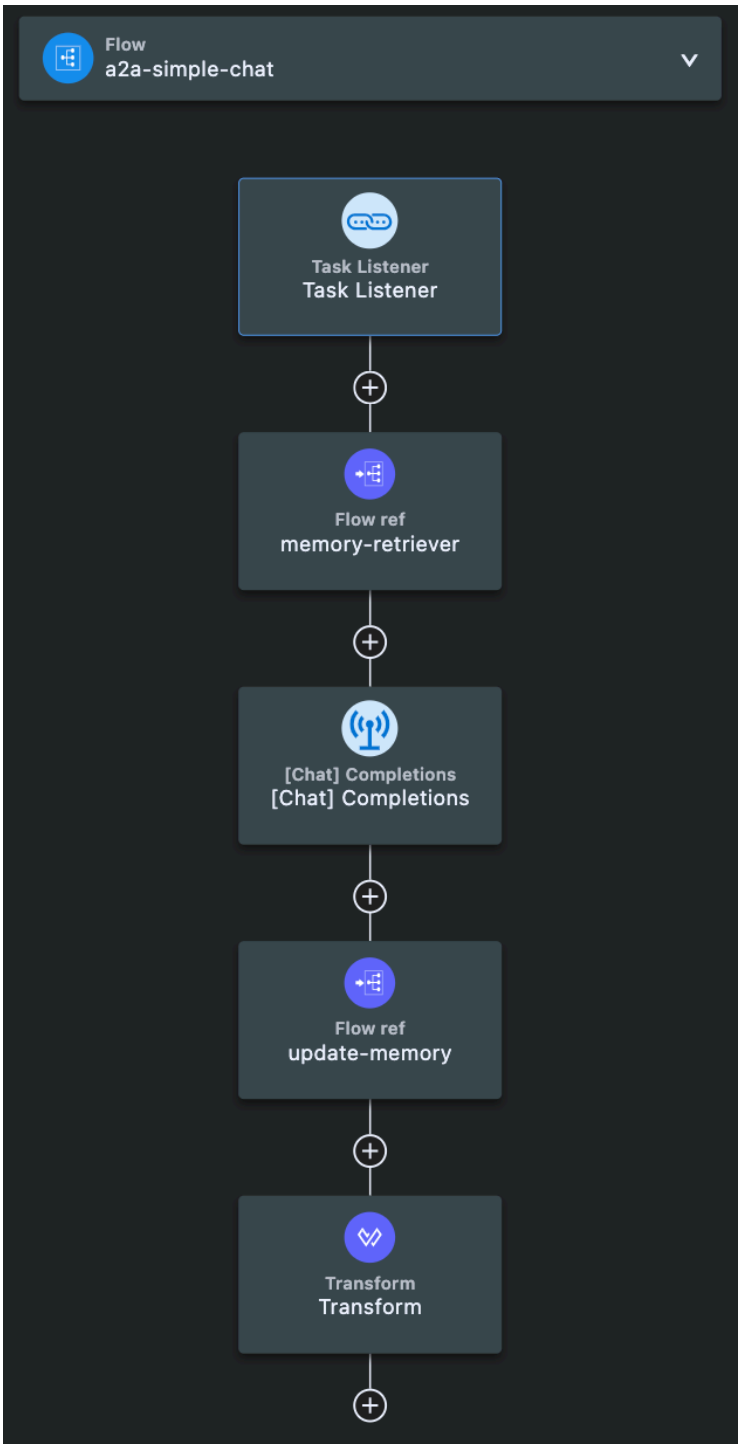
Step 2

Delete the Http Listener from the flow with the listener (/inference/chat) and rename it to a2a-simple-chat.



Step 3

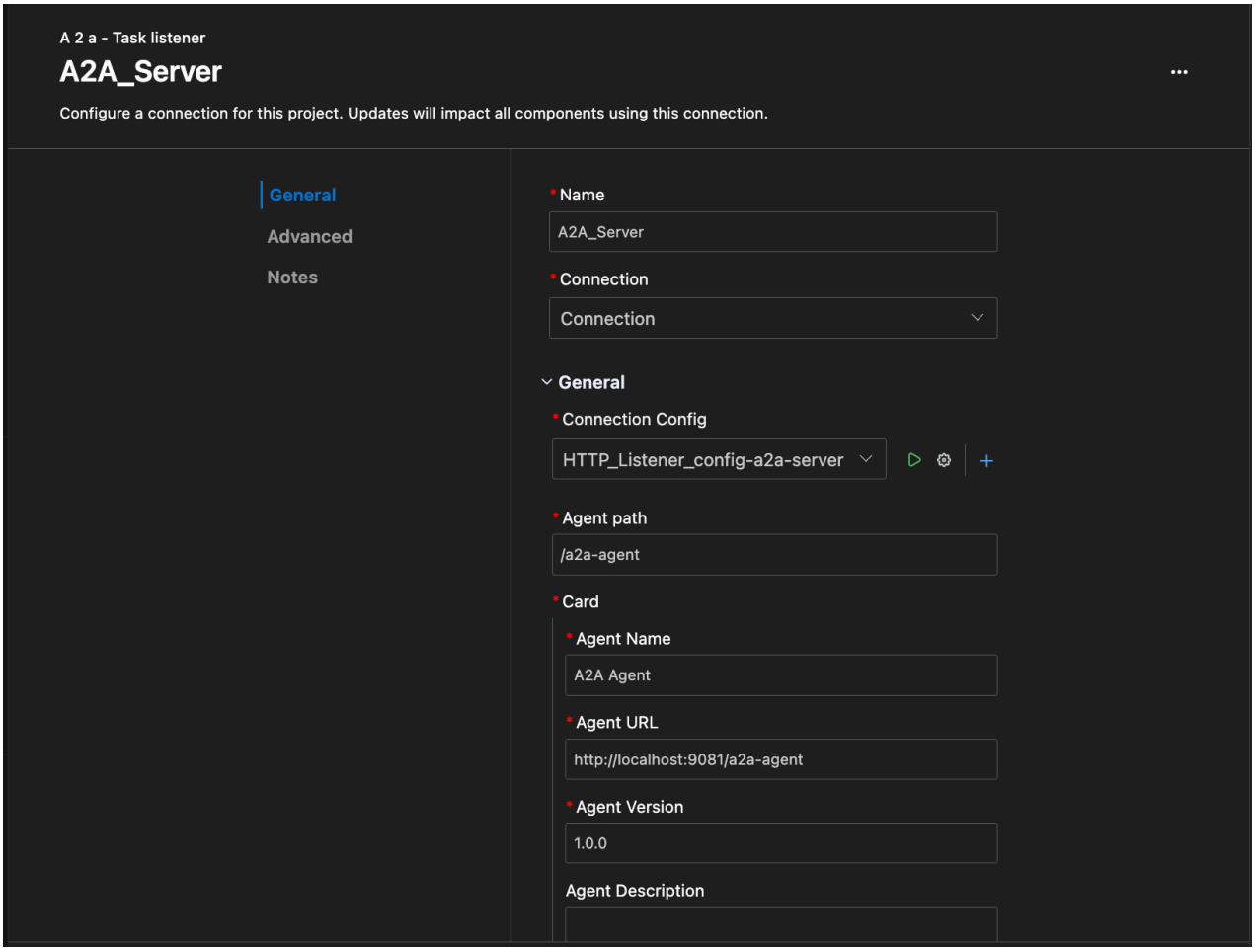
Add A2A Task Listener instead into the new flow as a source.



Step 4

Now add the following 2 configuration to your app.
Here we are adding a dedicated Http Listener for the A2A Task Listener and its Agent Configuration including Agent Card.

```
<http:listener-config name="HTTP_Listener_config-a2a-server"
doc:name="HTTP Listener config"
doc:id="133e35cd-f00a-4e7e-8e9f-cb125a4afd2a" >
<http:listener-connection host="0.0.0.0" port="9081" />
</http:listener-config>
<a2a:server-config name="A2A_Server" doc:name="A2A Server"
doc:id="ac631edc-6db4-4f8f-8ea1-868d8badaedd" >
<a2a:connection listenerConfig="HTTP_Listener_config-a2a-server"
agentPath="/a2a-agent" />
<a2a:card name="A2A Agent" url="http://localhost:9081/a2a-agent"
version="1.0.0" >
<a2a:description ><![CDATA[This is a simple chat
agent]]></a2a:description>
<a2a:skills >
<a2a:agent-skill id="1" name="ChatMemory" >
<a2a:description ><![CDATA[This agent has the ability to remember
conversation]]></a2a:description>
</a2a:agent-skill>
</a2a:skills>
</a2a:card>
</a2a:server-config>
```


A 2 a - Task listener
A2A_Server
Configure a connection for this project. Updates will impact all components using this connection.
General
Advanced
Notes
* Name
A2A_Server
* Connection
Connection
General
* Connection Config
HTTP_Listener_config-a2a-server
* Agent path
/a2a-agent
* Card
* Agent Name
A2A Agent
* Agent URL
http://localhost:9081/a2a-agent
* Agent Version
1.0.0
Agent Description

Step 5

Update the memory-retriever sub-flow with the following expression as replacement.

We are not anymore looking for the payload.messages for the userPrompt, but instead getting the userPrompt from an standarad a2a structured payload and addiitonally also setting the task Id, as well as the session Id as variables.

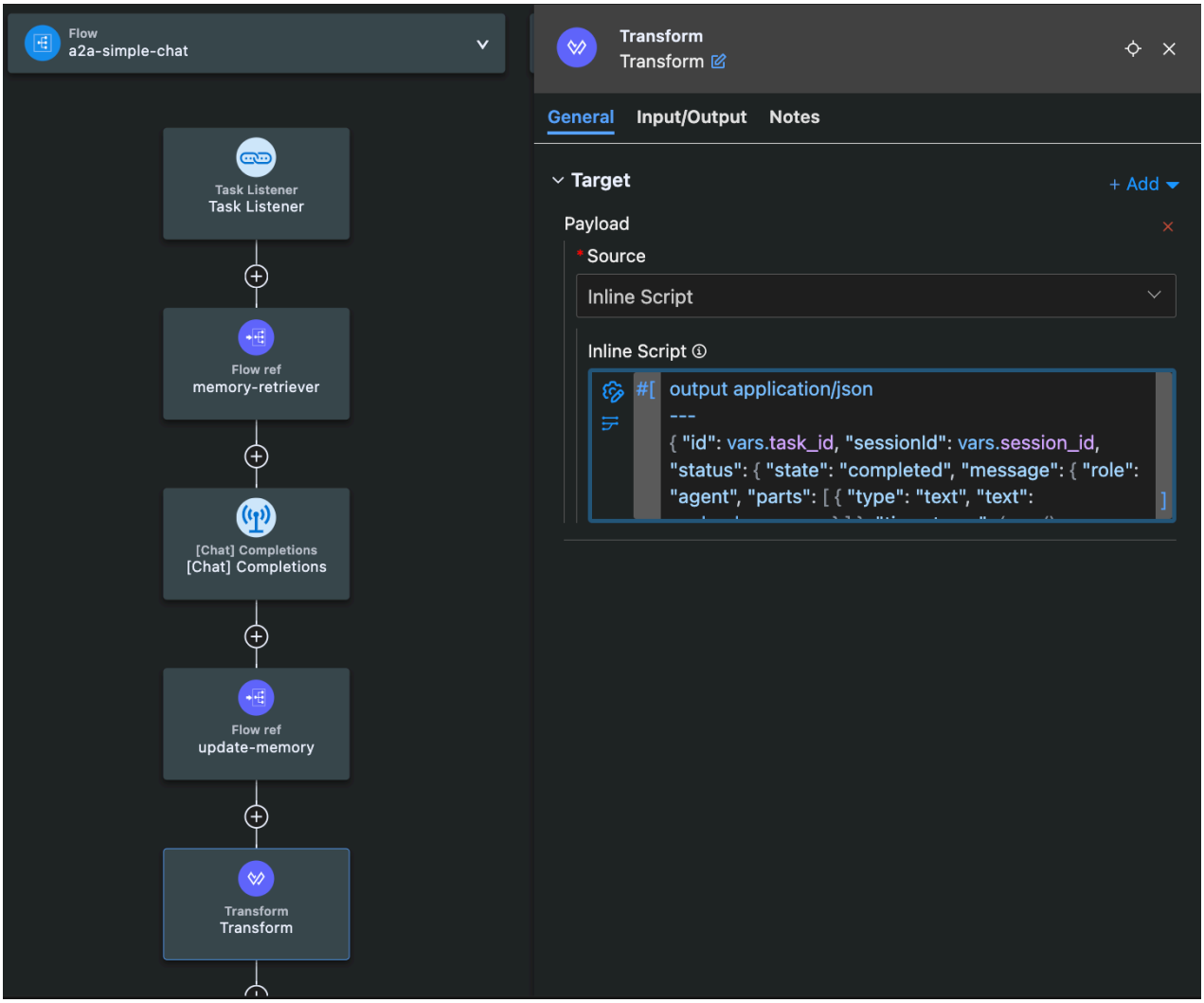
```
<sub-flow name="memory-retriever"
doc:id="72bf2565-f2b4-4533-ba5e-14f385b6b1c1" >
<set-variable value="#[payload.message.parts[0].text default
payload.messages]" doc:name="Set task user prompt"
doc:id="c15af2c5-dd80-4d99-b4e5-9df0aee6898b" variableName="userPrompt"
/>
<set-variable value="#[payload.id]" doc:name="Set Task Id"
doc:id="c4602681-f811-4d86-8088-56ad30b2b4bf" variableName="task_id" />
<set-variable value='#[(payload.sessionId) default ""]' doc:name="Set
Session Id" doc:id="5d513a8f-e069-4edf-a3c8-1fd88b0c1373"
variableName="session_id" />
<set-variable value='#[attributes.headers."userId" default
"amir-khan"]' doc:name="User Id"
doc:id="28824951-7f99-4313-b74f-50fd66fbd94b" variableName="userId"/>
<os:retrieve doc:name="Load Memory"
doc:id="4e5a4fb2-ca70-44dc-84f7-f5de36676a44" key="#[vars.userId]"
objectStore="Object_store" target="memory">
<os:default-value ><![CDATA[#[output application/json
---
{
chat_history: []
}]]]></os:default-value>
</os:retrieve>
</sub-flow>
```


Sub flow
memory-retriever
var
Set variable
Set task user prom...
var
Set variable
Set Task Id
var
Set variable
Set Session Id
var
Set variable
User Id
Retrieve
Load Memory

Step 6

The Transform Message processor needs to be updated with the following dataweave expression as a replacement. This is the A2A response format to comply to the A2A protocol.
We are setting 3 A2A specific attributes with taskId, sessionId, and the text in the parts as the response.

```
%dw 2.0
output application/json
---
{
"id": vars.task_id,
"sessionId": vars.session_id,
"status": {
"state": "completed",
"message": {
"role": "agent",
"parts": [
{
"type": "text",
"text": payload.response
}
]
},
"timestamp": (now() >> 'UTC') as String {format:
"yyyy-MM-dd'T'HH:mm:ss'Z'"}
},
"artifacts": [
{
"name": "Answer",
"index": 0,
"parts": [
{
"type": "text",
"text": payload.response
}
]
}
]
}
```

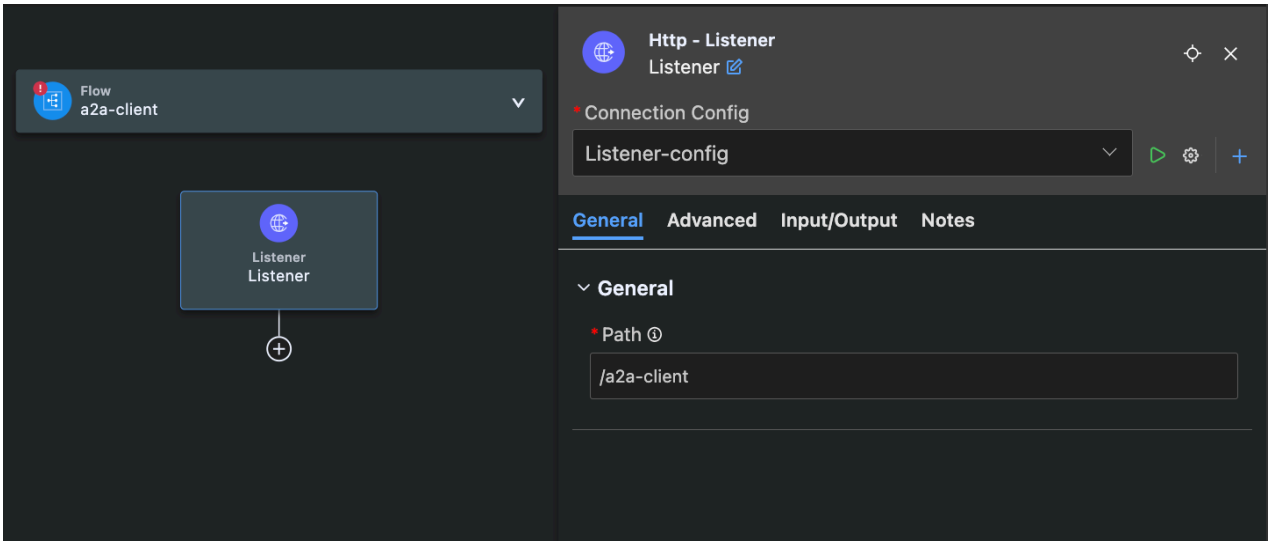

Flow
a2a-simple-chat
Task Listener
Task Listener
Flow ref
memory-retriever
[Chat] Completions
[Chat] Completions
Flow ref
update-memory
Transform
Transform
Transform
Transform
General
Input/Output
Notes
Target
+ Add
Payload
Source
Inline Script
Inline Script
#[output application/json

{ "id": vars.task_id, "sessionId": vars.session_id,
"status": { "state": "completed", "message": { "role":
"agent", "parts": [{ "type": "text", "text":
]

Step 7

Now our A2A Server Agent is ready. We need to add now a A2A client call to the same app in order to communicate with the A2A server agent. We drop a new http listener (/inference/a2a-client) in to a new flow (a2a-client).

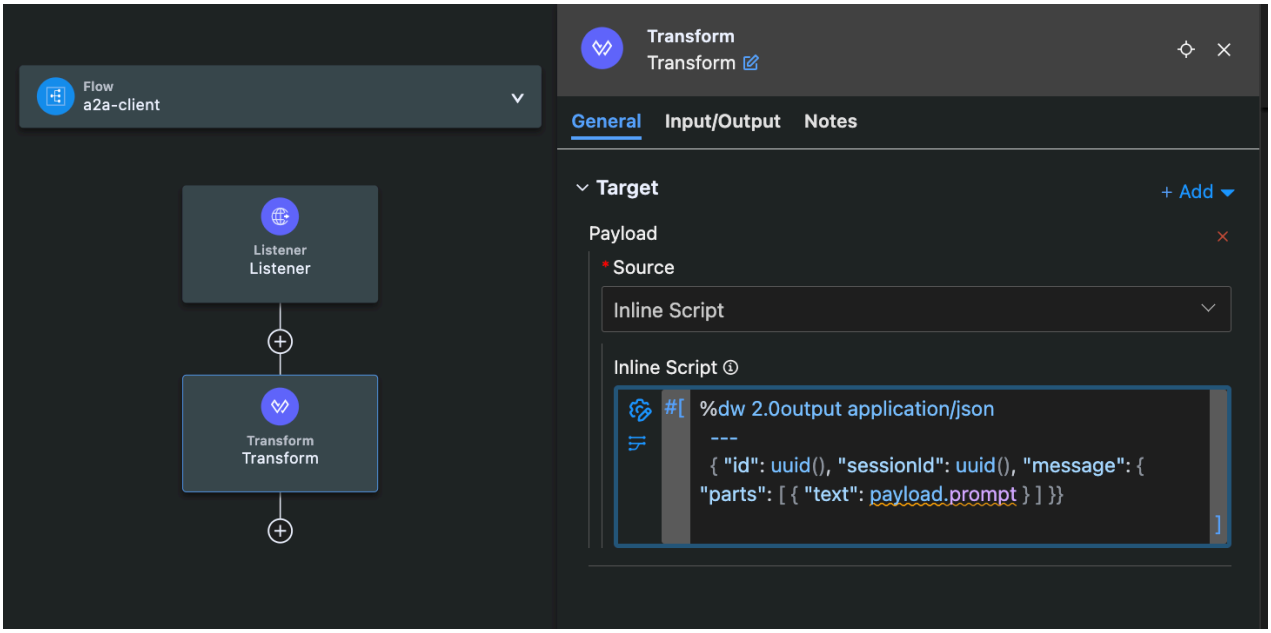
Make sure to reuse the http configuration running on 8081 and not the a2a server http configuration.



Step 8

Add a Transform Message into the a2a-client flow with the following dataweave expression. This is the standard way of doing a client call to an A2A server. Key attributes as TaskId, SessionId and message must be provided.

```
%dw 2.0
output application/json
---
{
"id": uuid(),
"sessionId": uuid(),
"message": {
"parts": [
{
"text": payload.prompt
}
]
}
}
```

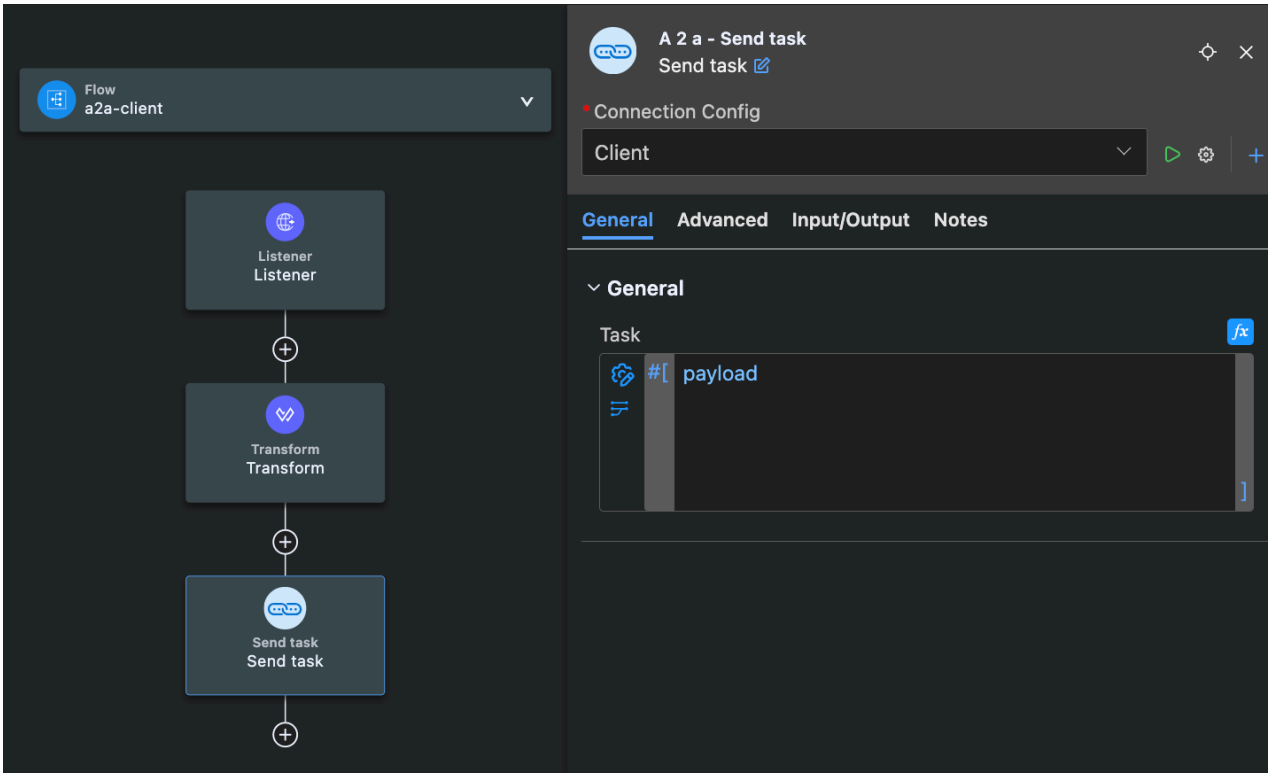

Flow
a2a-client
Listener
Listener
Transform
Transform
Transform
Transform
General
Input/Output
Notes
Target
+ Add
Payload
*Source
Inline Script
Inline Script
#[%dw 2.0output application/json

{ "id": uuid(), "sessionId": uuid(), "message": {
"parts": [{ "text": payload.prompt }] }}
]

Step 9

Finally add a Send Task processor from the A2A Connector after the Transform Message operation.

In the configuration add the following agent url:
`http://localhost:9081/a2a-agent`



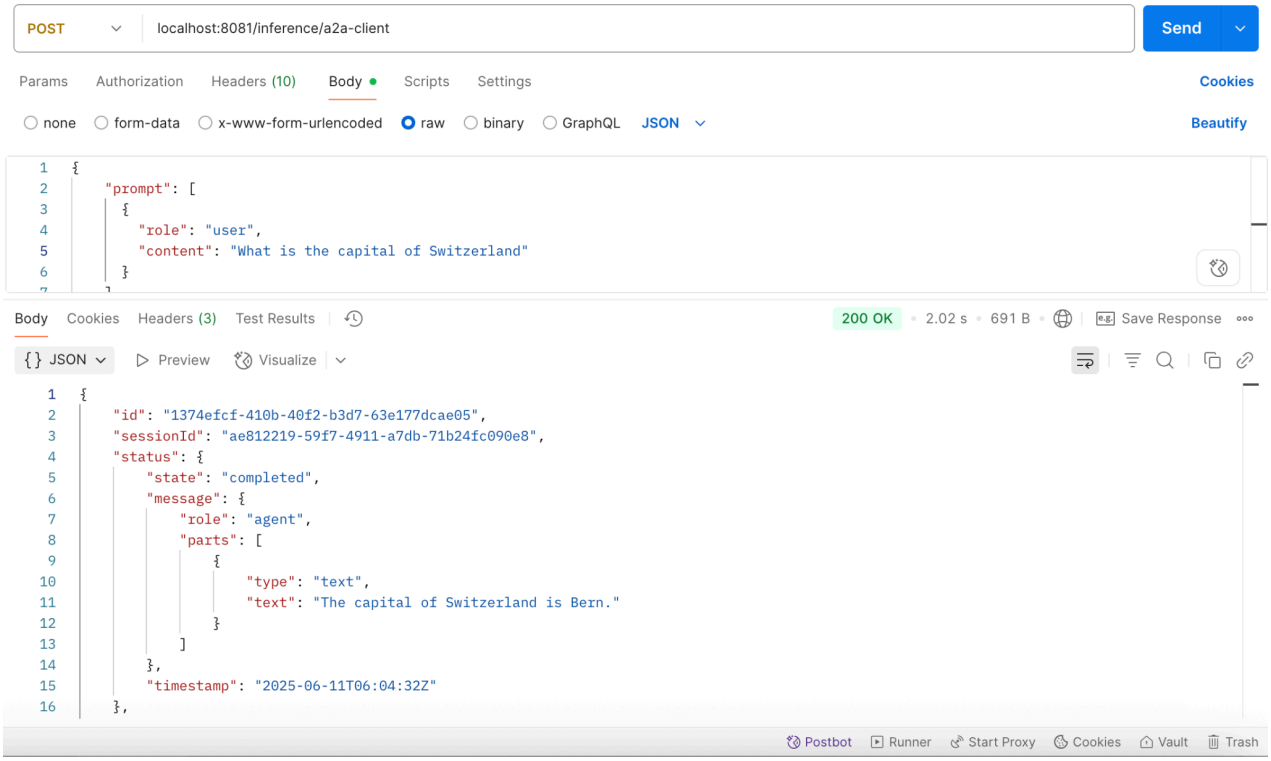
Step 10

Deploy the application locally

Step 11

Use postman or any other API Client to execute the following request:

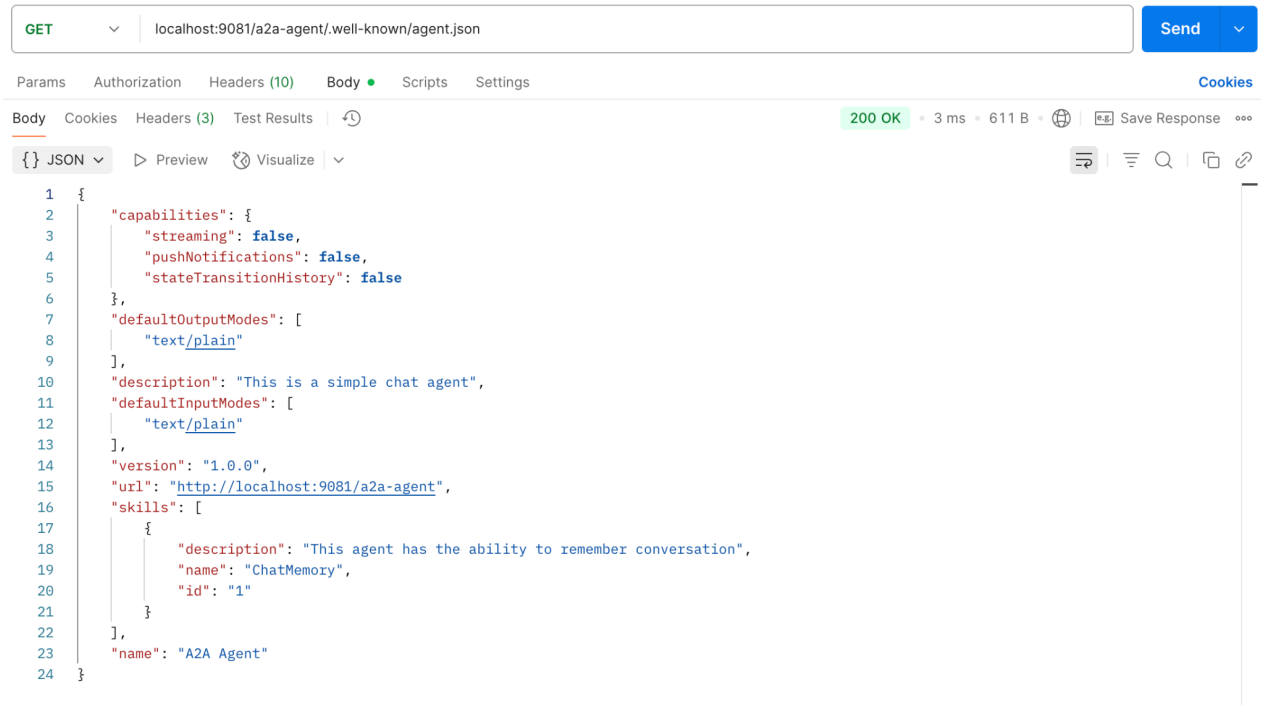
```
curl --location 'localhost:8081/inference/a2a-client' \
--header 'userId: amir-khan' \
--header 'Content-Type: application/json' \
--data '{
"prompt": [
{
"role": "user",
"content": "What is the capital of Switzerland"
}
]
}'
```



Step 12

Get the **agent card** by using the following request.

```
curl --location --request GET
'localhost:9081/a2a-agent/.well-known/agent.json' \
--header 'Content-Type: application/json' \
```



Vectors Hands-on

Vectors Connector Hands-on

Prerequisites

- Setup PostgresSQL from SE Platform

ACB (Vectors)

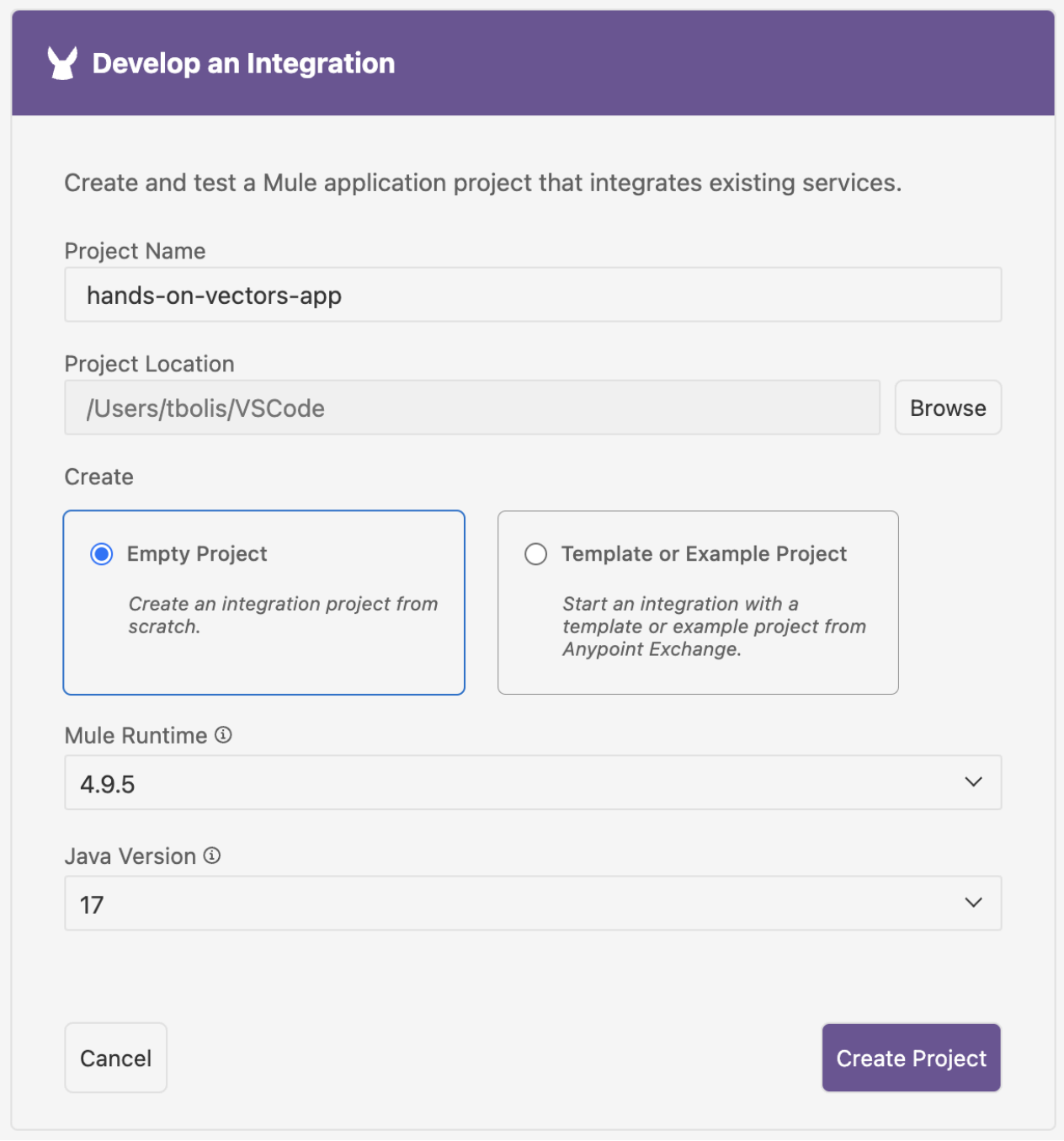
```
#heroku-hands-on configurations
herokuhandson:
  host: cloud-services.demos.mulesoft.com
  port: '30389'
  username: postgres
  databaseName: sampledb
  password: TVK4pah_tze3dqe5qvx
  rootPassword: ''
```

ACB (Vectors)

Exercise 1- Data Load

Step 1

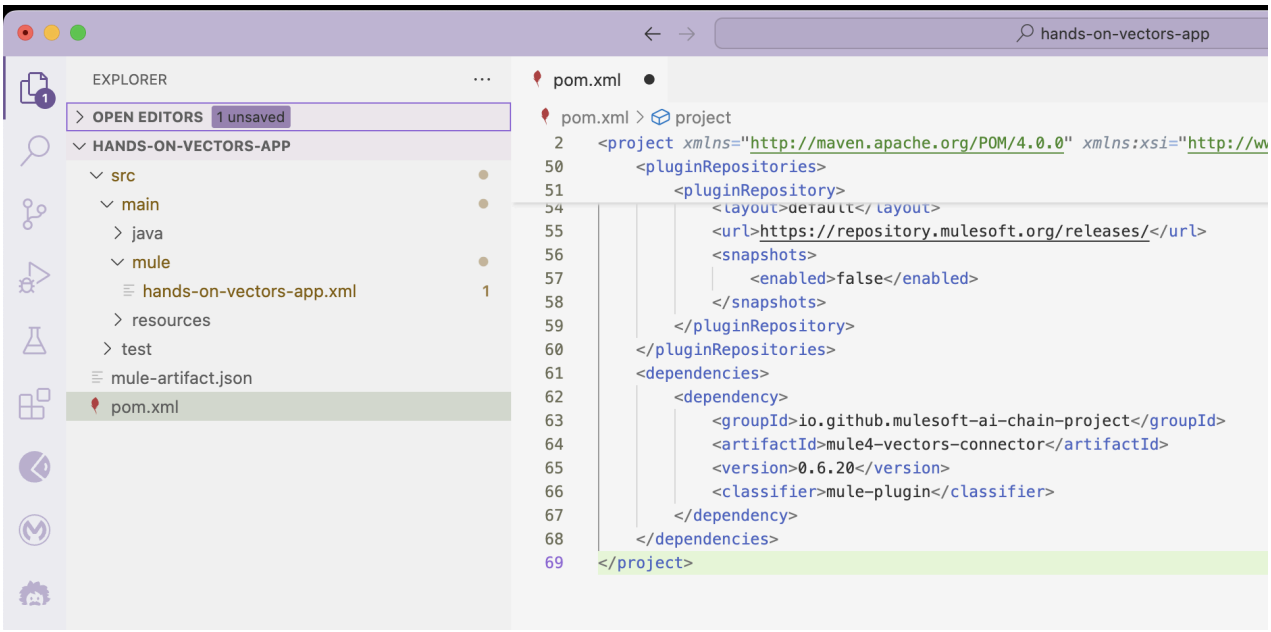
Create a new mule app `hands-on-vectors-app` using Studio or ACB.



Step 2

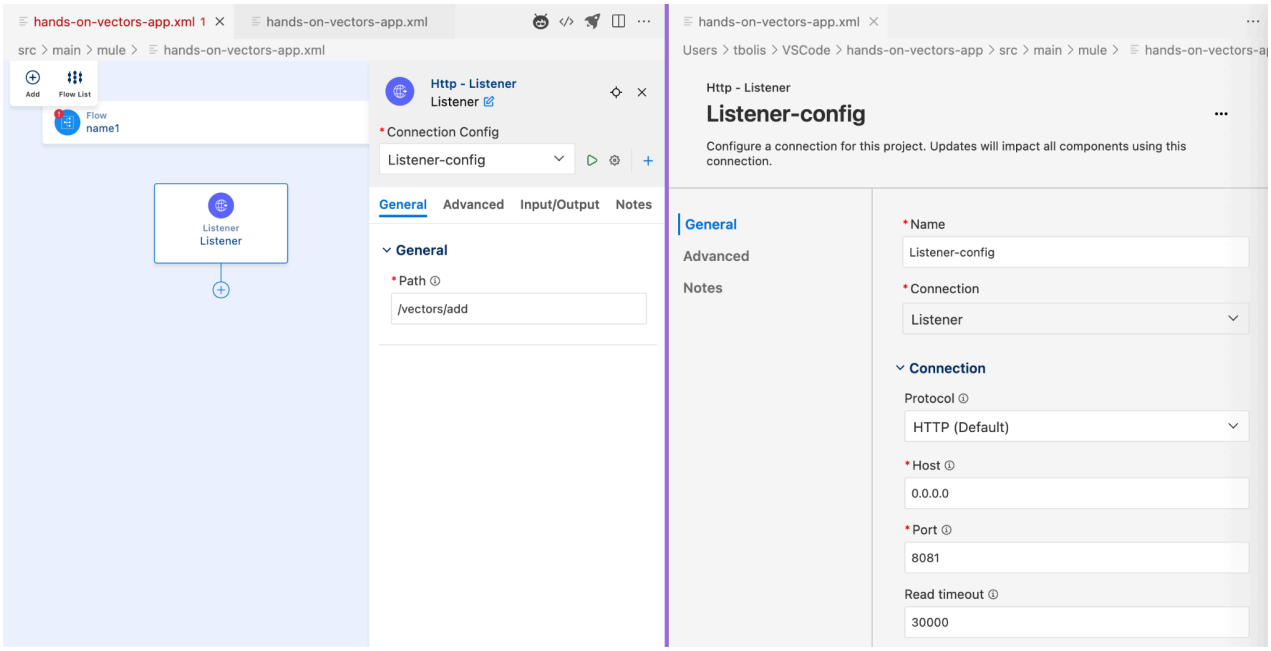
Add Vectors connector dependency to your pom.xml

```
...
<dependencies>
...
<dependency>
<groupId>io.github.mulesoft-ai-chain-project</groupId>
<artifactId>mule4-vectors-connector</artifactId>
<version>0.6.20</version>
<classifier>mule-plugin</classifier>
</dependency>
...
</dependencies>
...
```



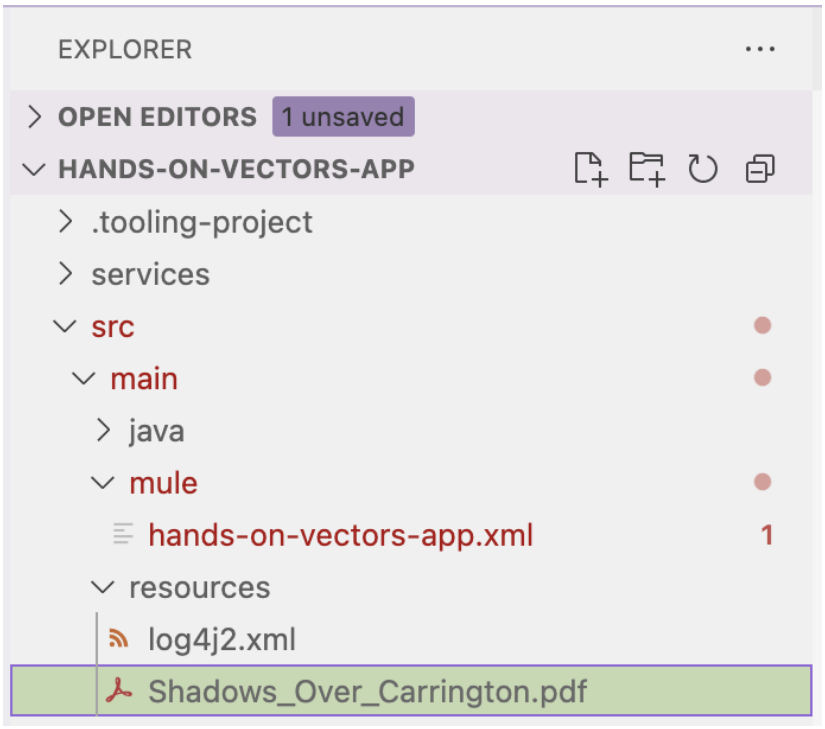
Step 3

Drop http listener (`/vectors/add`) in a new flow, you'll use it to load data into store using postman



Step 4

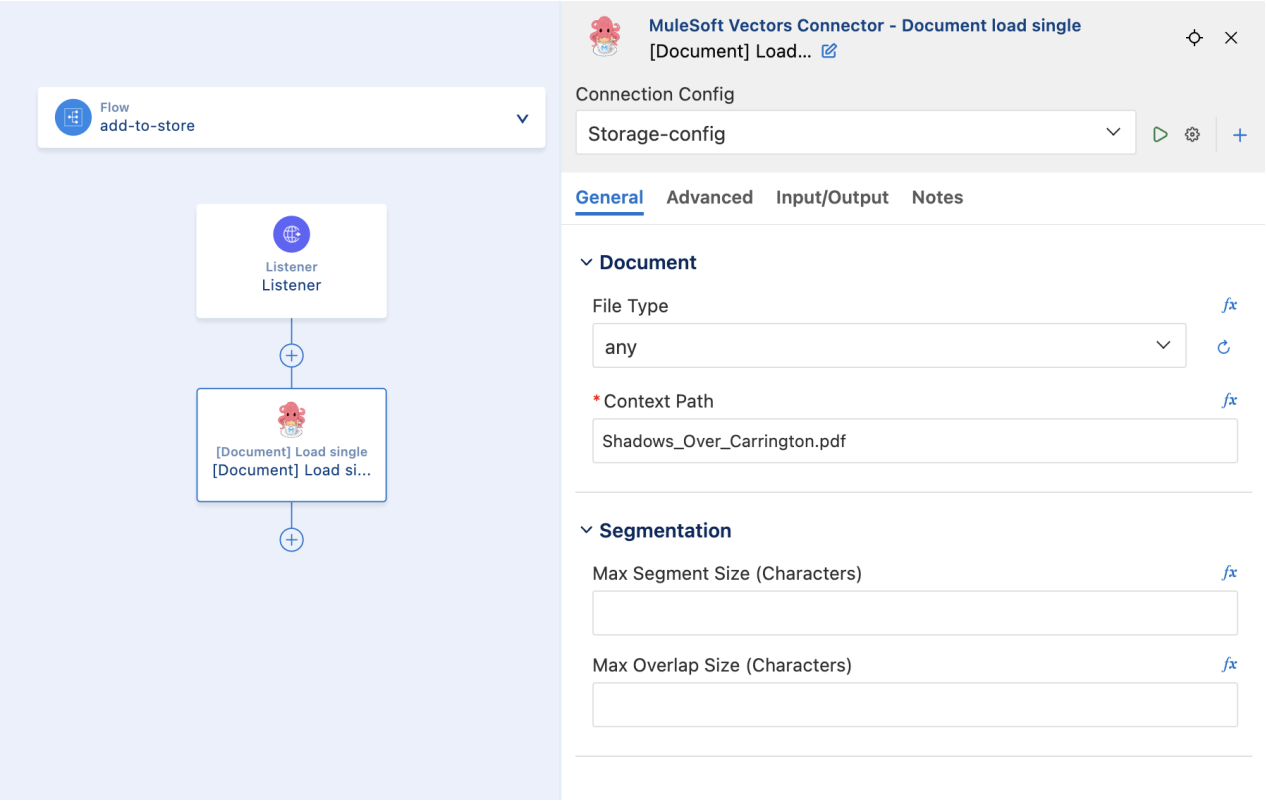
Save the test file under your src/main/resources folder.



Step 5

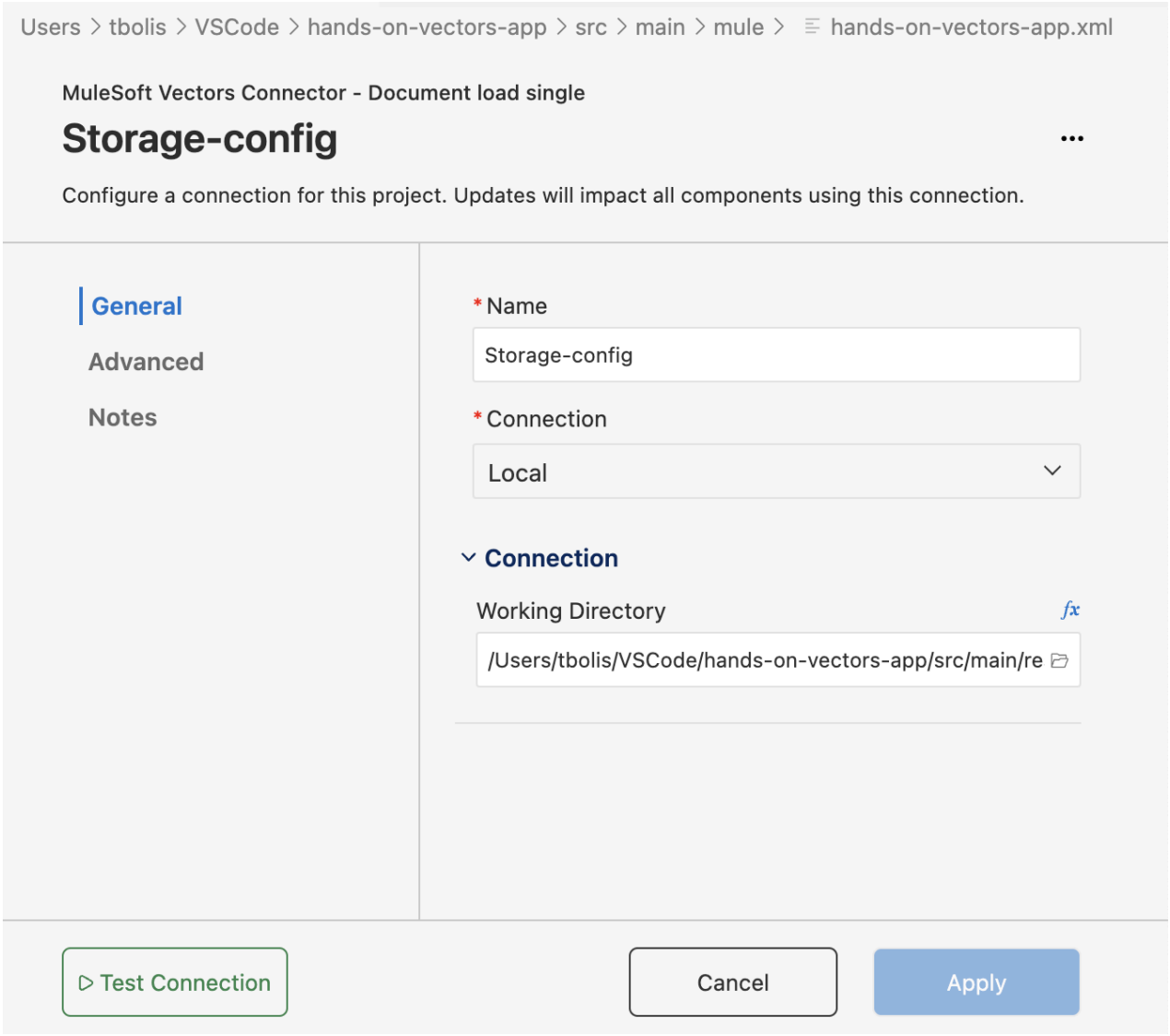
Drop [Document] Load single operation after listener. Leave segmentation parameters empty.

Set:
- File Type to `any`
- Context Path to `Shadows_Over_Carrington.pdf`



Step 6

Create a Storage Configuration (with Local Connection). Set <`project-path>/src/main/resources` as working dir.



Step 7

Add Shared Libraries.

- ACB: directly edit the pom.xml

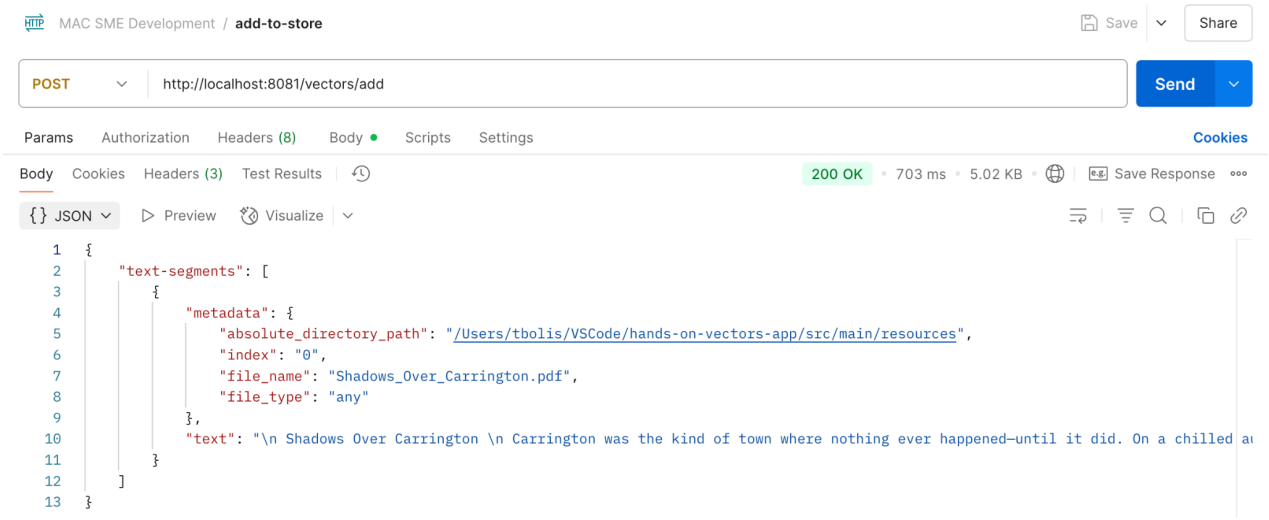
```
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-clean-plugin</artifactId>
<version>3.1.0</version>
</plugin>
```

```
<plugin>
<groupId>org.mule.tools.maven</groupId>
<artifactId>mule-maven-plugin</artifactId>
<version>${mule.maven.plugin.version}</version>
<extensions>true</extensions>
<configuration>
<sharedLibraries>
<sharedLibrary>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j</artifactId>
</sharedLibrary>
<sharedLibrary>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j-document-transformer-jsoup</artifactId>
</sharedLibrary>
<sharedLibrary>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j-document-parser-apache-tika</artifactId>
</sharedLibrary>
...
</sharedLibraries>
</configuration>
</plugin>
</plugins>
</build>
...
<dependencies>
...
<dependency>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j</artifactId>
<version>1.0.1</version>
</dependency>
<dependency>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j-document-transformer-jsoup</artifactId>
<version>1.0.1-beta6</version>
</dependency>
<dependency>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j-document-parser-apache-tika</artifactId>
<version>1.0.1-beta6</version>
```

```
</dependency>
...
</dependencies>
...
```

Step 8

Start you application
Hit http://localhost:8081/vectors/add and check response.

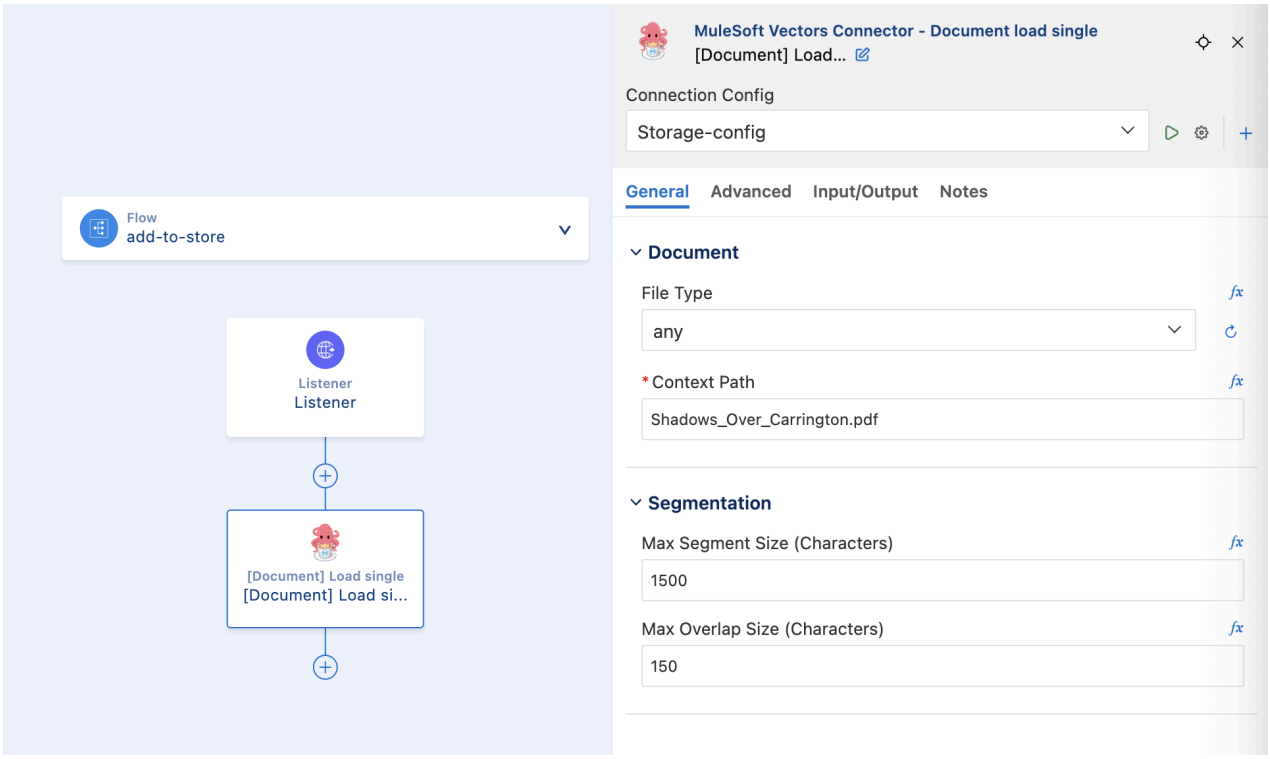


Exercise 2 - Data Splitting

Start from the exercise 1 app

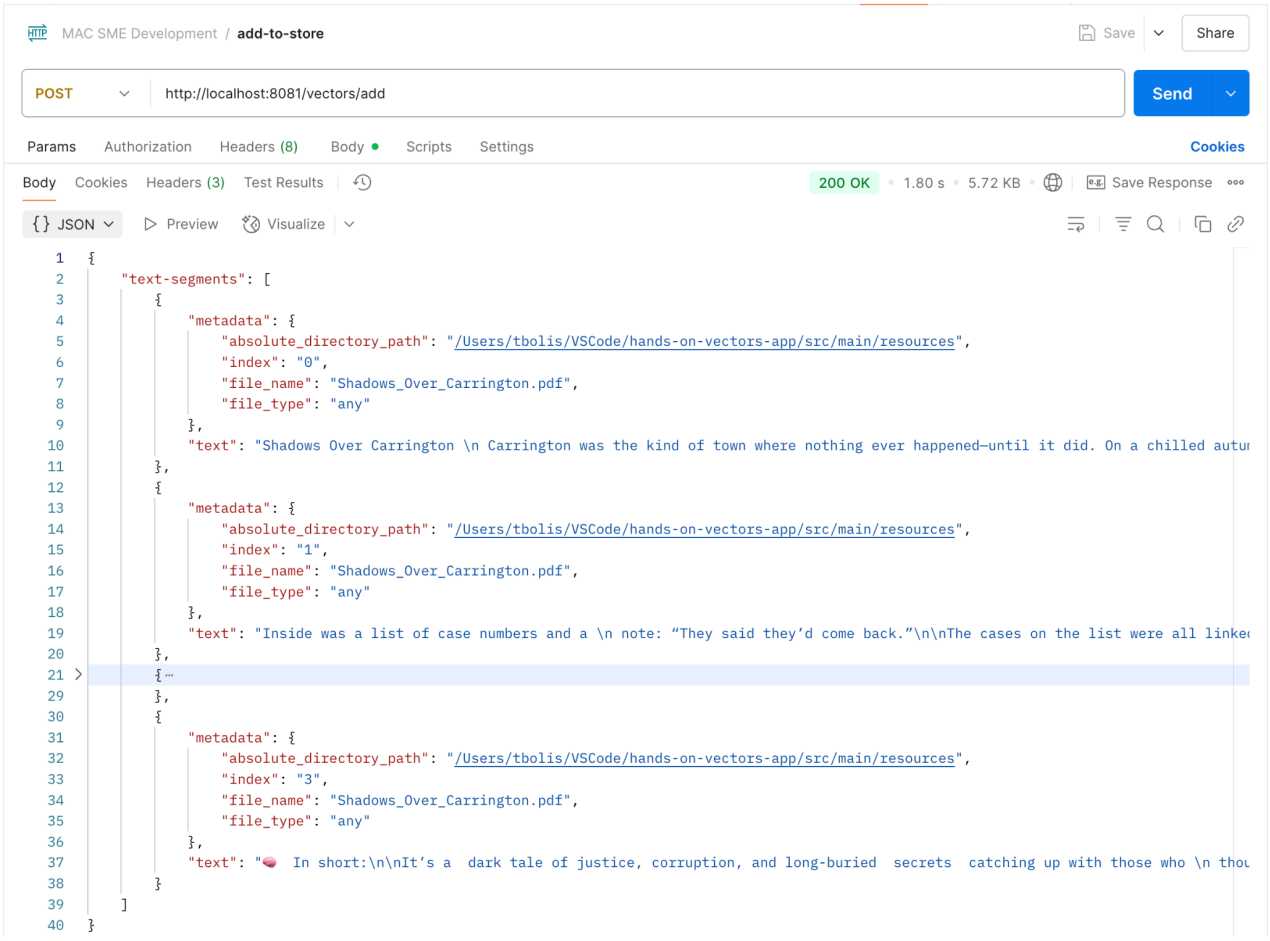
Step 1

Update [Document] Load single operation you have after listener.
Set segmentation parameters as follow:
* Max segmentation chars = 1500
* Max overlaps chars=150



Step 2

Restart your application
Hit http://localhost:8081/vectors/add and check response.



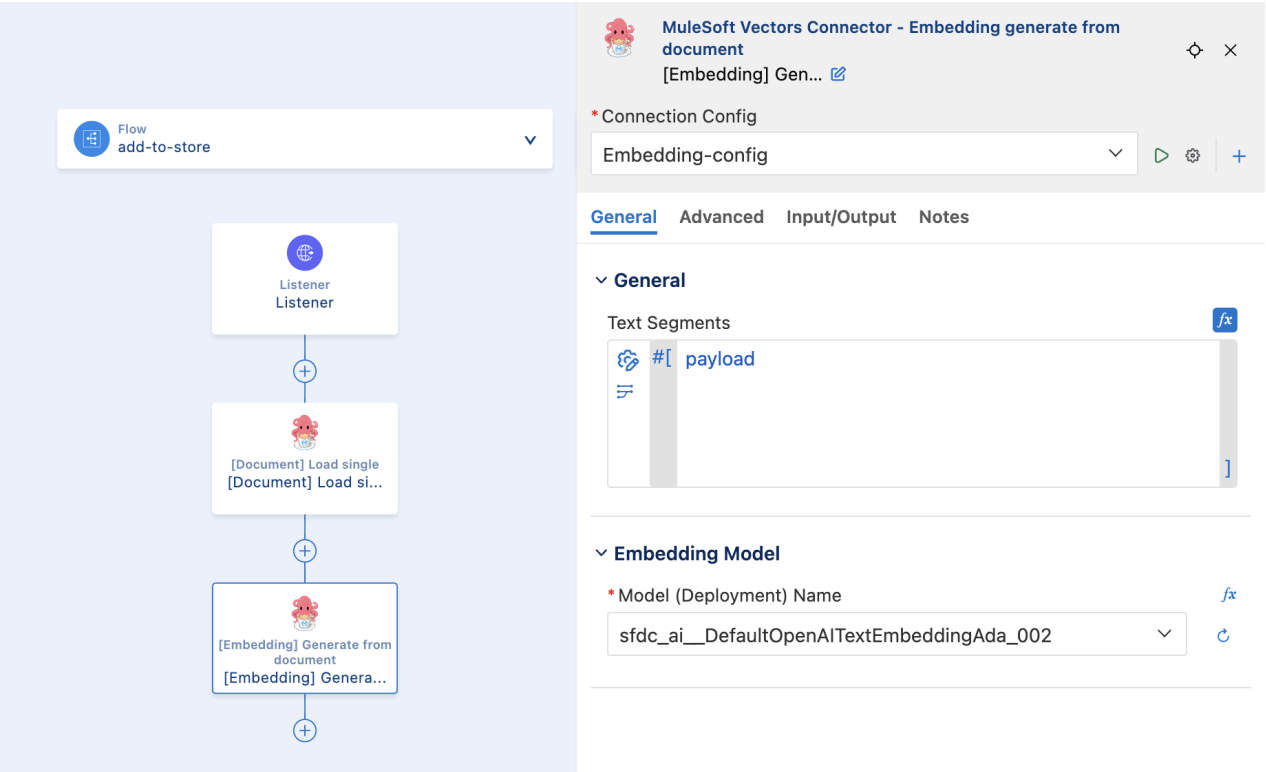
Exercise 3 - Embedding Generation

Start from the exercise 2 app

```
#hands-on-vectors configurations
handsonvectors:
Salesforce Org: storm-2fb9bac3f0ef38.my.salesforce.com
Consumer Key:
3MVG9kb26yEQGZW3tPydgyJYjCFxn9y_ipKZvKTxaEP6Kk_3Iwd5VlES3167etTkqMGFyJt
RUFerXj0tpeks2
Consumer Secret:
2CDA7023D43C6626B5E3A2BBF440D6EEE5CF0FD1708FABA8B24888F9B68A8E5A
```

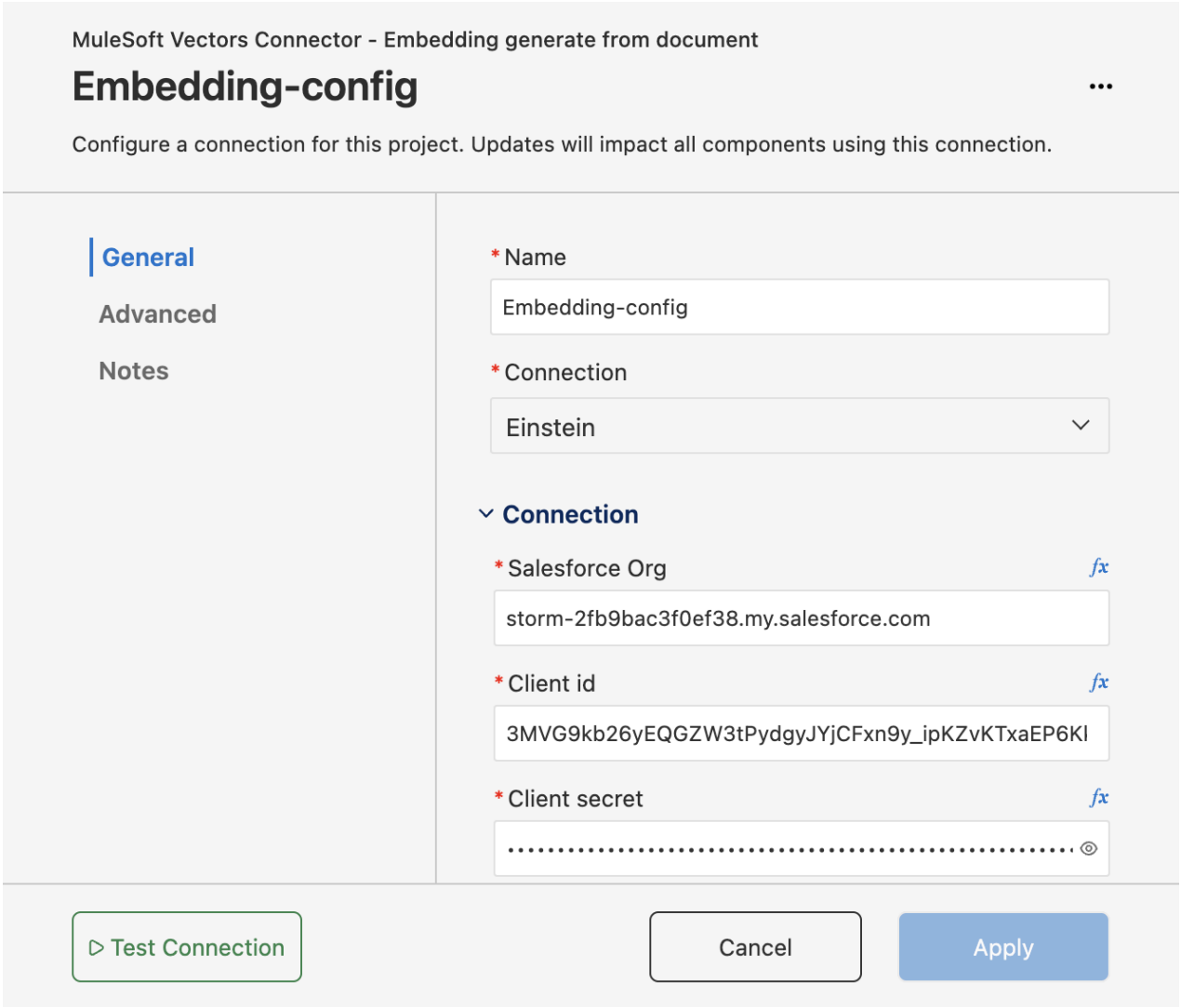
Step 1

Drop `[Embedding] Generate from Document` operation right after the `[Document] Load single` operation.



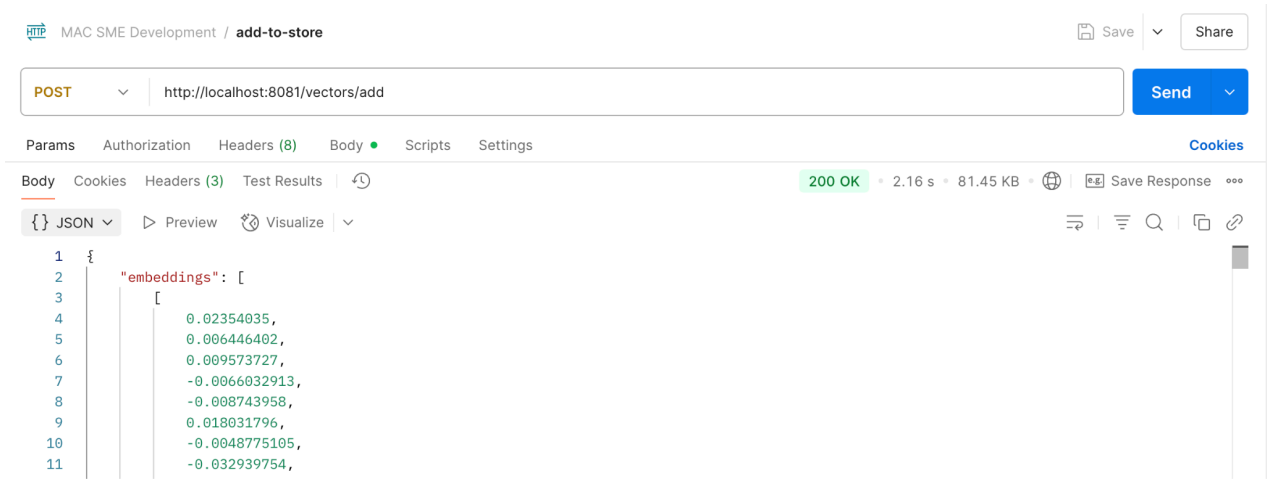
Step 2

Create an Embedding Configuration using either Einstein or Azure Open AI (from Embark) as model connection.



Step 3

Start you application
Hit http://localhost:8081/vectors/add and check response.



Exercise 4 - Store Embeddings

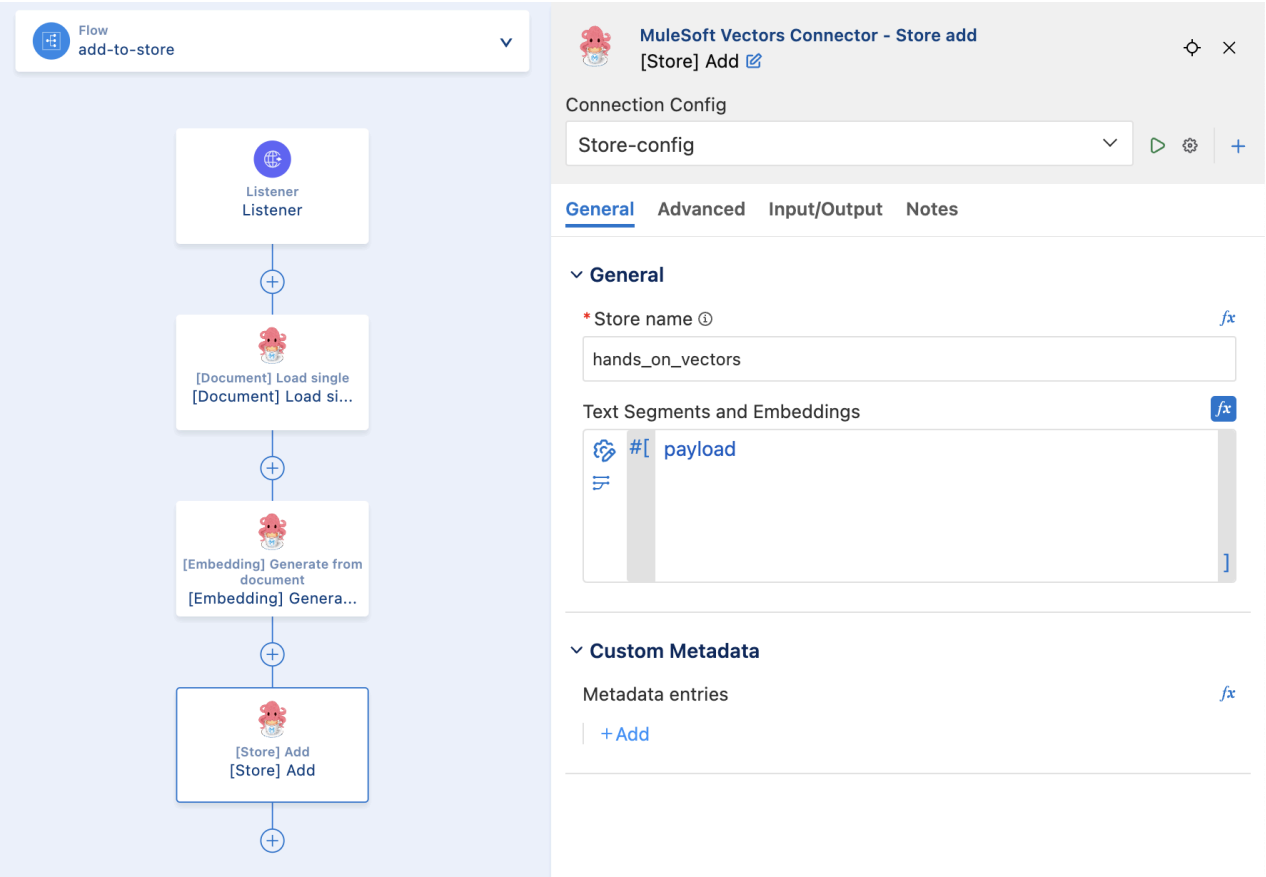
```
#heroku-hands-on configurations
herokuhandson:
  host: cloud-services.demos.mulesoft.com
  port: '30389'
  username: postgres
  databaseName: sampledb
  password: TVK4pah_tze3dqe5qvx
  rootPassword: ''
```

Start from the exercise 3 app

Step 1

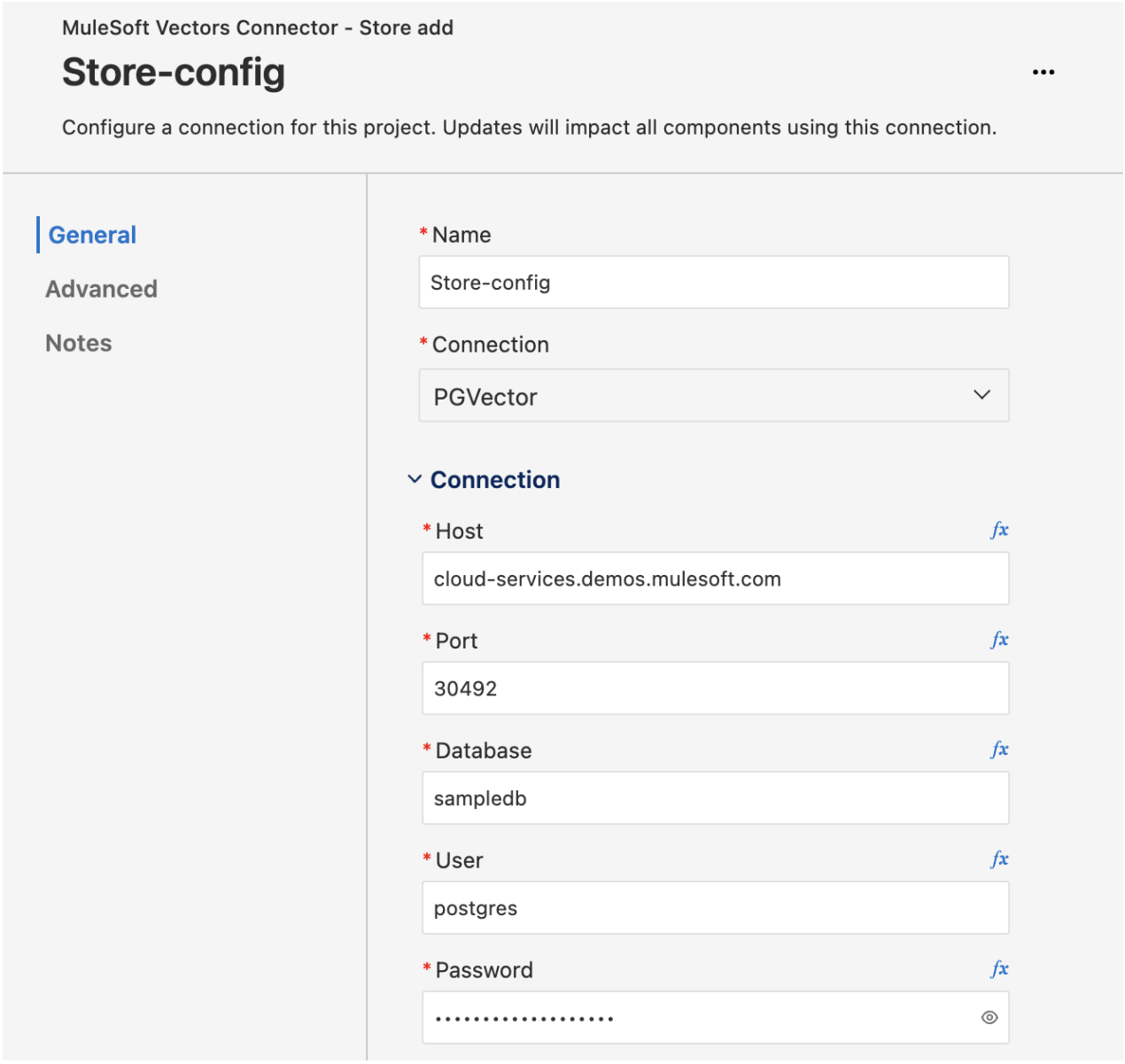
Drop [Store] Add operation right after the [Embedding] Generate from Document operation. Configure the operation by setting a store name.

Store name: `hands_on_vectors_<your-user>`



Step 2

Create a Store Configuration using PGVector as store connection.



Step 3

Add Shared Libraries by editing directly edit the pom.xml

```
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-clean-plugin</artifactId>
<version>3.1.0</version>
</plugin>
<plugin>
```

```
<groupId>org.mule.tools.maven</groupId>
<artifactId>mule-maven-plugin</artifactId>
<version>${mule.maven.plugin.version}</version>
<extensions>true</extensions>
<configuration>
<sharedLibraries>
<sharedLibrary>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j-pgvector</artifactId>
</sharedLibrary>
...
</sharedLibraries>
</configuration>
</plugin>
</plugins>
</build>
...
<dependencies>
...
<dependency>
<groupId>dev.langchain4j</groupId>
<artifactId>langchain4j-pgvector</artifactId>
<version>1.0.1-beta6</version>
</dependency>
...
</dependencies>
...
```

Step 4

Start your application
Hit http://localhost:8081/vectors/add and check response.

MAC SME Development / **add-to-store**

POST http://localhost:8081/vectors/add

Params Authorization Headers (8) Body Scripts Settings

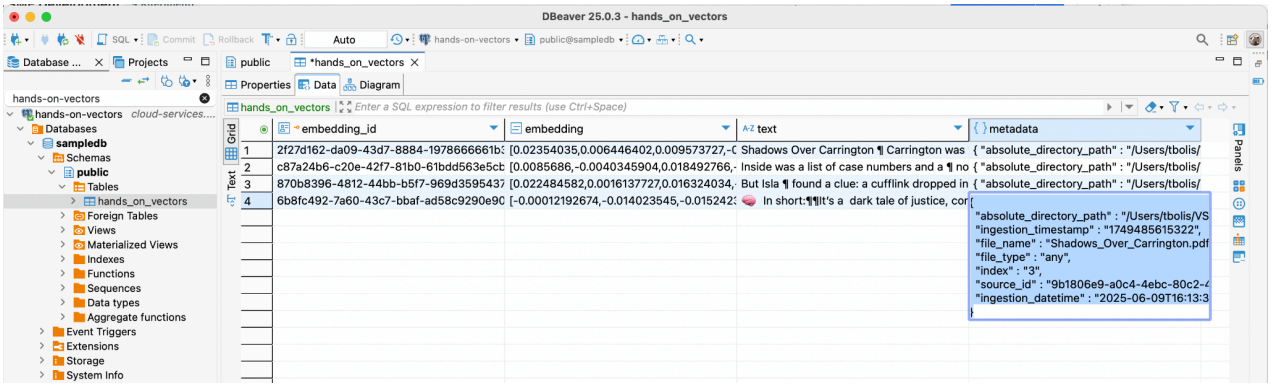
Body Cookies Headers (3) Test Results

{} JSON Preview Visualize

```
{
    "sourceId": "9b1806e9-a0c4-4ebc-80c2-4164c38c2dc2",
    "embeddingIds": [
        "2f27d162-da09-43d7-8884-1978666661b3",
        "c87a24b6-c20e-42f7-81b0-61bdd563e5cb",
        "870b8396-4812-44bb-b5f7-969d35954377",
        "6b8fc492-7a60-43c7-bbaf-ad58c9290e90"
    ],
    "status": "updated"
}
```

Step 5

Connect to your PostgreSQL database using a client as DBeaver and check the data.



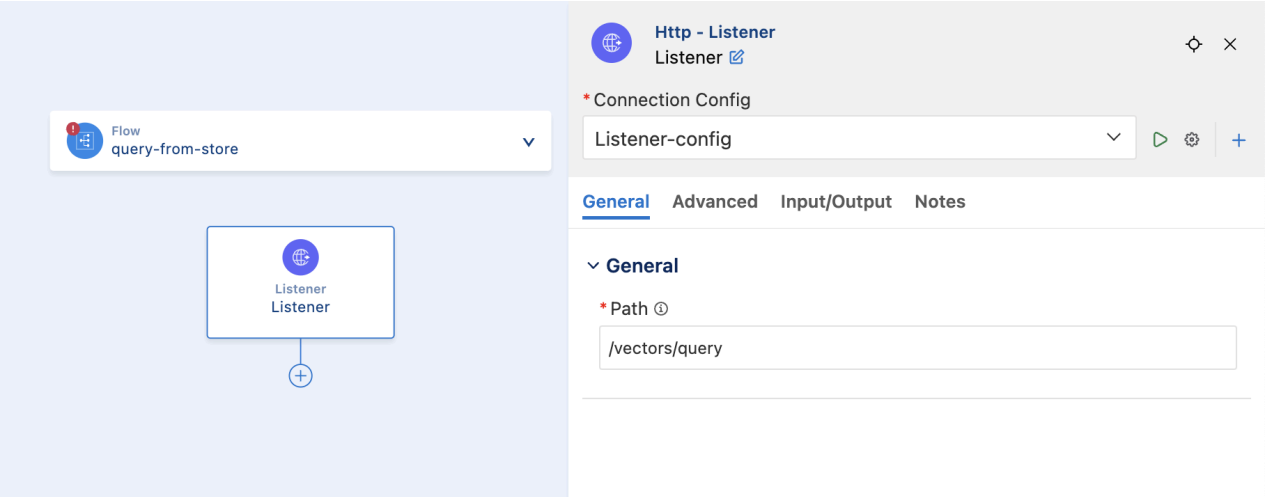
Exercise 5 - Query Vector Store

Start from the exercise 4 app.

Step 1

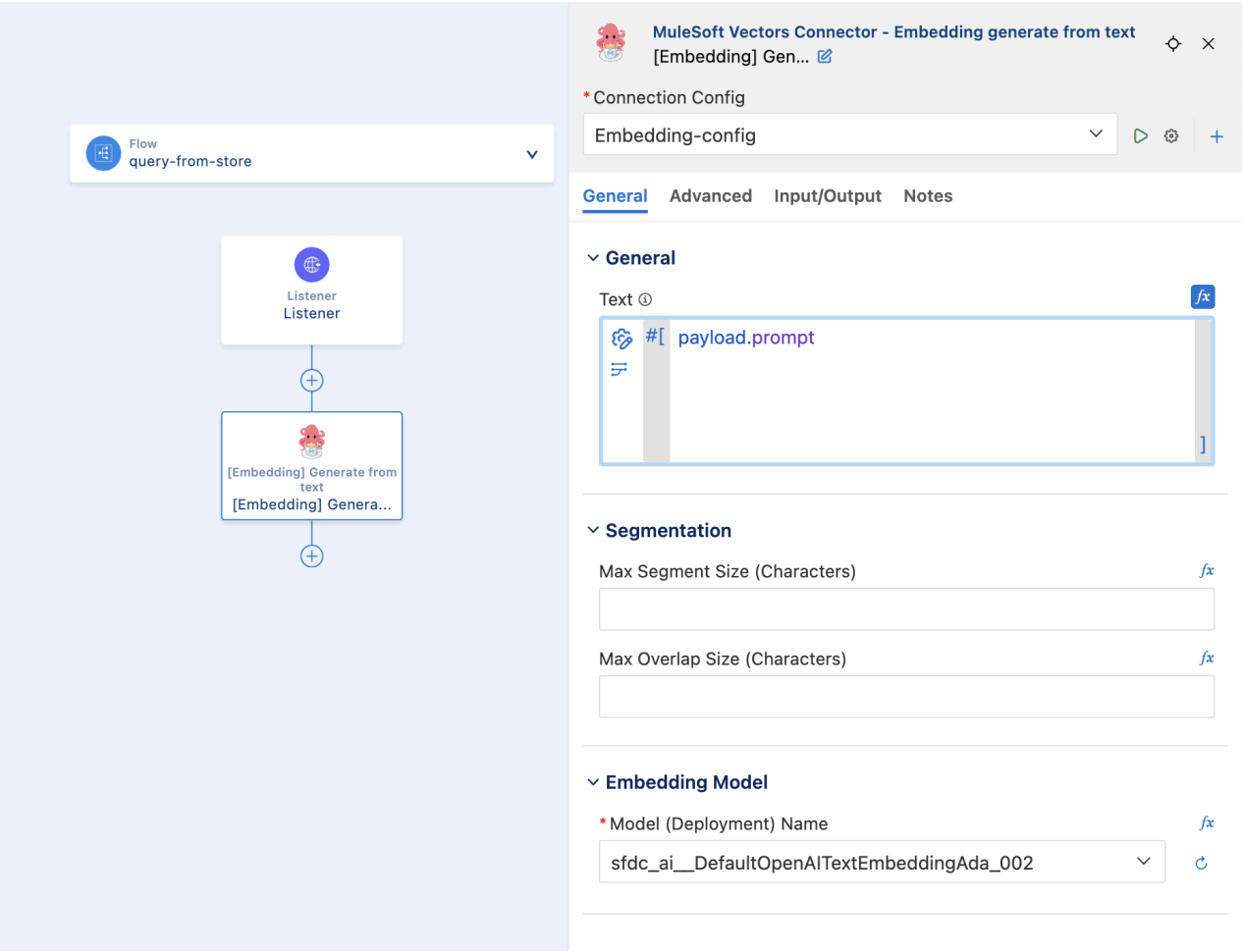
Drop http listener (/vectors/query) in a new flow, you'll use it to query the store using postman. It should accept a json payload like

```
{
"prompt": "question"
}
```



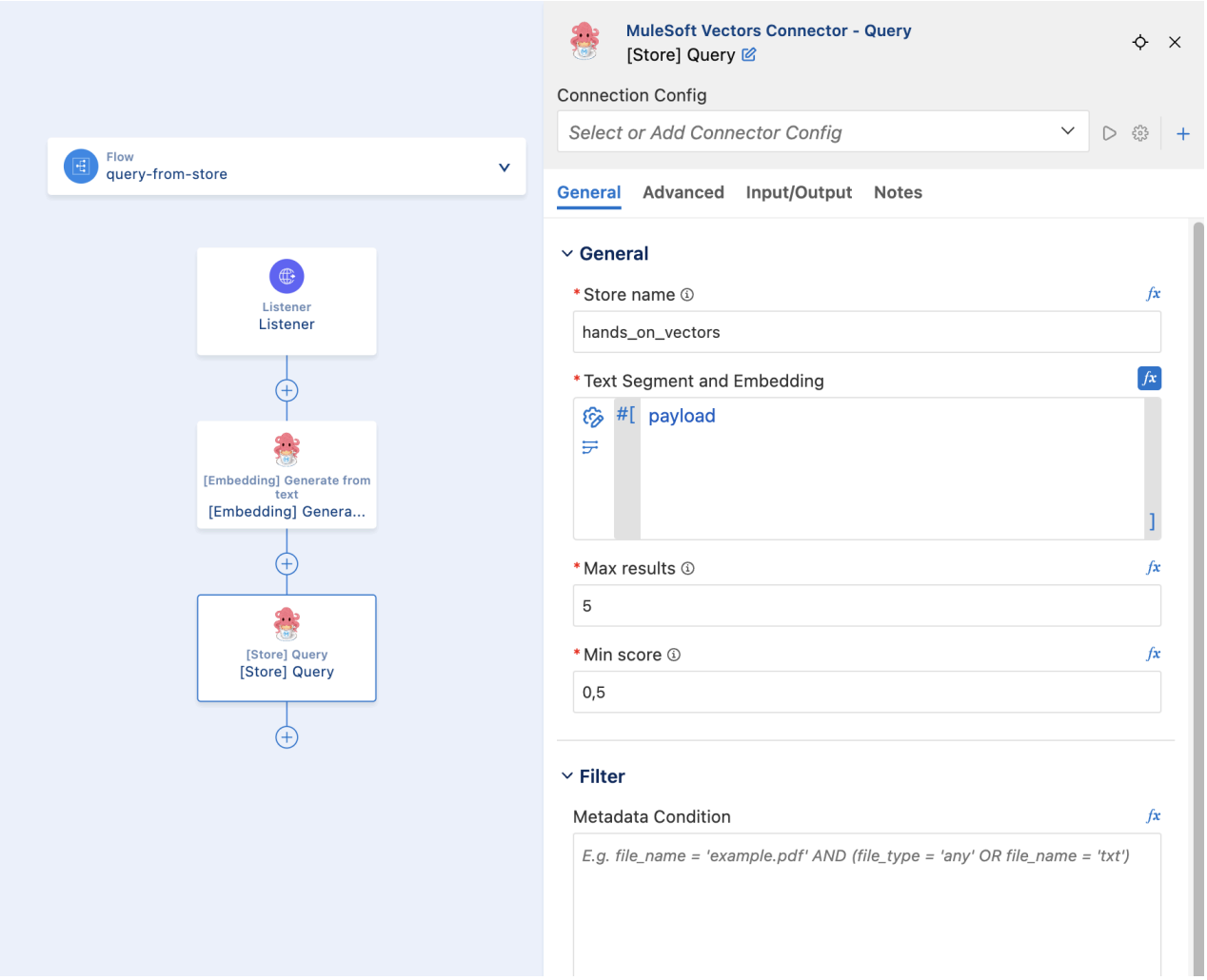
Step 2

Drop `[Embedding] Generate from Text` operation after the listener. Set payload.prompt as input.



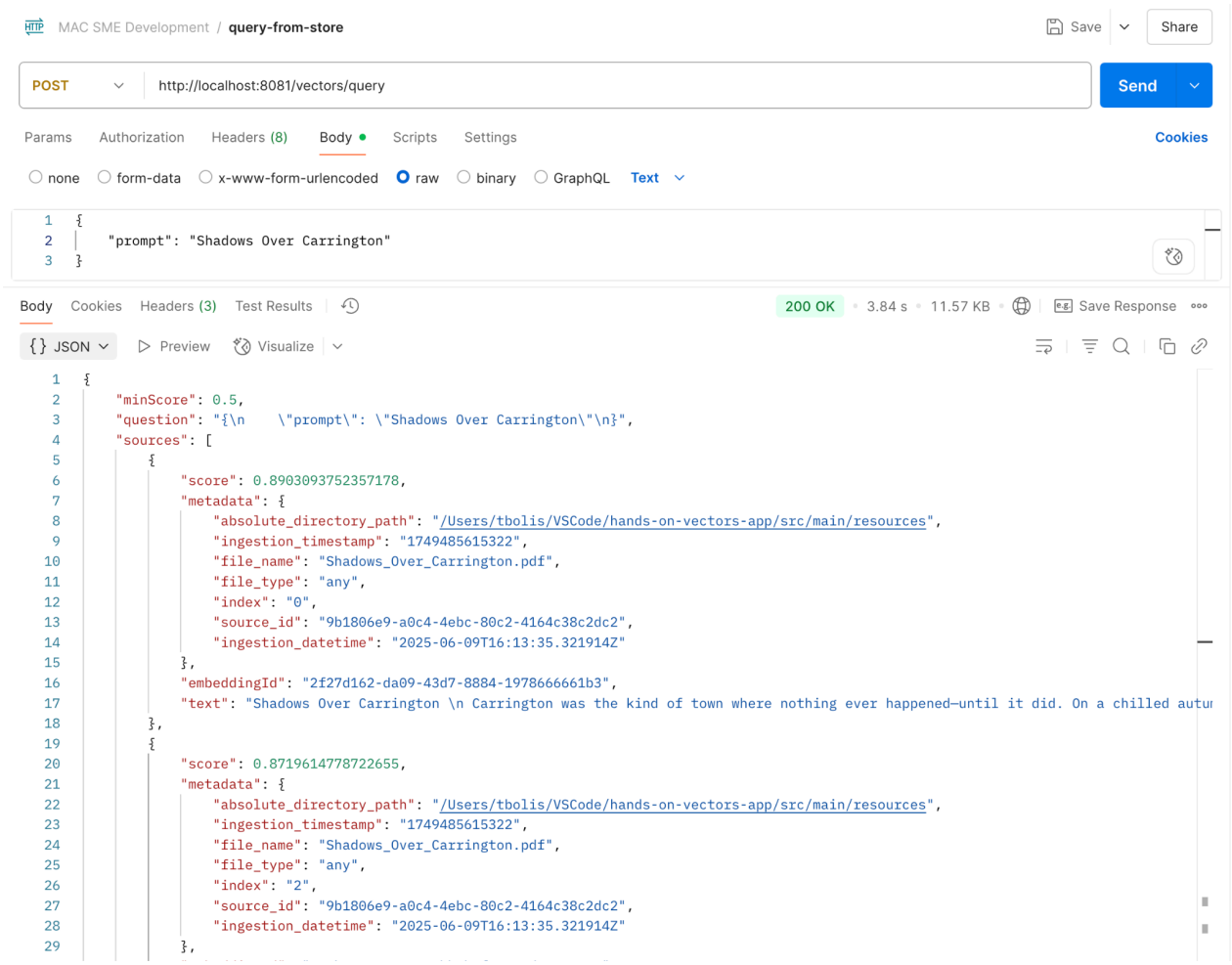
Step 3

Drop `[Store] Query` operation right after the `[Embedding] Generate from Text` operation. properly set the store name.



Step 4

Start you application
Hit http://localhost:8081/vectors/query and check response.

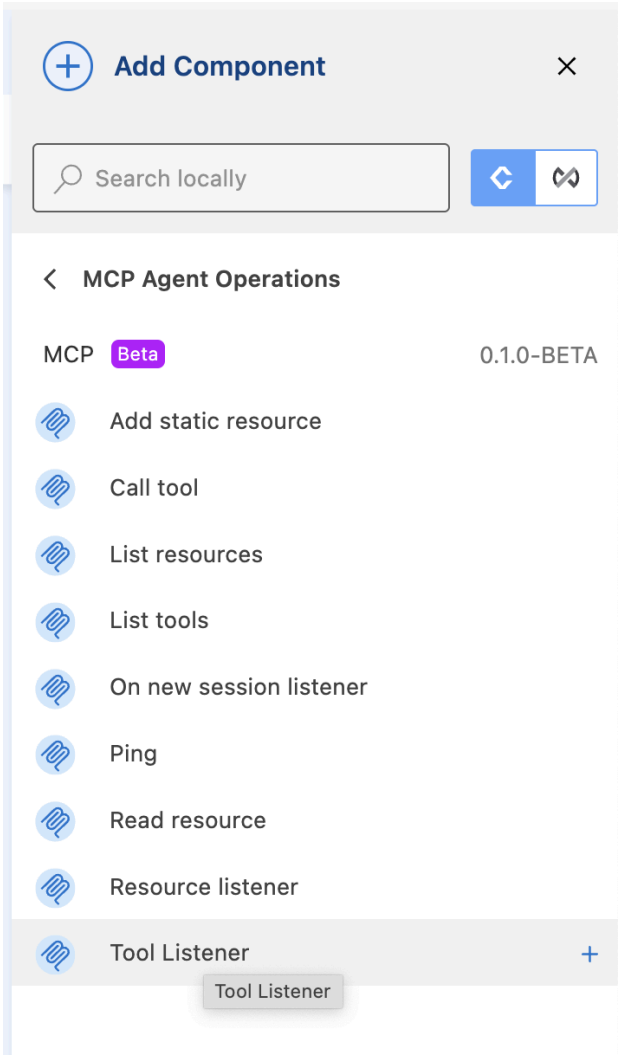


Exercise 6 - Vector Store as MCP Tool

Start from the exercise 5 app.

Step 1

Add MCP Connector to your app taking it from exchange.



Step 2

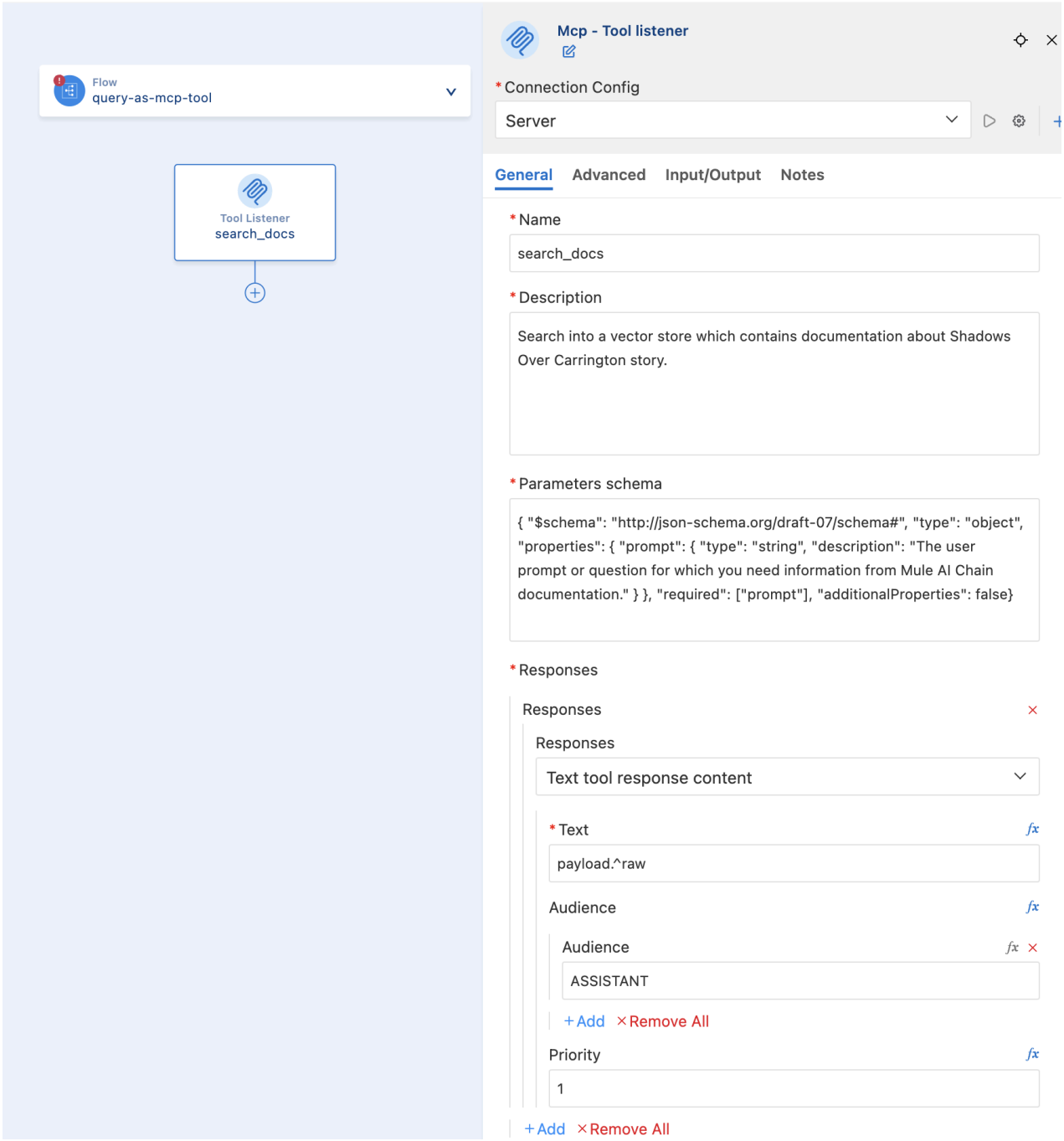
Drop MCP Tool Listener and configure it as follow:

- Name: `search_docs`
- Description: `Search into a vector store which contains documentation about Shadows Over Carrington story.`
- Parameters schema:

```
{
"$schema": "http://json-schema.org/draft-07/schema#",
"type": "object",
"properties": {
"prompt": {
```

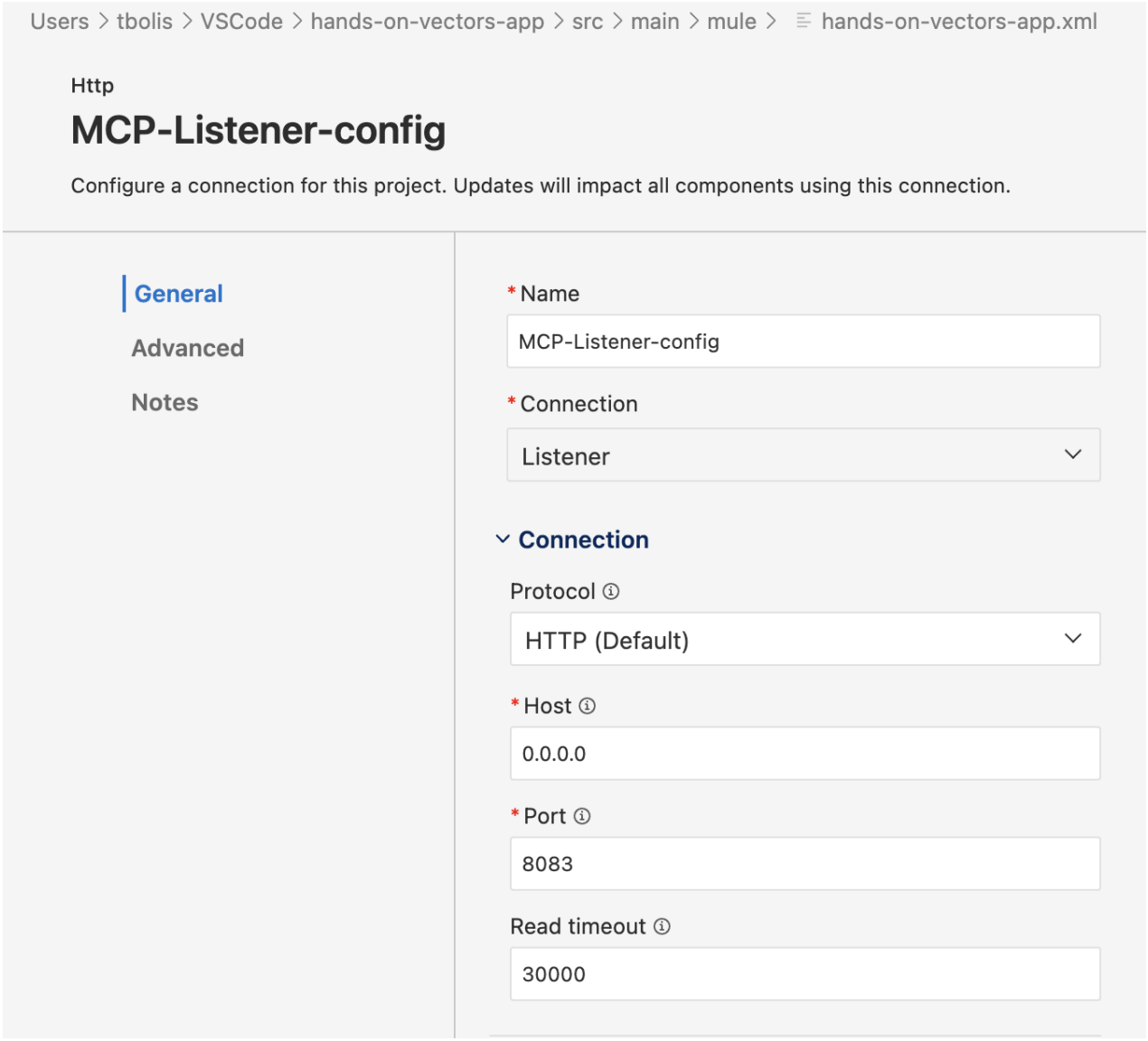
```
"type": "string",
"description": "The user prompt or question for which you need
information from Mule AI Chain documentation."
}
},
"required": ["prompt"],
"additionalProperties": false
}
```

- Responses:
- Text: `payload.^raw`
- Audience: `ASSISTANT`
- Priority: `1`



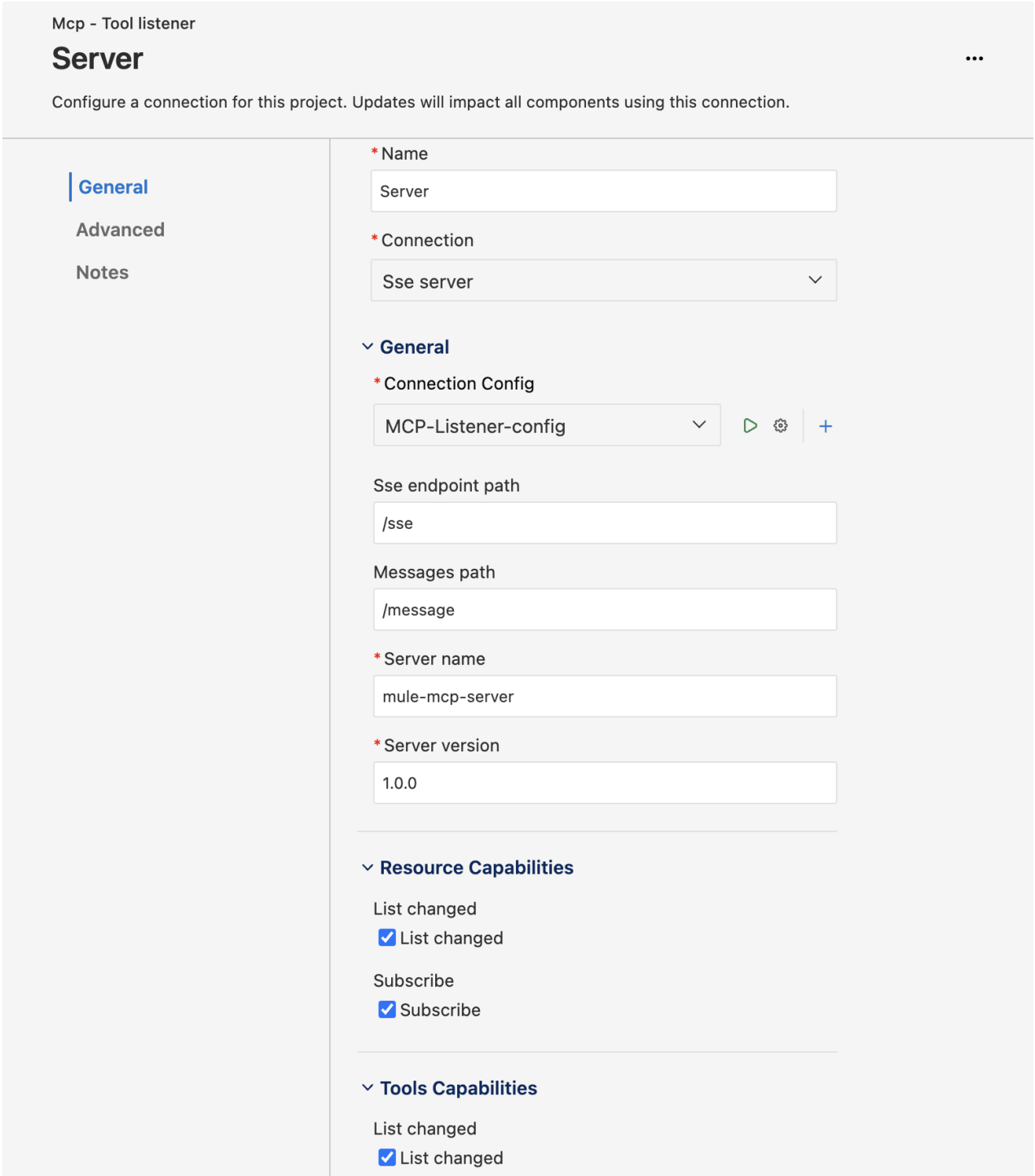
Step 3

Create a new HTTP Listener configuration using port 8083



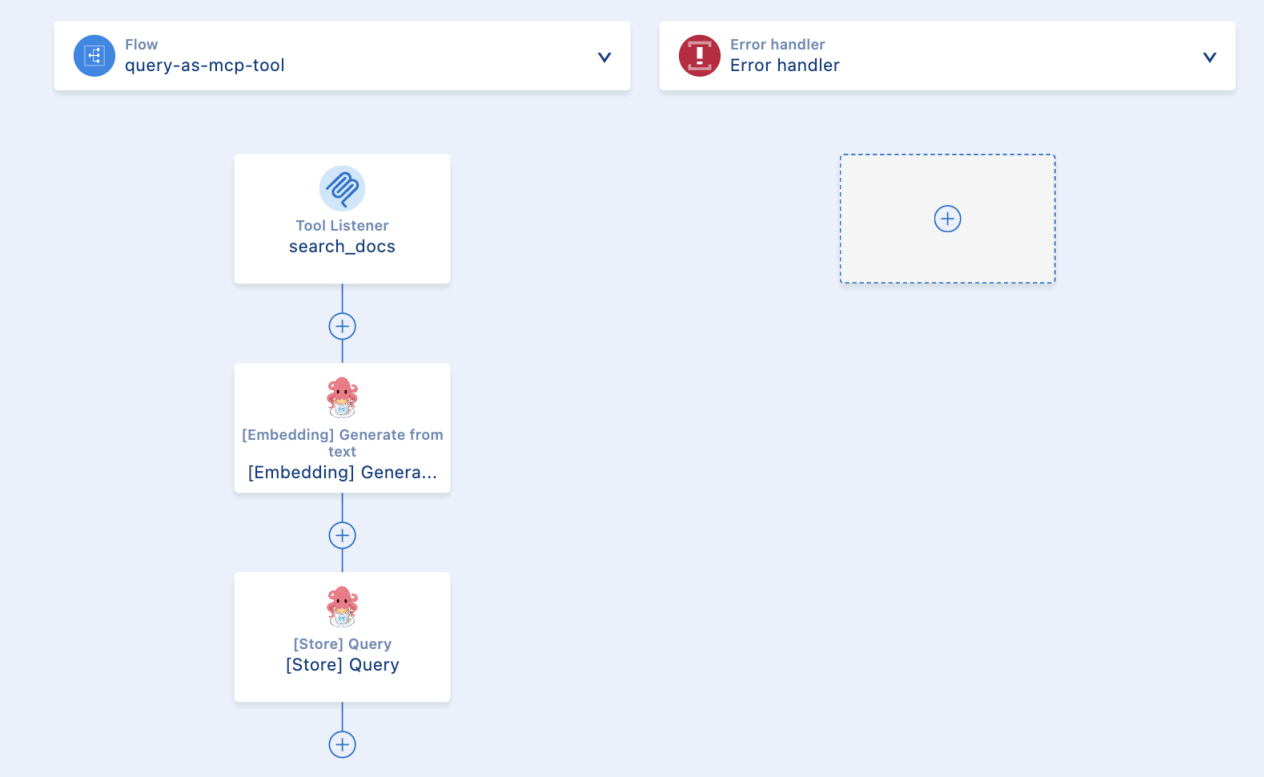
Step 4

Create a new MCP Server Configuration using Http Listener configuration created at previous step



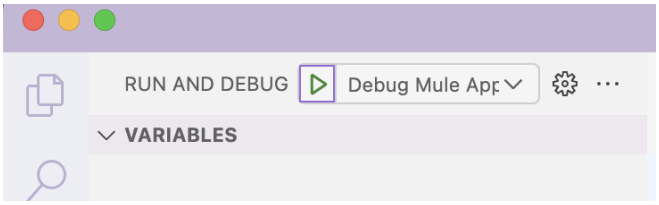
Step 5

After MCP Tool Listener add same logic implemented in previous exercise for querying



Step 6

Start your application

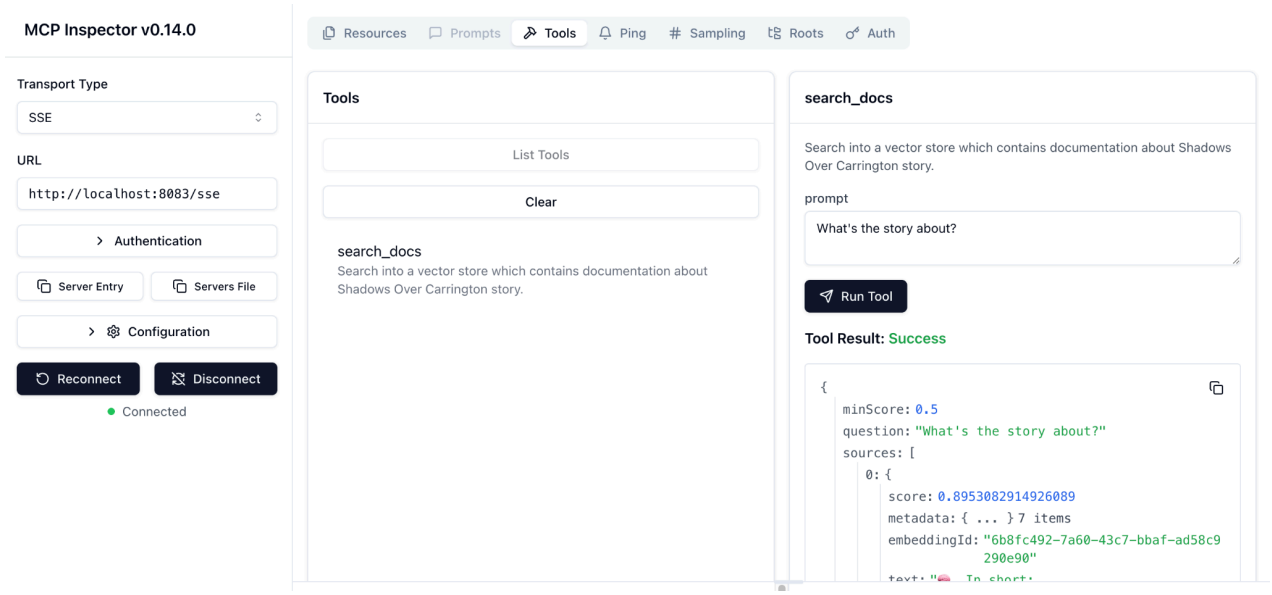


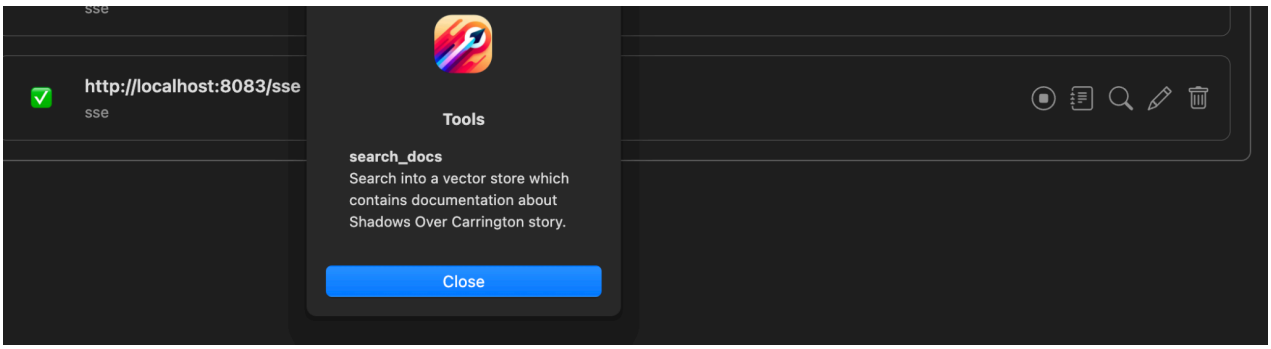
Step 7

Test you MCP Server with URL http://localhost:8083/sse using one client as Witsy or the MCP Inspector.

- MCP Inspector available at https://github.com/modelcontextprotocol/inspector

- Witsy available at https://witsyai.com/


MCP Inspector v0.14.0
Transport Type
SSE
URL
http://localhost:8083/sse
Authentication
Server Entry
Servers File
Configuration
Reconnect
Disconnect
Connected
Resources
Prompts
Tools
Ping
Sampling
Roots
Auth
Tools
List Tools
Clear
search_docs
Search into a vector store which contains documentation about Shadows Over Carrington story.
search_docs
Search into a vector store which contains documentation about Shadows Over Carrington story.
prompt
What's the story about?
Run Tool
Tool Result: Success
{
minScore: 0.5
question: "What's the story about?"
sources: [
0: {
score: 0.8953082914926089
metadata: { ... } 7 items
embeddingId: "6b8fc492-7a60-43c7-bbaf-ad58c9290e90"


http://localhost:8083/sse
sse
Tools
search_docs
Search into a vector store which contains documentation about Shadows Over Carrington story.
Close

WebCrawler Hands-on

WebCrawler Connector Hands-on

Anypoint Studio (WebCrawler)

ACB (WebCrawler)

ACB (WebCrawler)

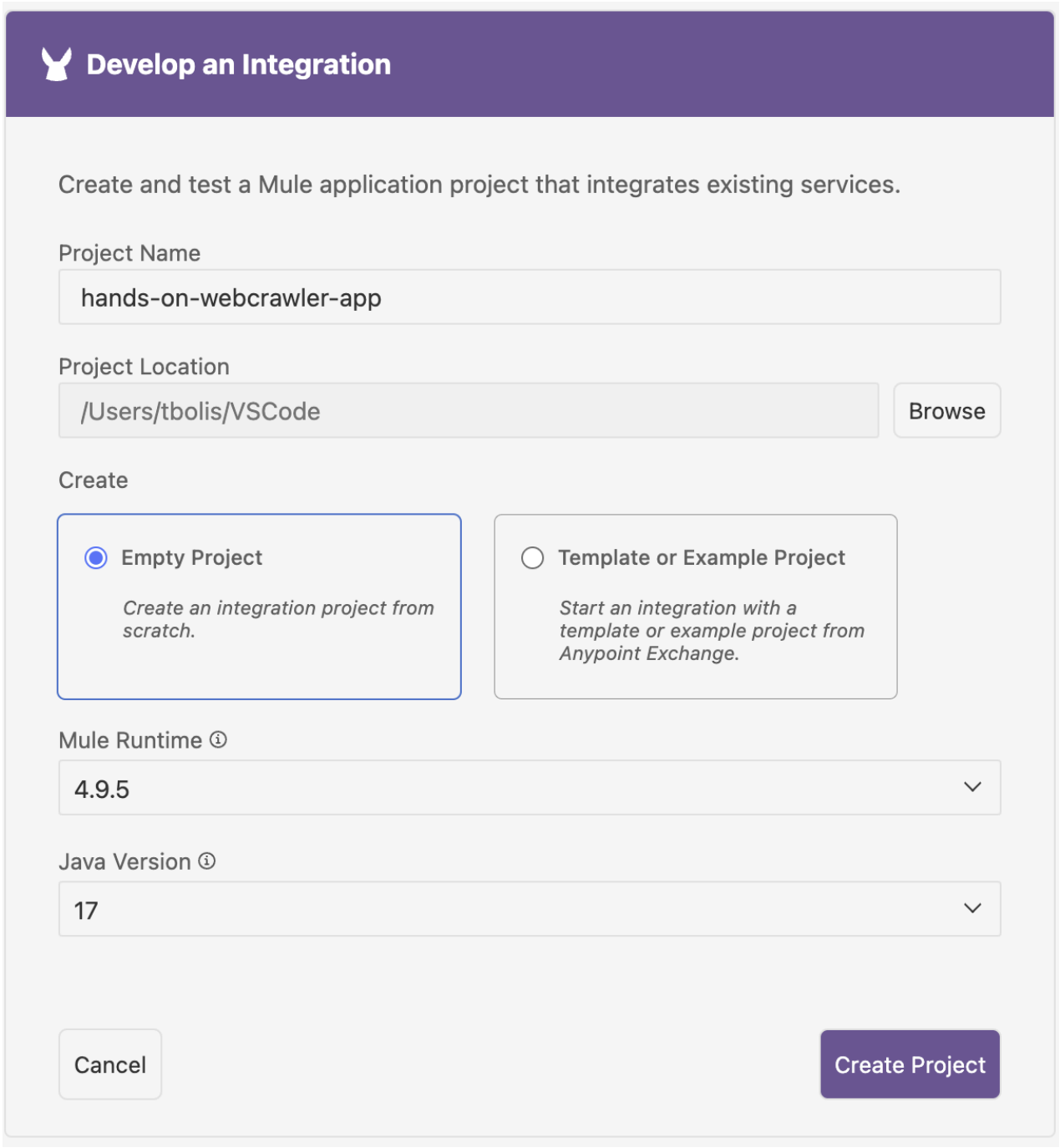
Table of Contents

Exercise 1 - Get Static Page **1**
Exercise 2 - Get Dynamic Page **1**
Exercise 3 - Crawl Website **1**
Exercise 4 - Crawl and ingest into Store **1**
Exercise 5 - Page Crawl as MCP Tool **1**

Exercise 1 - Get Static Page

Step 1

Create a new mule app `hands-on-webcrawler-app` using Studio or ACB.
or
Start from the hands-on-vectors-app if you plan to do Exercise 4

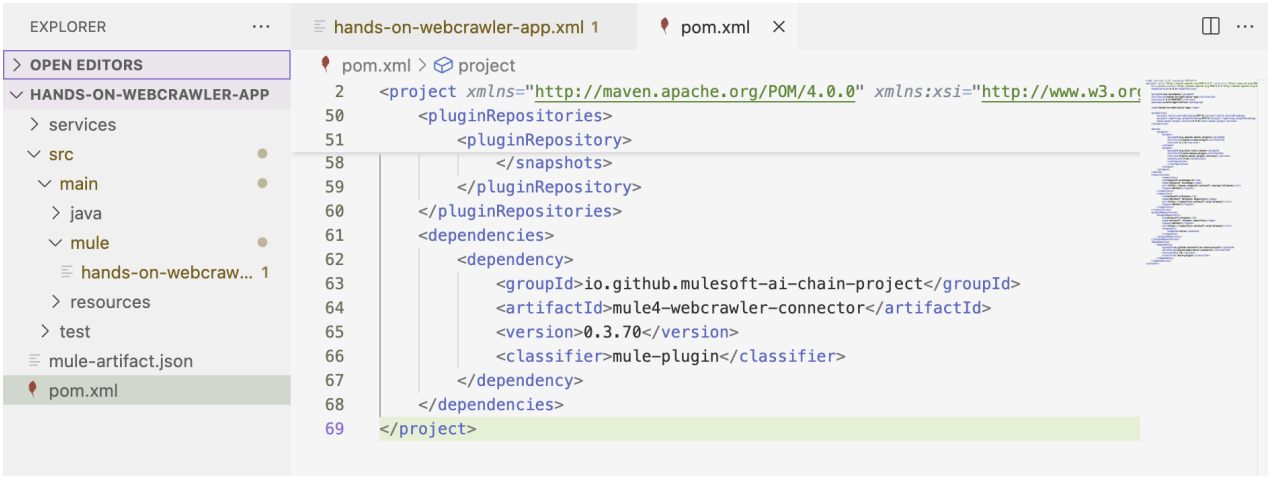


Step 2

Add WebCrawler dependency to your pom.xml

```
<dependency>
<groupId>io.github.mulesoft-ai-chain-project</groupId>
<artifactId>mule4-webcrawler-connector</artifactId>
```

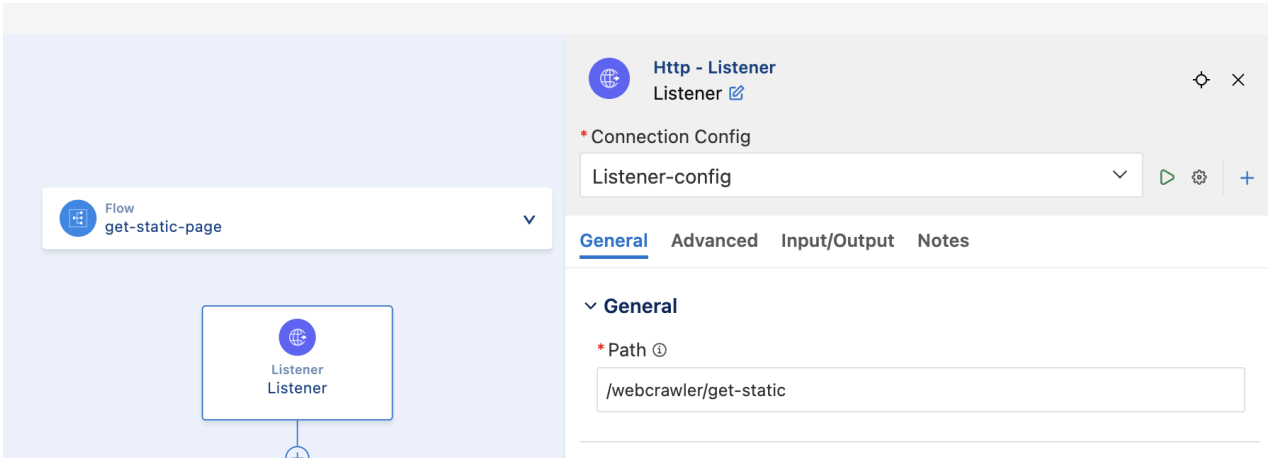
```
<version>0.3.70</version>
<classifier>mule-plugin</classifier>
</dependency>
```



Step 3

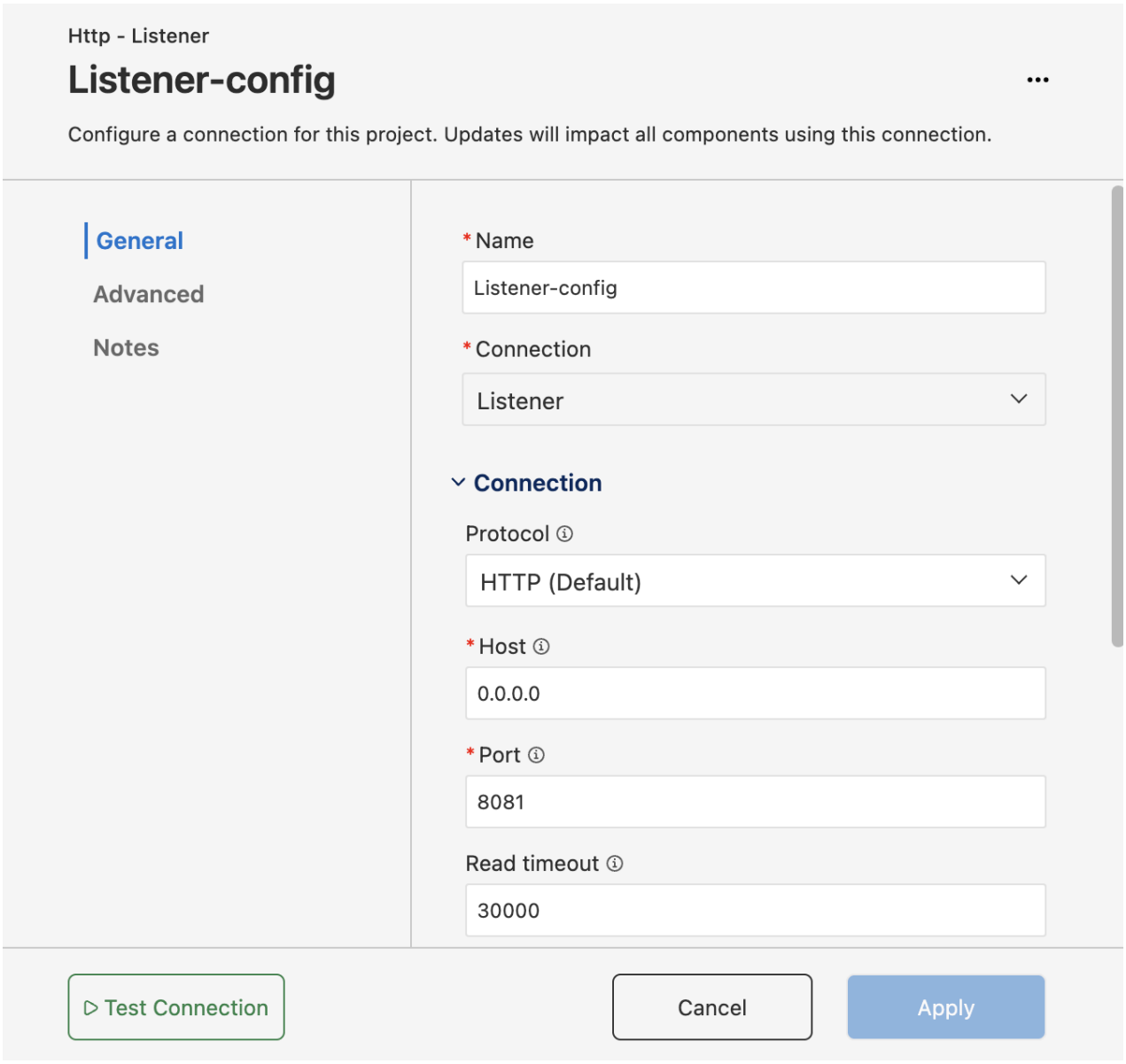
Drop http listener (/webcrawler/get-static) in a new flow, you'll use it to get page content in a static way. You'll use it to get page content using postman. It should accept a json payload like

```
{
"url": "https://mac-project.ai/",
"outputFormat": "MARKDOWN"
}
```



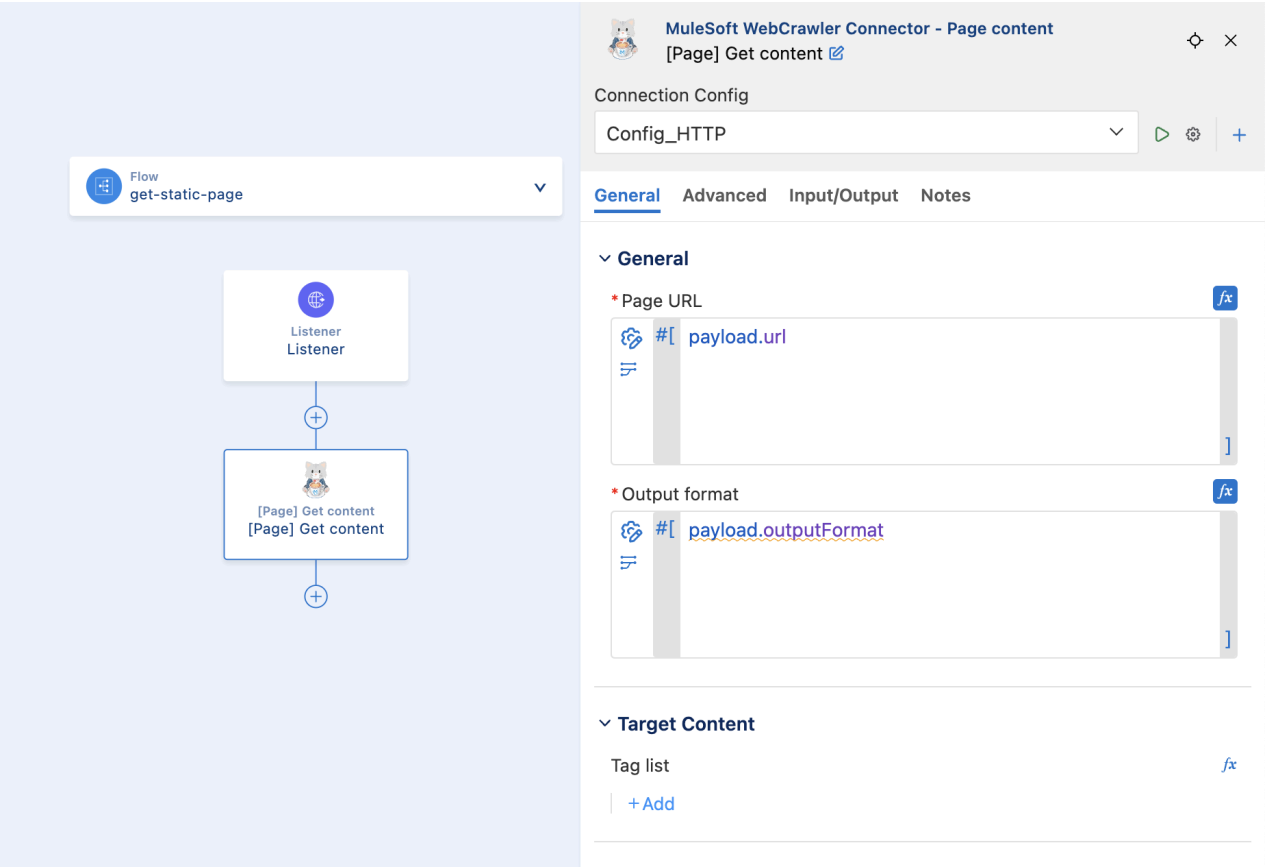
Step 4

Create Listener Configuration



Step 5

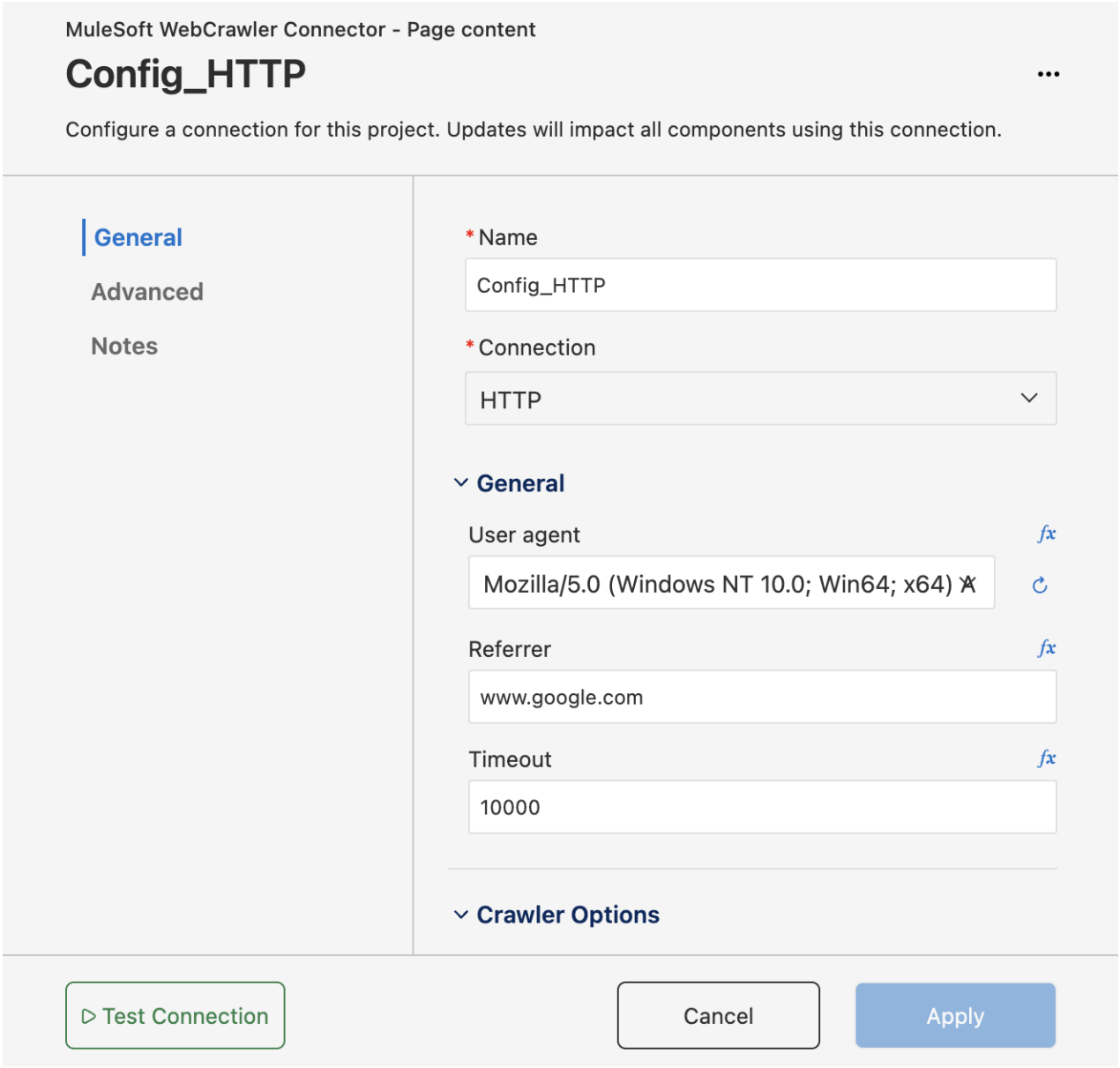
Drop [Page] Get content operation after listener.
Set url as payload.url and output format as payload.outputFormat



Step 6

Create WebCrawler Configuration (with HTTP Connection).

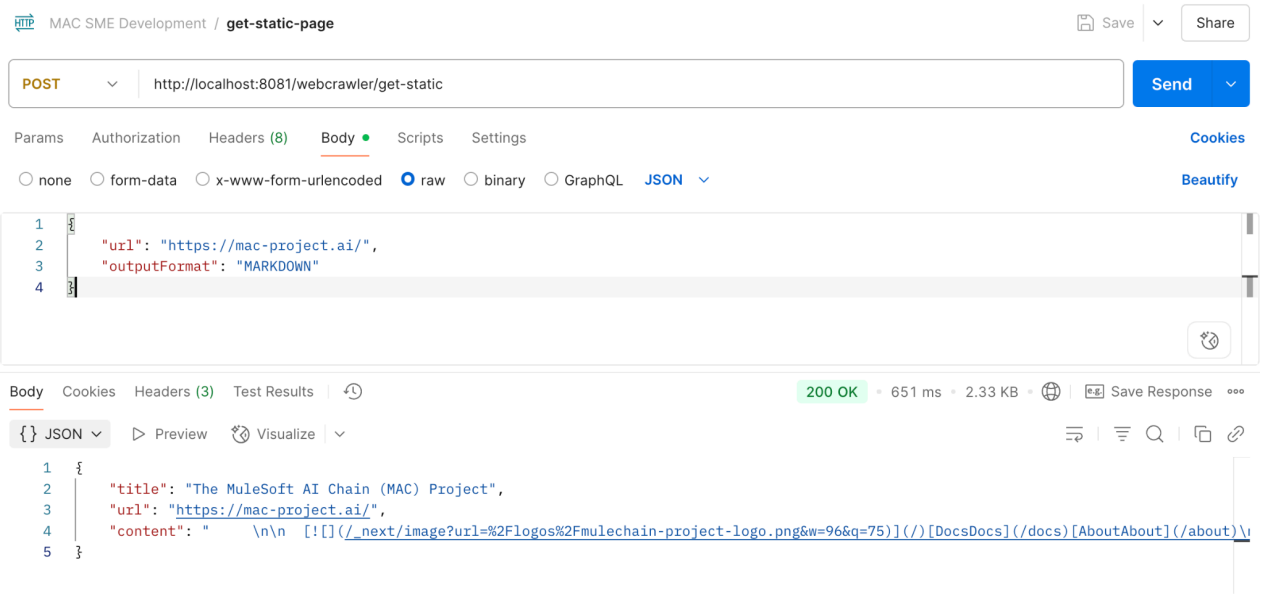
* User Agent: `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36`



Step 7

Start you application
Hit http://localhost:8081/webcrawler/get-static and check response.
Request payload:

```
{
"url": "https://mac-project.ai/",
"outputFormat": "MARKDOWN"
}
```

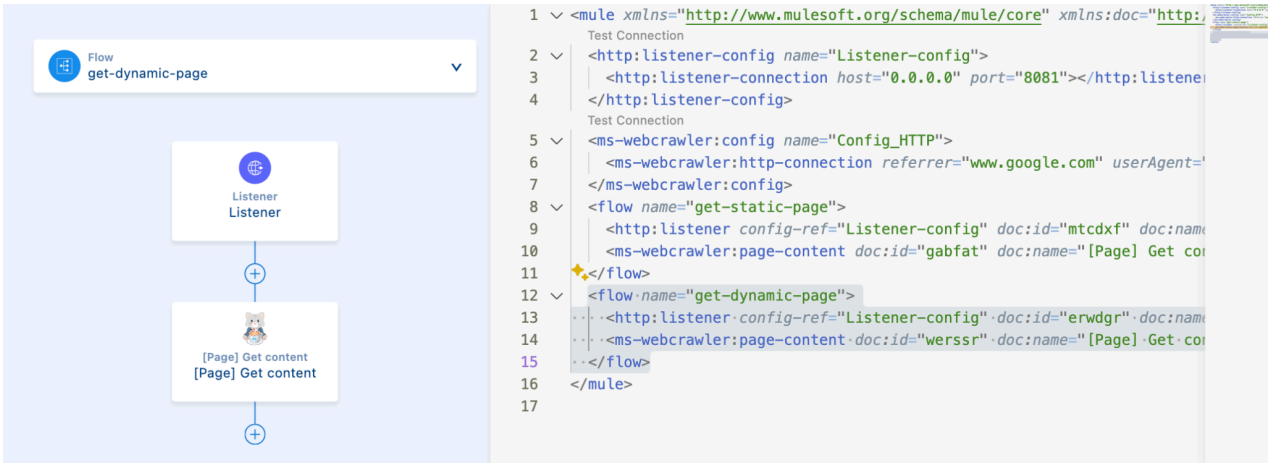


Exercise 2 - Get Dynamic Page

Start from the exercise 1 app.

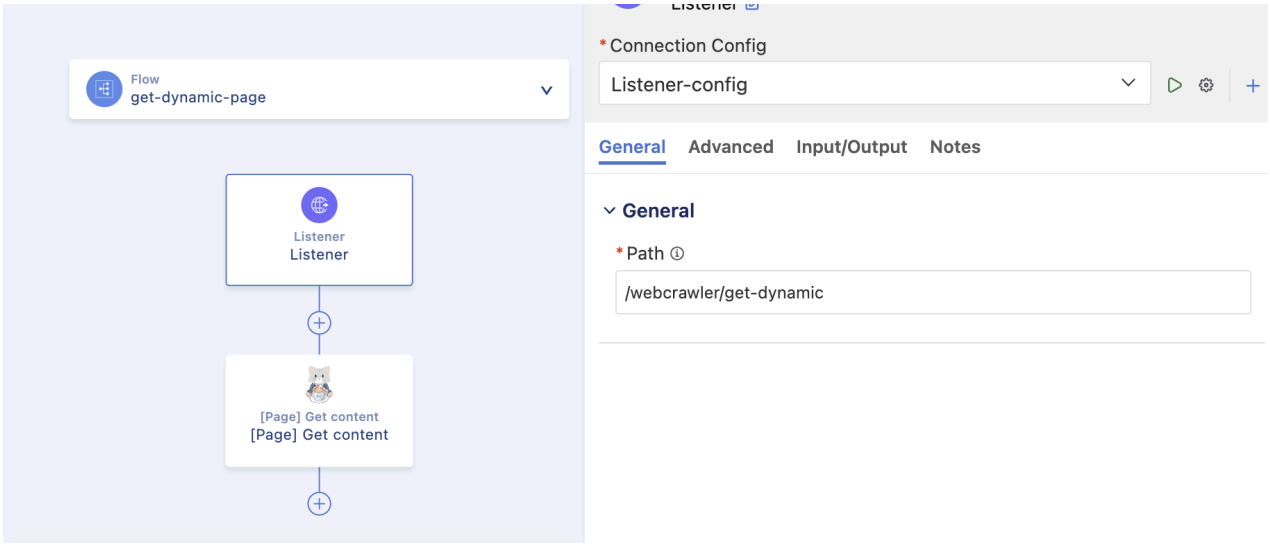
Step 1

Copy the flow from previous exercise.



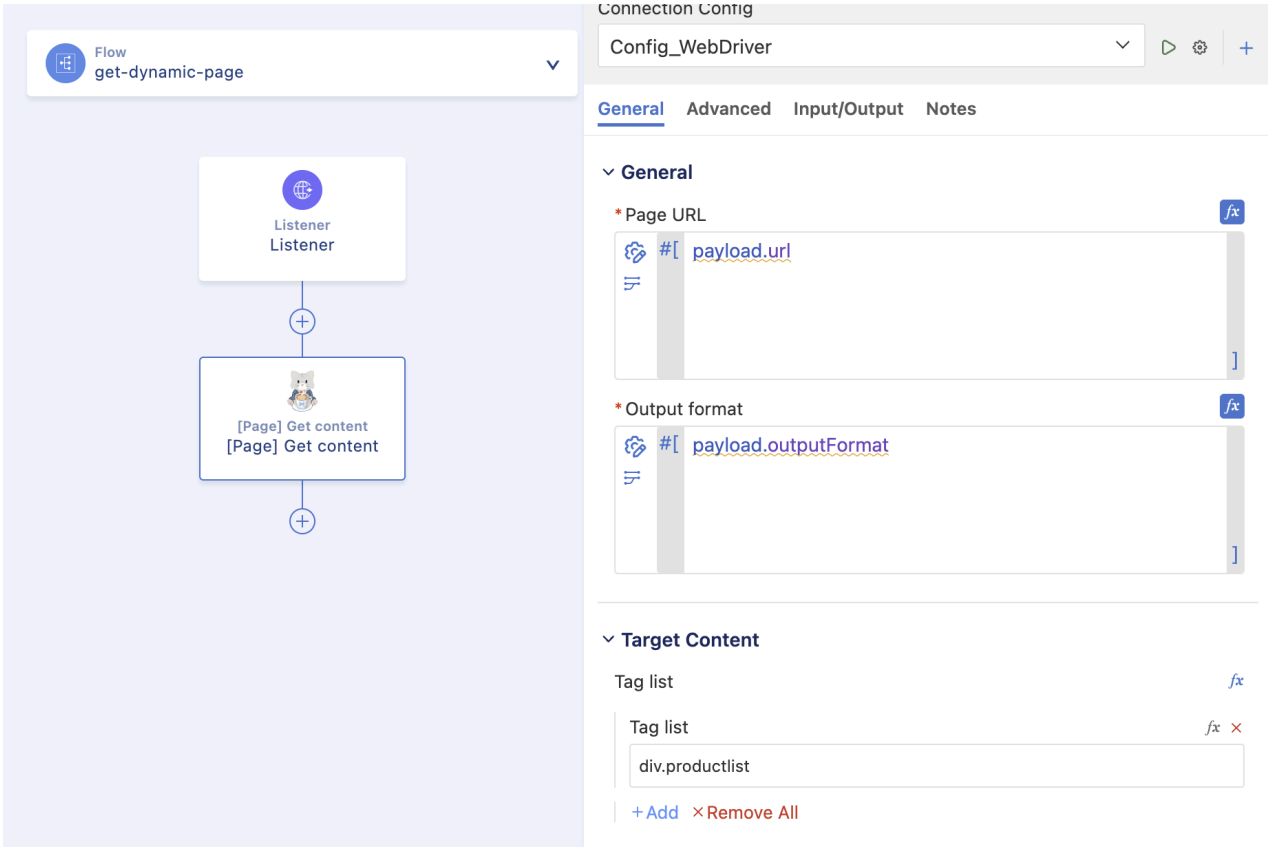
Step 2

Update listener with path /webcrawler/get-dynamic



Step 3

Update `[Page] Get content` operation and set a new Tag `div.productlist`



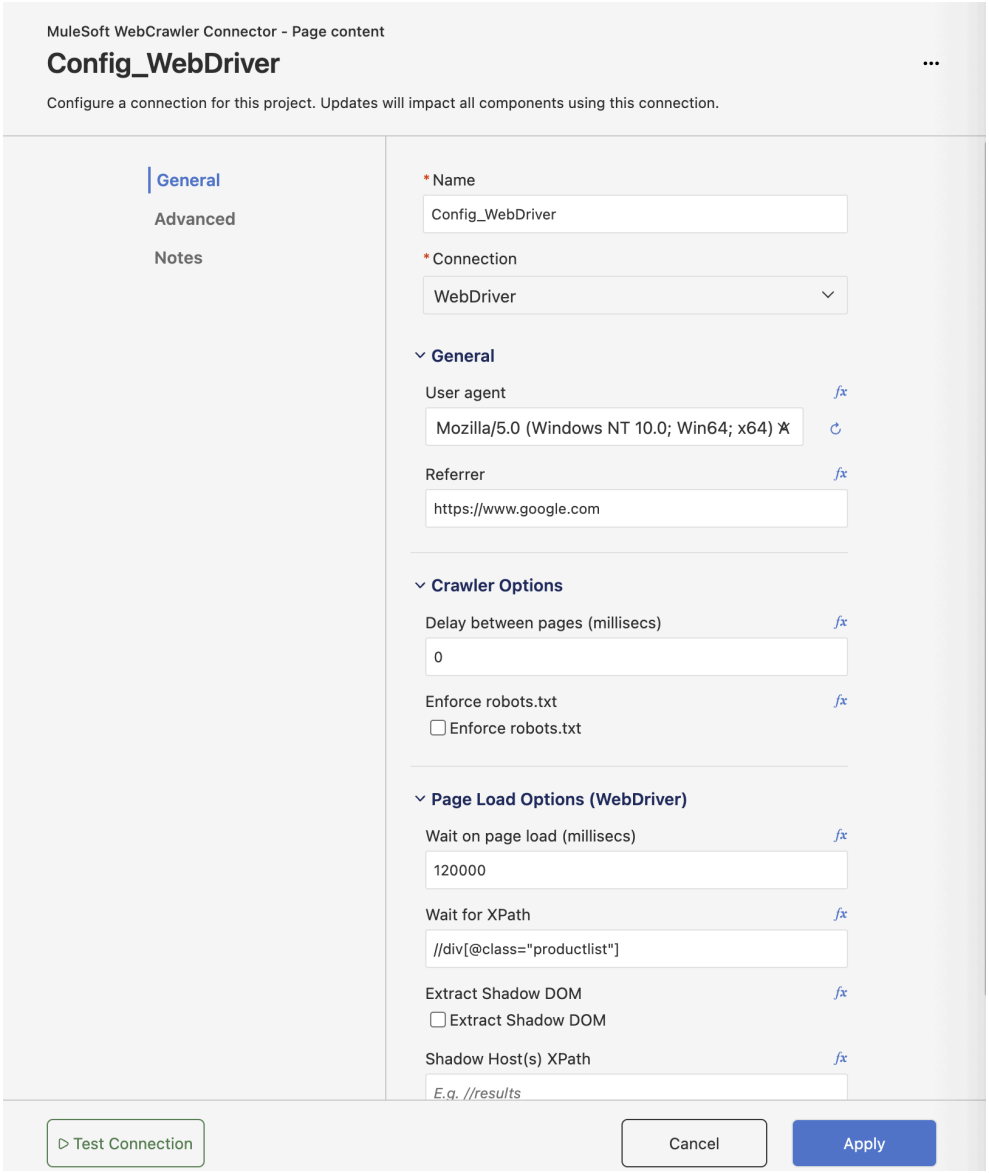
Step 4

Create WebCrawler Configuration (with WebDriver Connection). User Agent drop down only working with Anypoint Studio

* User Agent: `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36`

* Referrer `https://www.google.com`

* Set Wait on page load (millisec) to `120000`.
* Set Wait for xPath to `//div[@class="productlist"]`



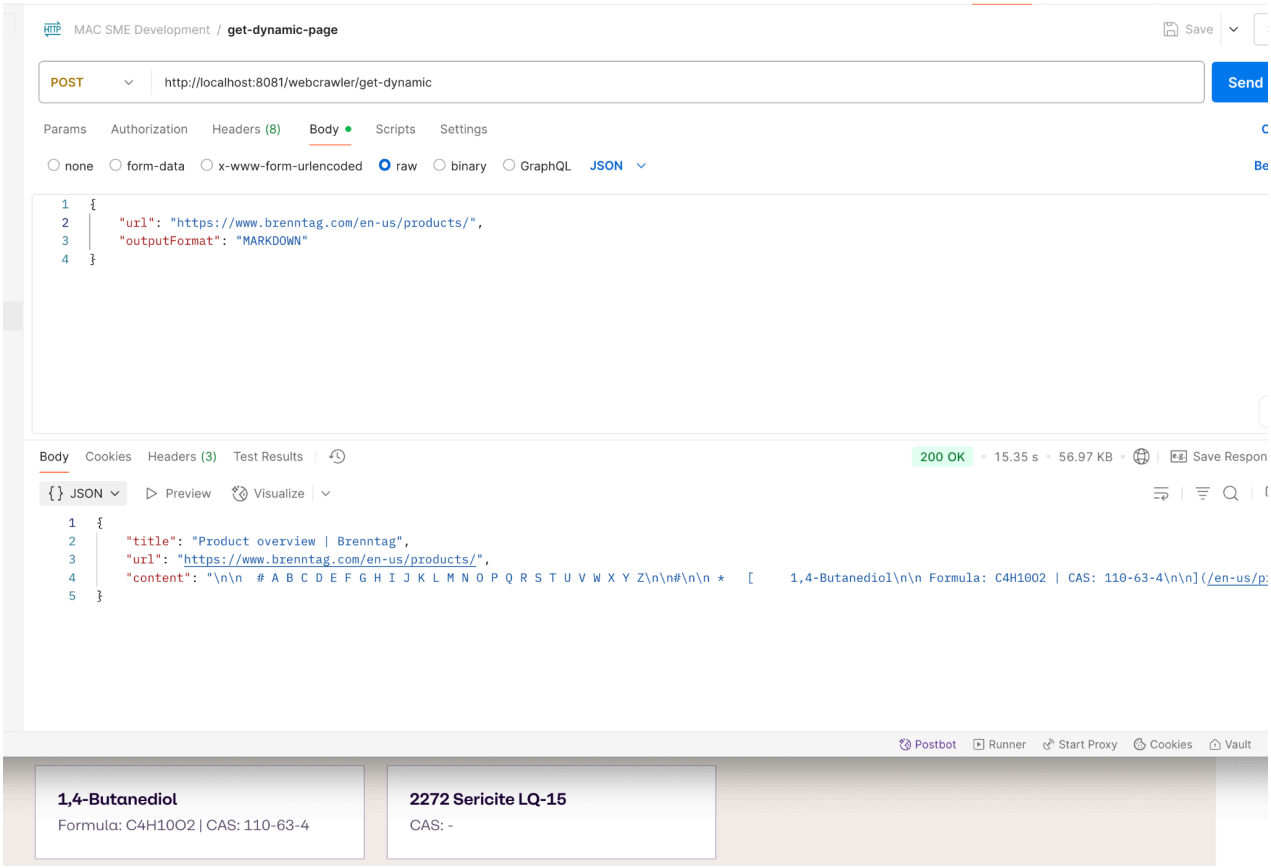
Step 5

Start you application
Hit http://localhost:8081/webcrawler/get-dynamic and check response.

Request payload:

```
{
"url": "https://www.brenntag.com/en-us/products",
"outputFormat": "MARKDOWN"
```

}

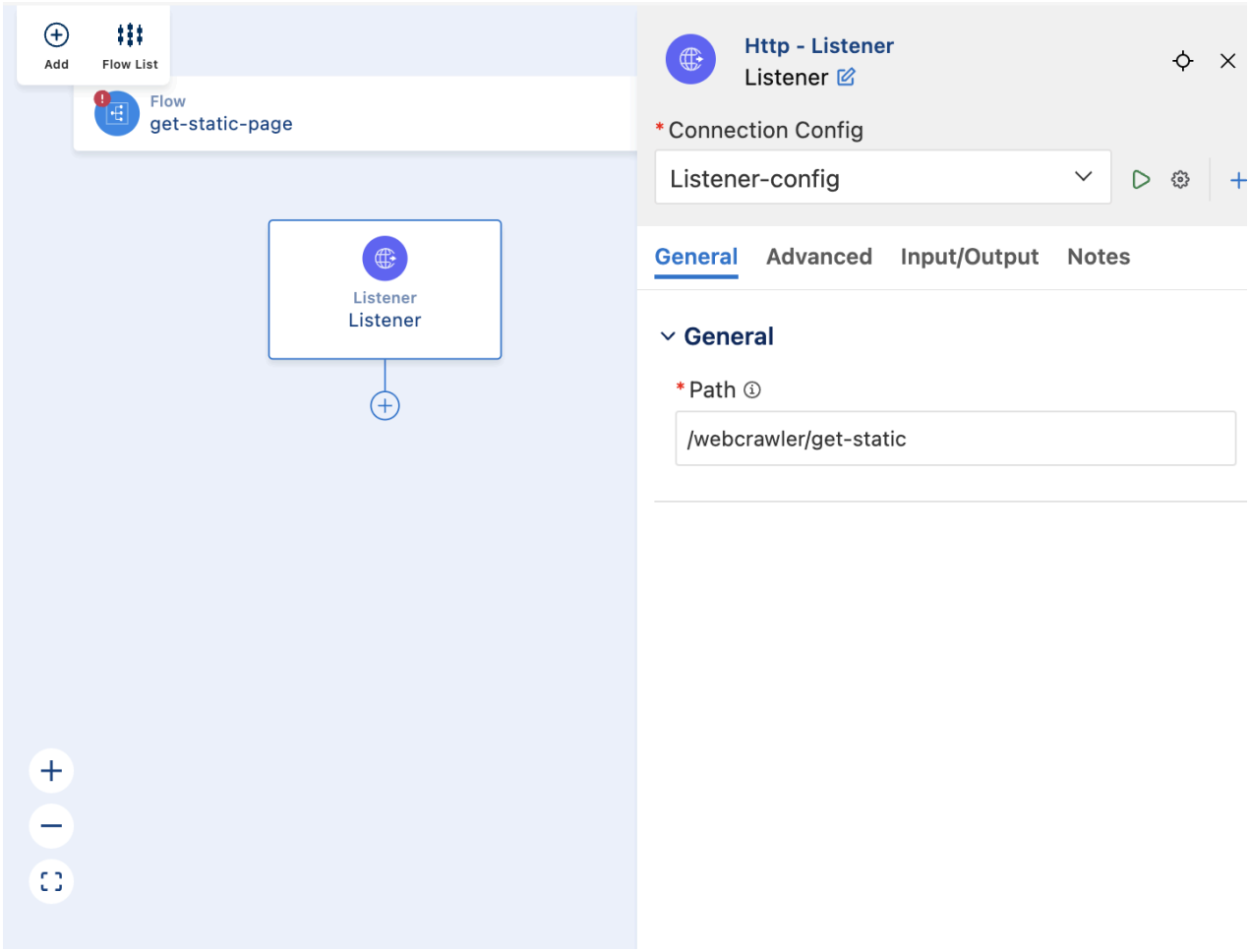


Exercise 3 - Crawl Website

Start from the exercise 2 app.

Step 1

Drop http listener (with path `/webcrawler/crawl`) in a new flow, you'll use it to download the site content.

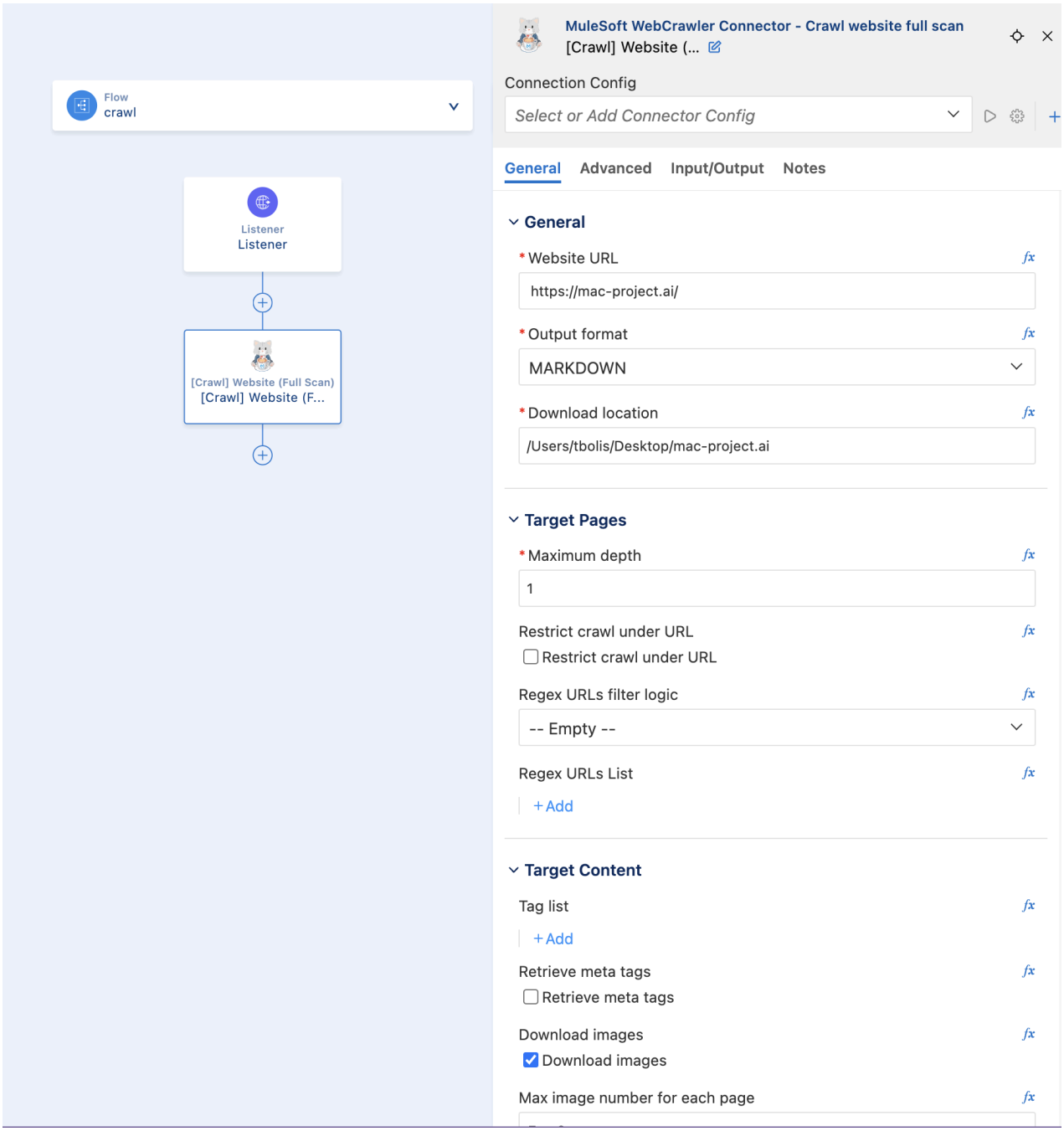


Step 2

Drop `[Crawl] Website (Full Scan)` operation after listener. Set: url, output format, download location, mac depth, etc…

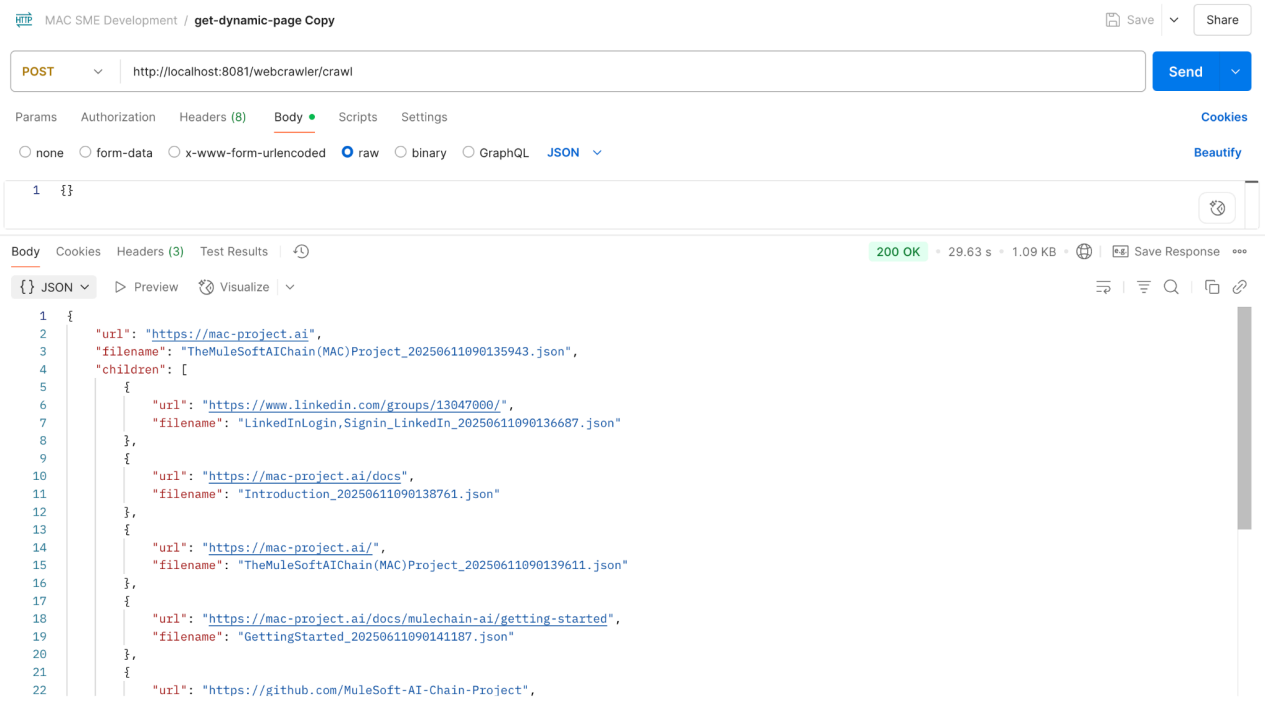
Use the configuration we created for exercise 1.

* url: https://mac-project.ai
* output format: `MARKDOWN`
* download location: `/Users/<your-user>/Desktop/mac-project.ai`
* depth: `1`
* Restric crawl under URL:false
* Download images: `true`



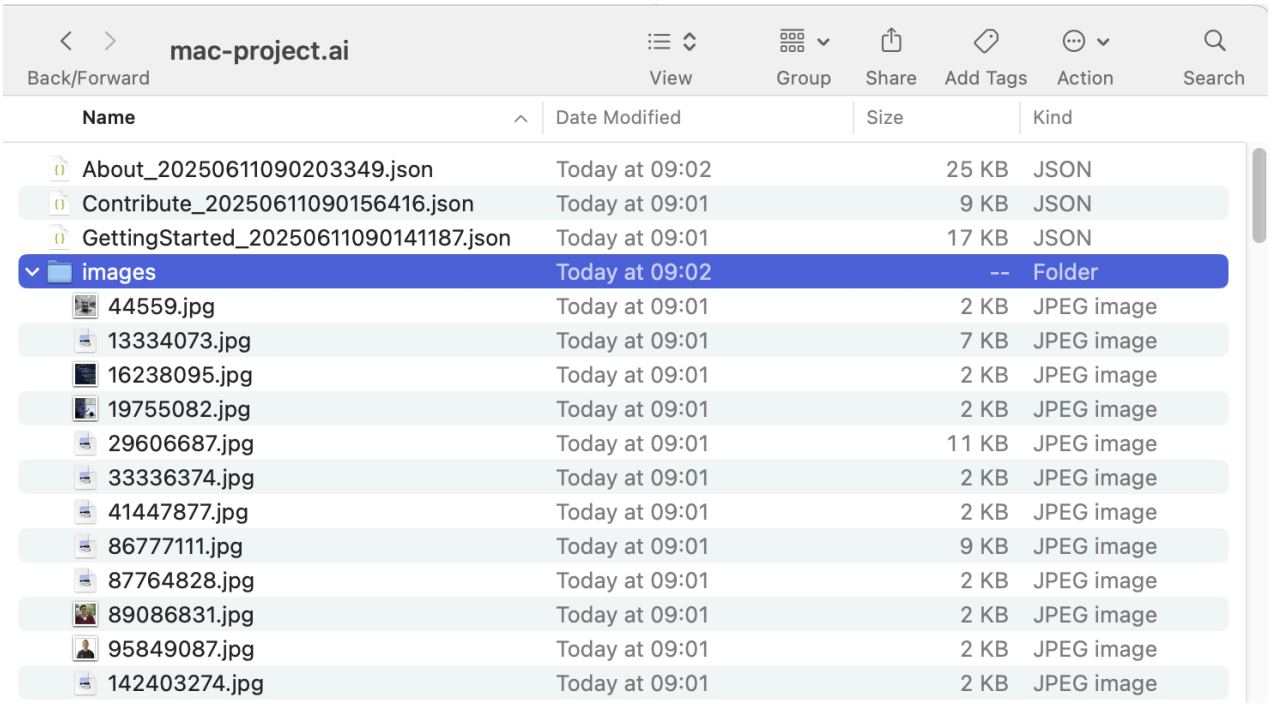
Step 3

Start you application
Hit http://localhost:8081/webcrawler/crawl.



Step 4

check download location.



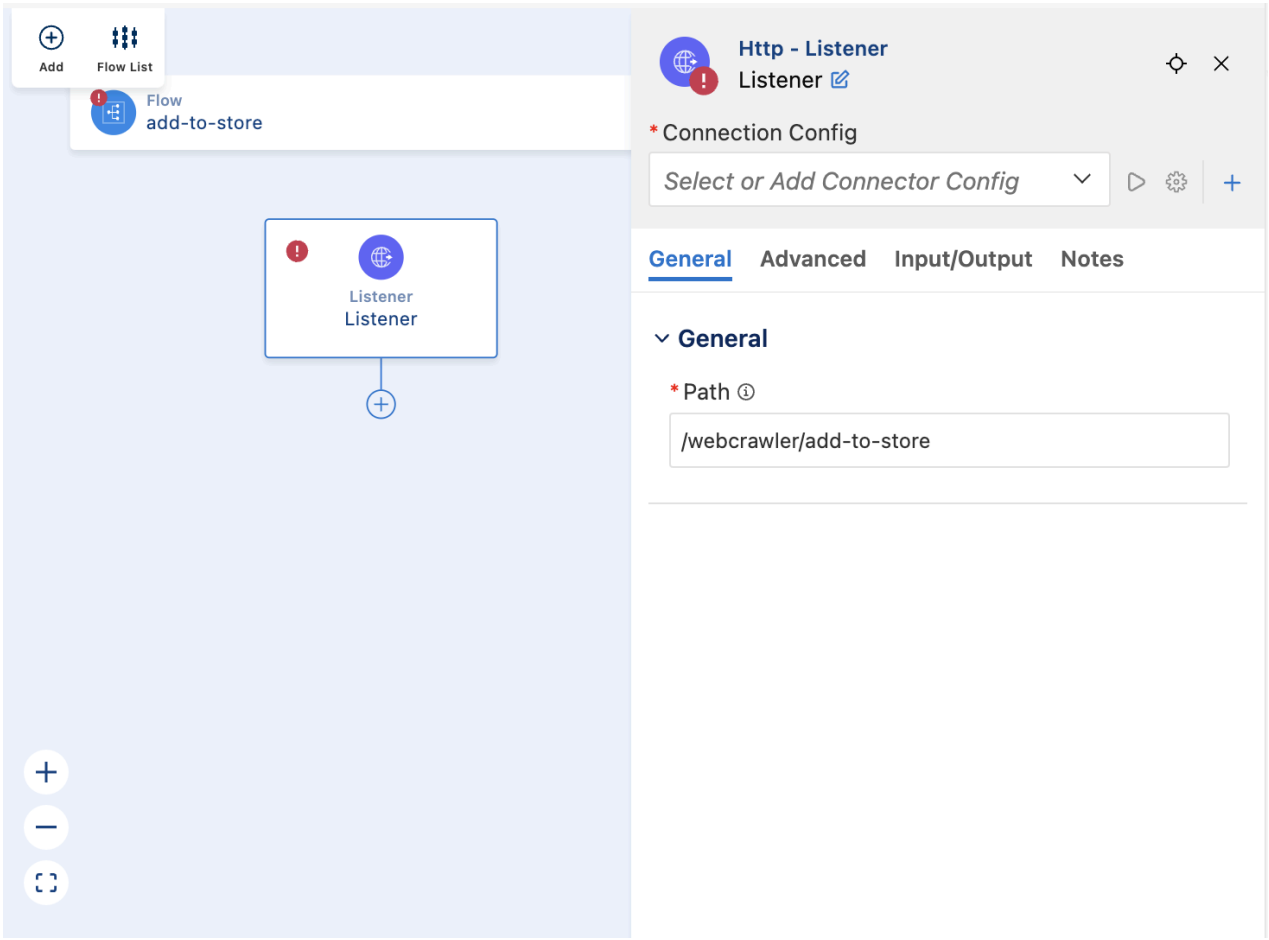
Exercise 4 - Crawl and ingest into Store

Prerequisites

- Setup SDO for Einstein Model APIs
- Setup PostgresSQL from SE Platform
- Vectors Hands-on

Step 1

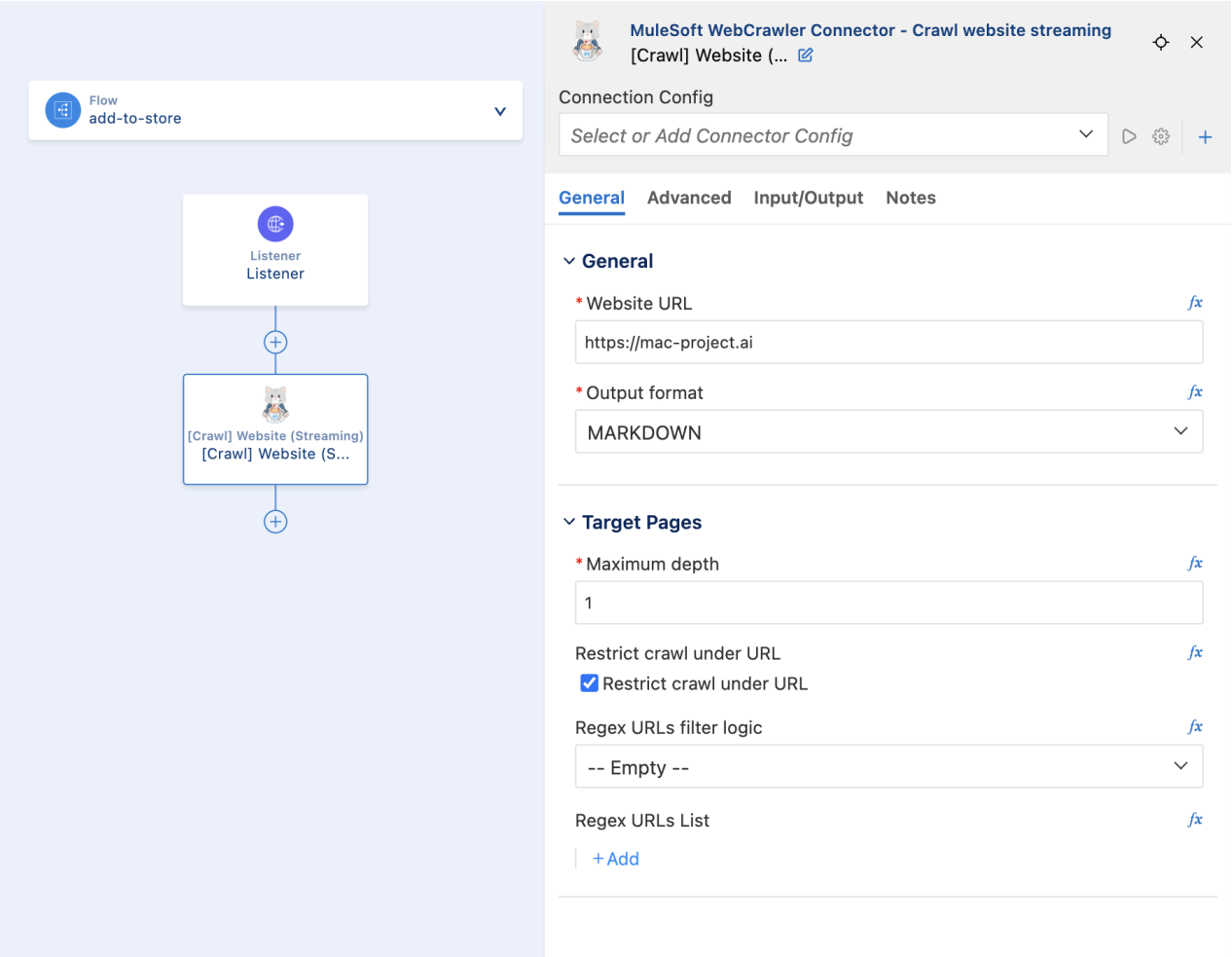
Drop http listener (/webcrawler/add-to-store) in a new flow, you'll use it load site content inside vector store.



Step 2

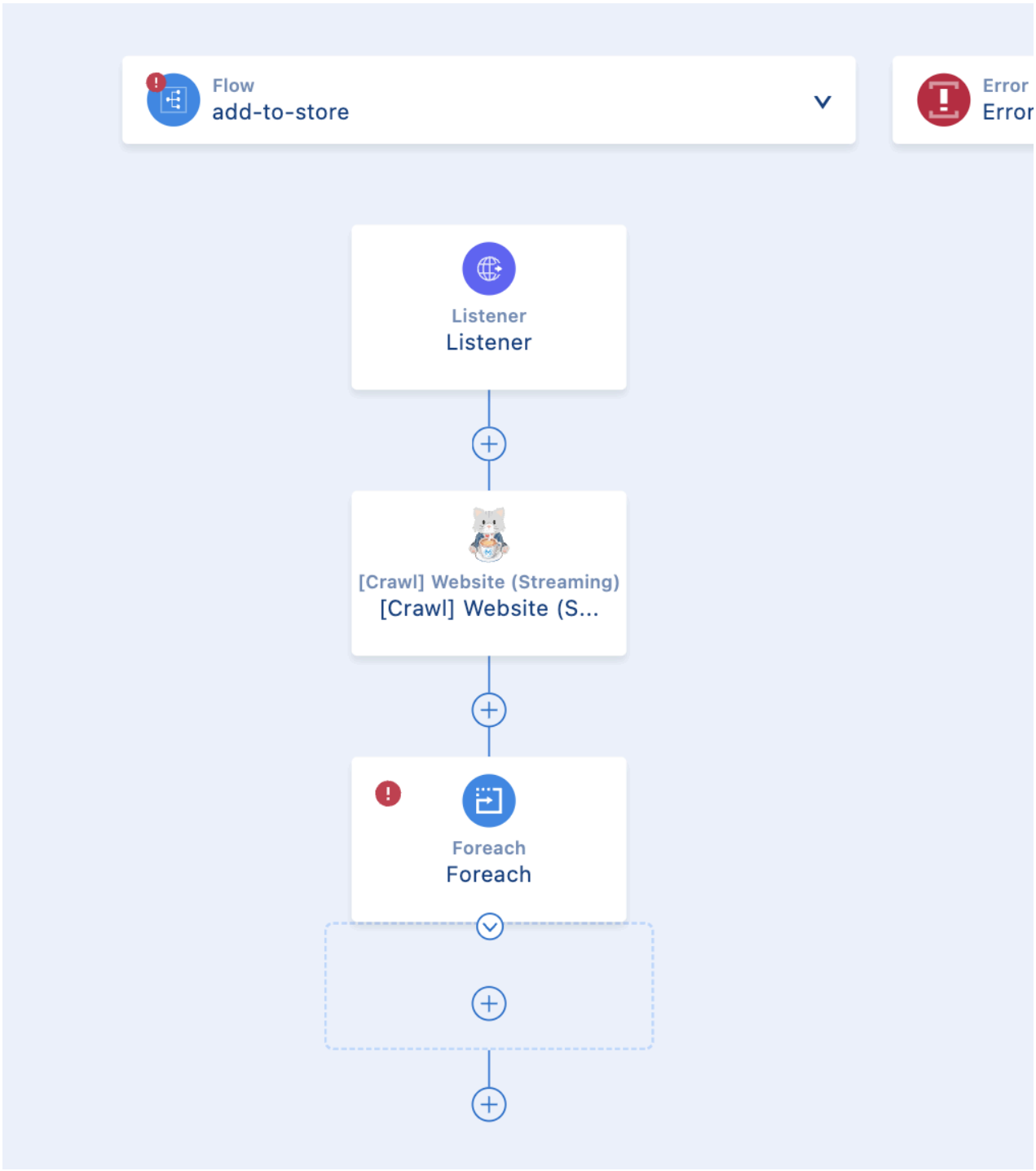
Drop [Crawl] Website (Streaming)

* url: https://mac-project.ai/
* output format: MARKDOWN
* download location: /Users/<your-user>/Desktop/mac-project.ai
* depth: 1
* Restrict crawl under URL: true



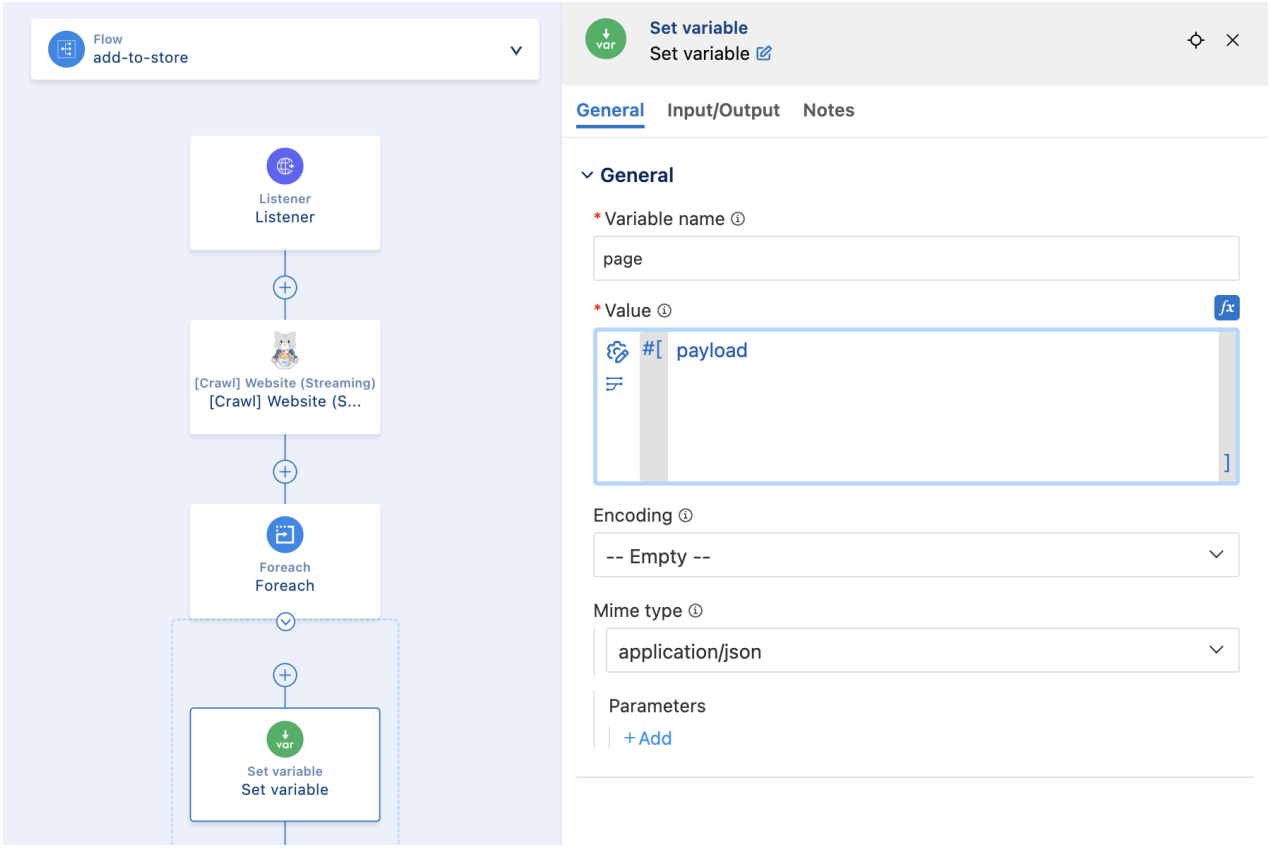
Step 3

Drop For Each



Step 4

Set payload into a page variable



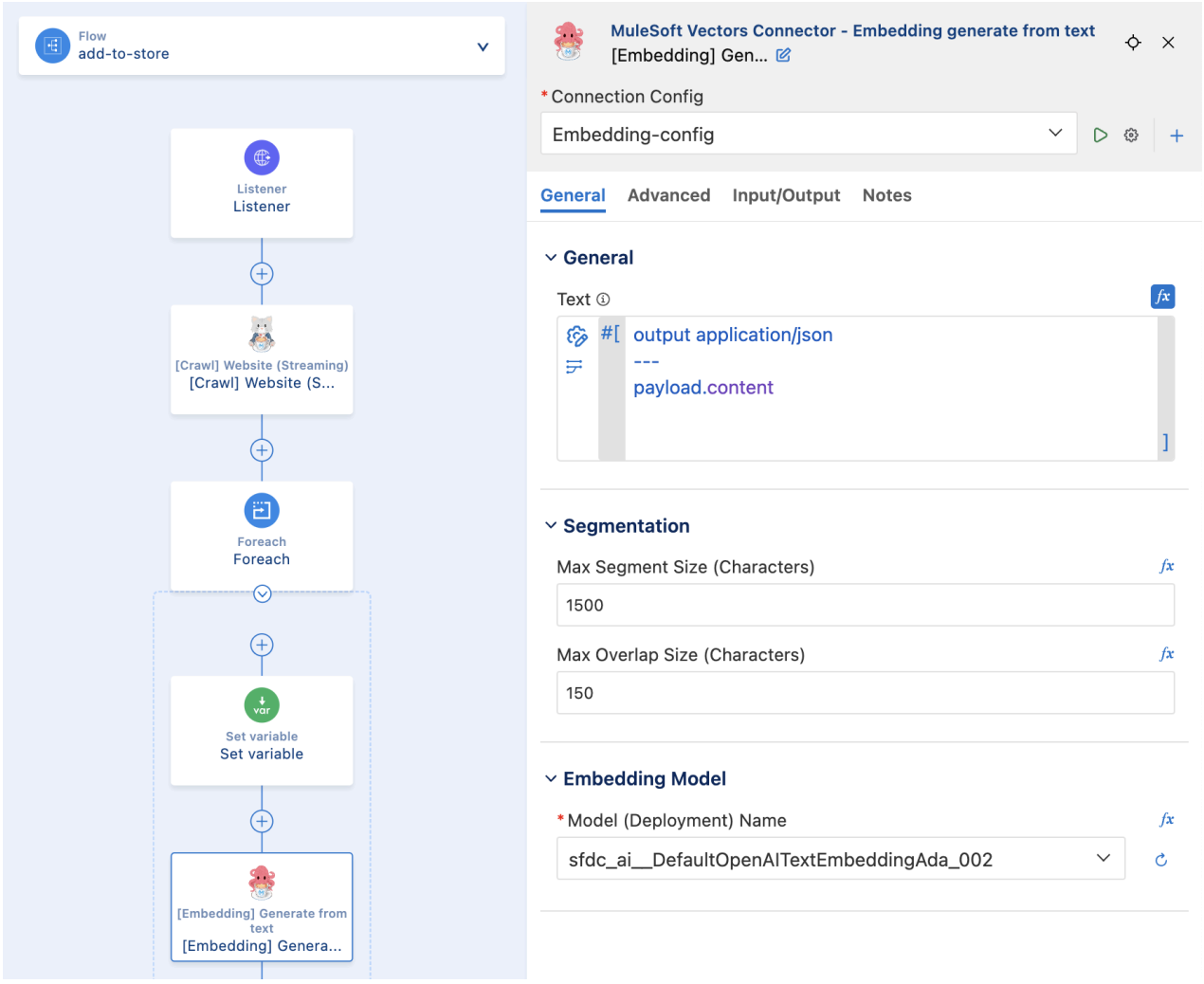
Step 5

Drop [Embedding] Generate from Text (inside for each)

Use configuration from vectors exercise

Set payload to:

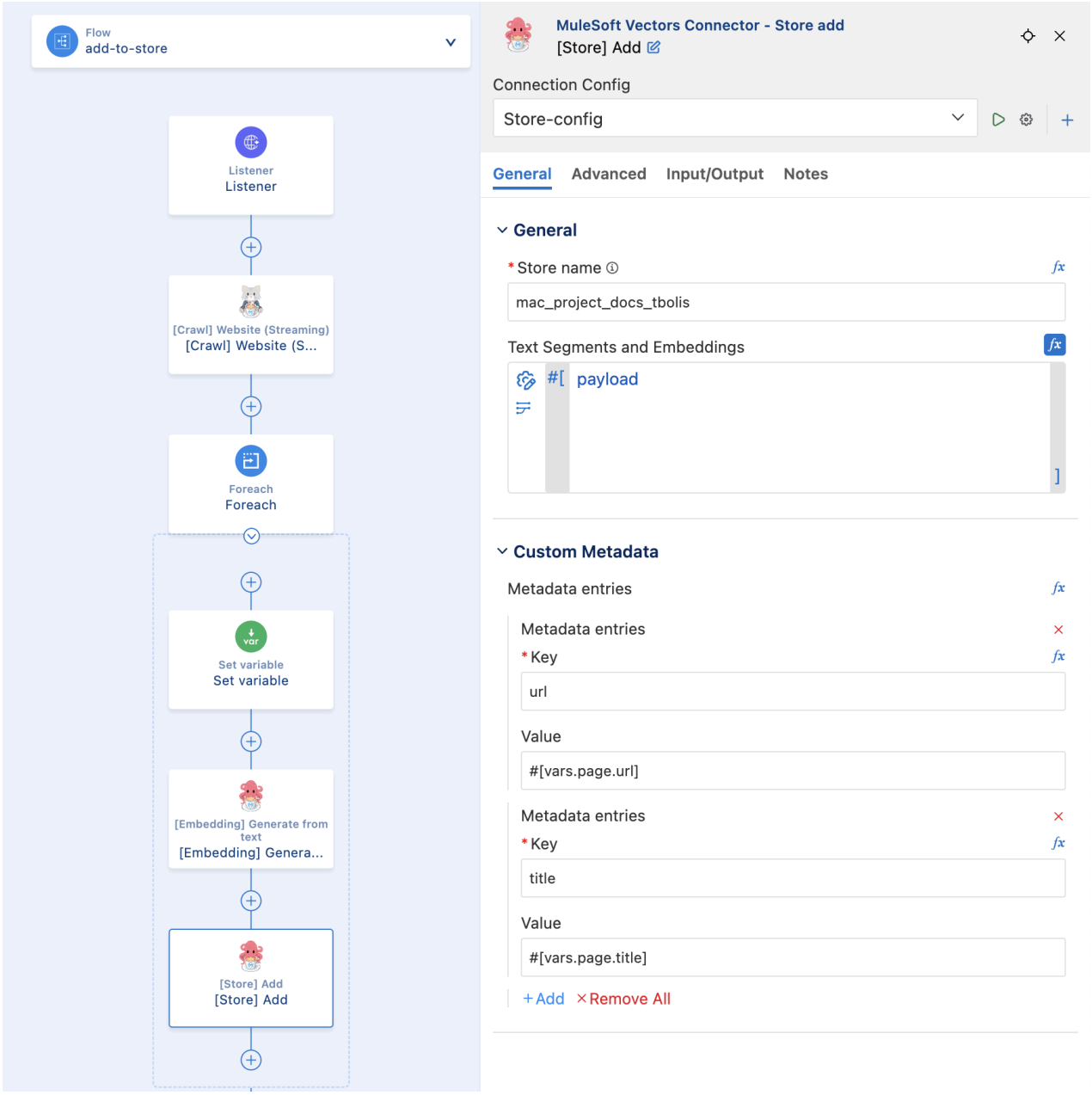
```
output application/json
---
payload.content
```



Step 6

Drop [Store] Add (inside for each).

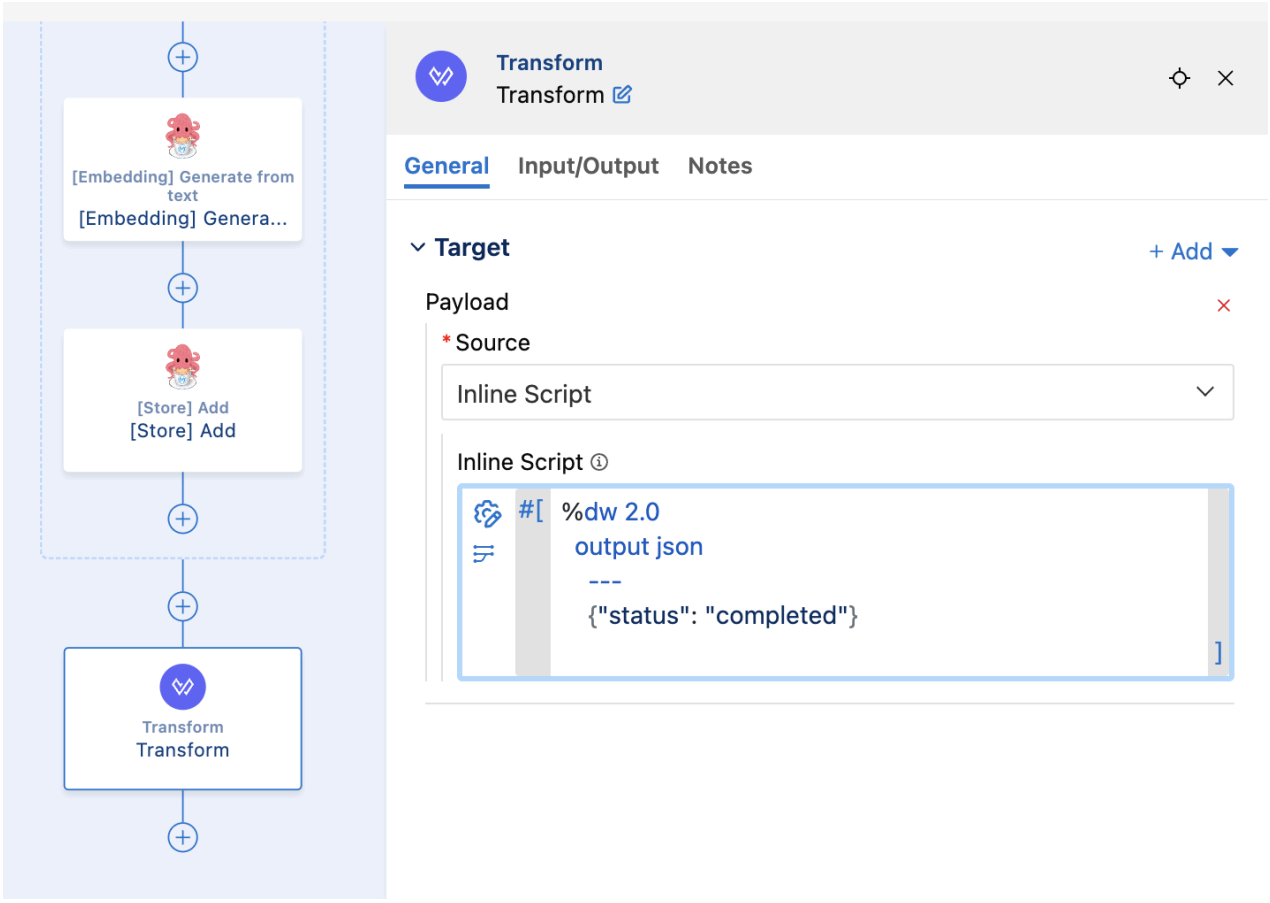
* storeName: `mac_project_docs_<your-user>`



Step 7

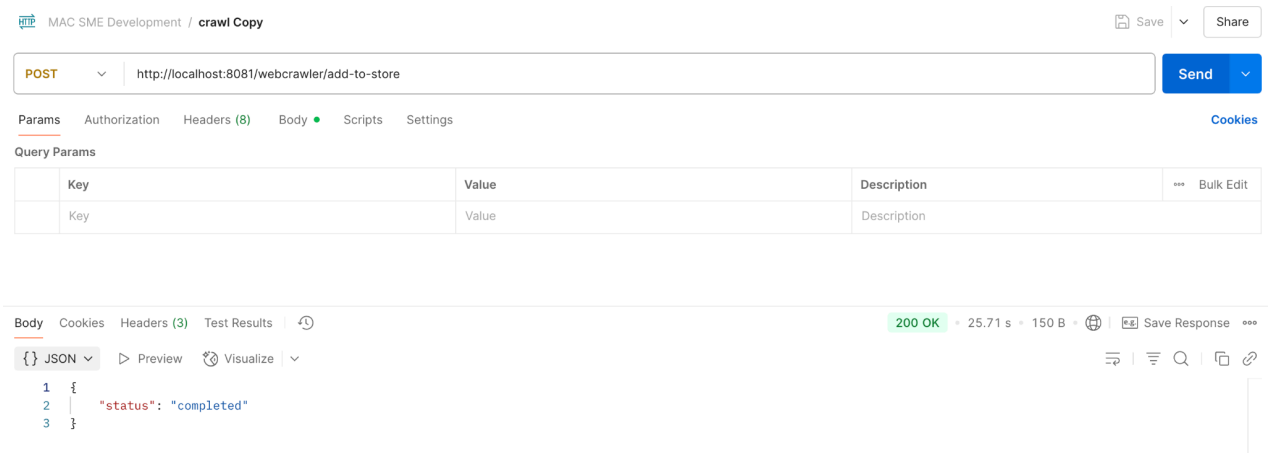
Drop a Trasform with:

```
%dw 2.0
output json
---
{"status": "completed"}
```



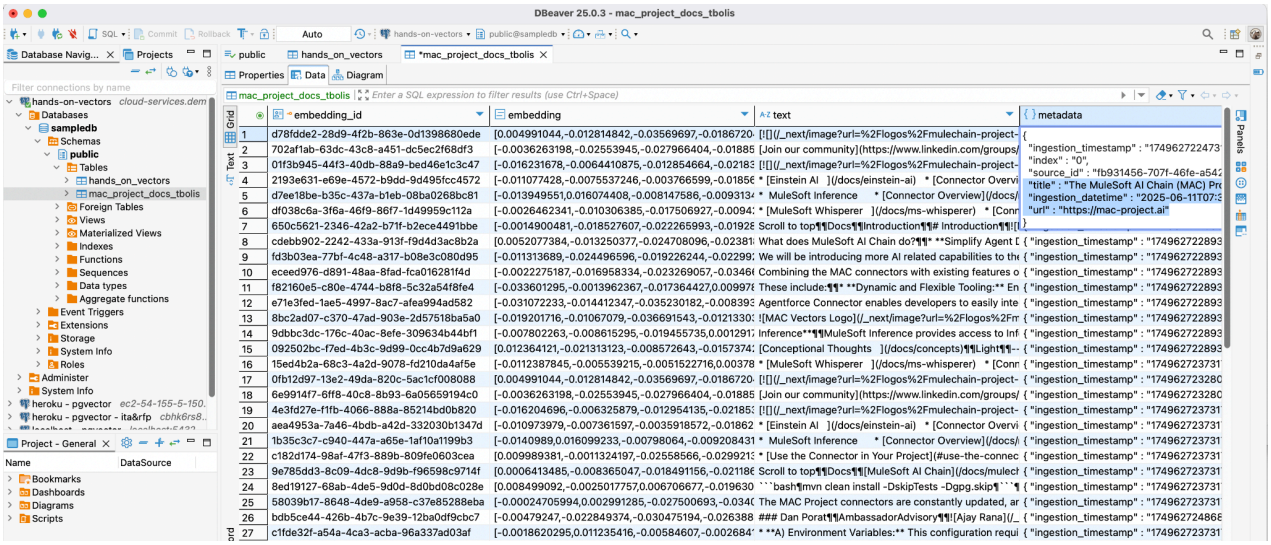
Step 8

Start you application
Hit http://localhost:8081/webcrawler/add-to-store and check the response.



Step 9

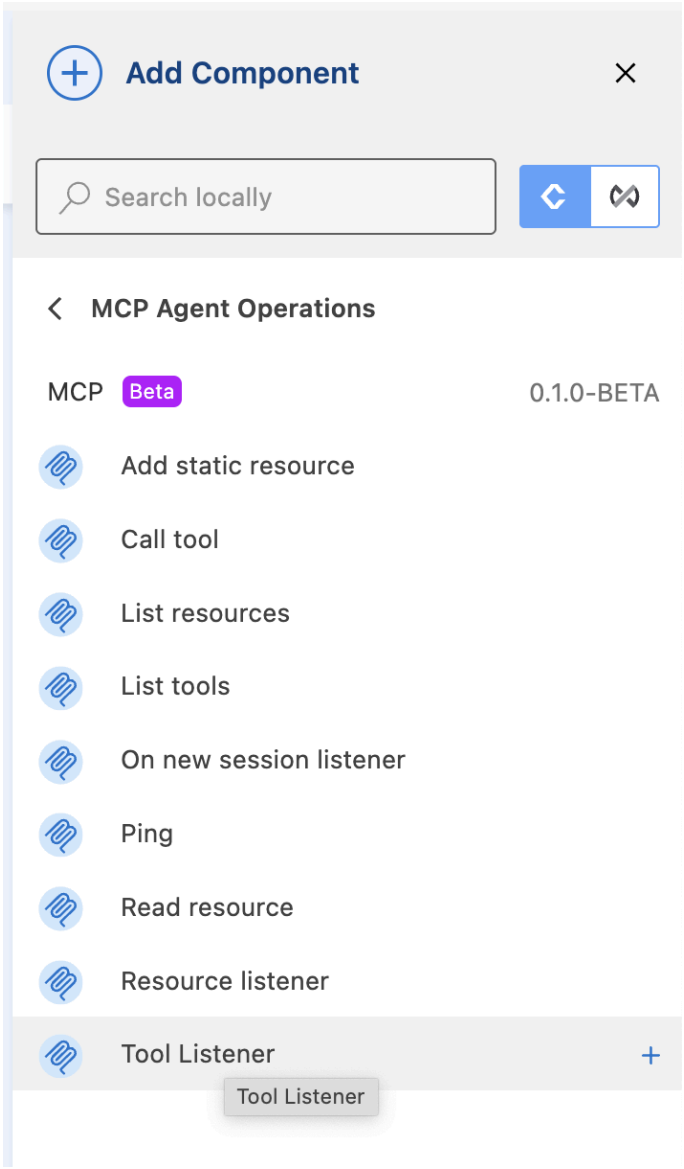
Check rows inserted into DB



Exercise 5 - Page Crawl as MCP Tool

Step 1

Add MCP Connector to your app taking it from exchange.



Step 2

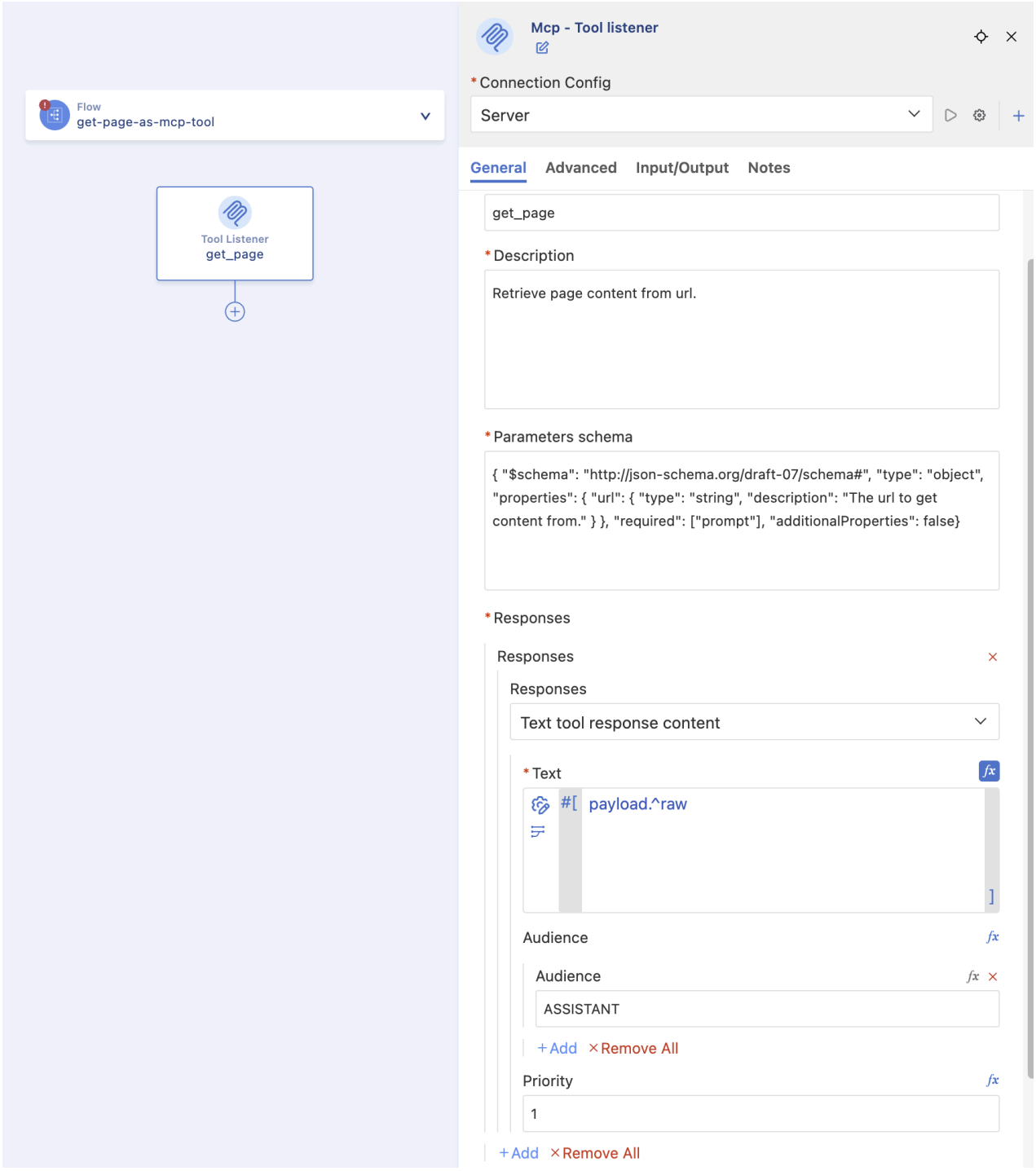
Drop MCP Tool Listener and configure it as follow:

- Name: `get_page`
- Description: `Retrieve page content from url.`
- Parameters schema:

```
{
"$schema": "http://json-schema.org/draft-07/schema#",
"type": "object",
```

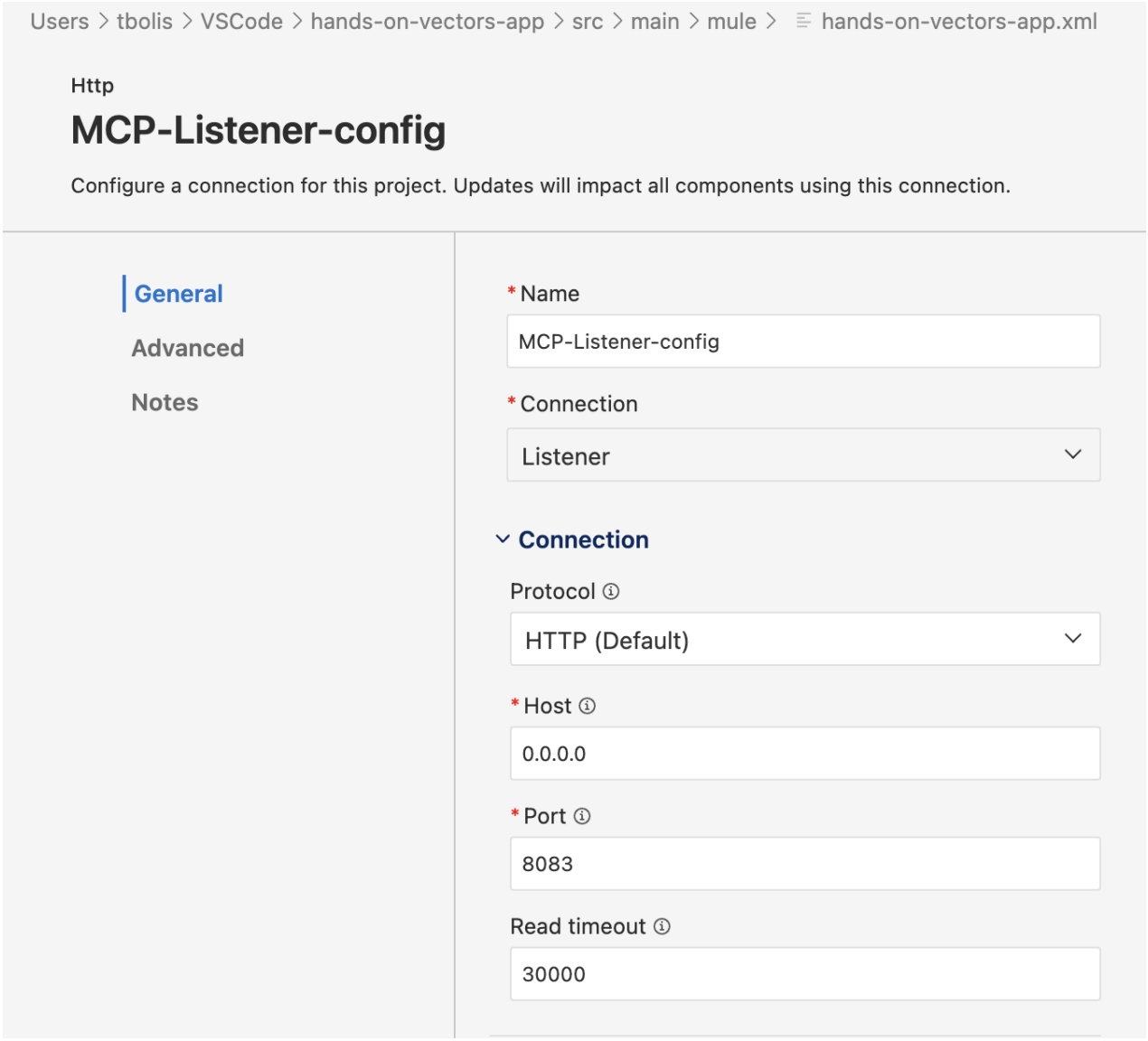
```
"properties": {
"url": {
"type": "string",
"description": "The url to get content from."
}
},
"required": ["url"],
"additionalProperties": false
}
```

- Responses:
- Text: `payload.^raw`
- Audience: `ASSISTANT`
- Priority: `1`



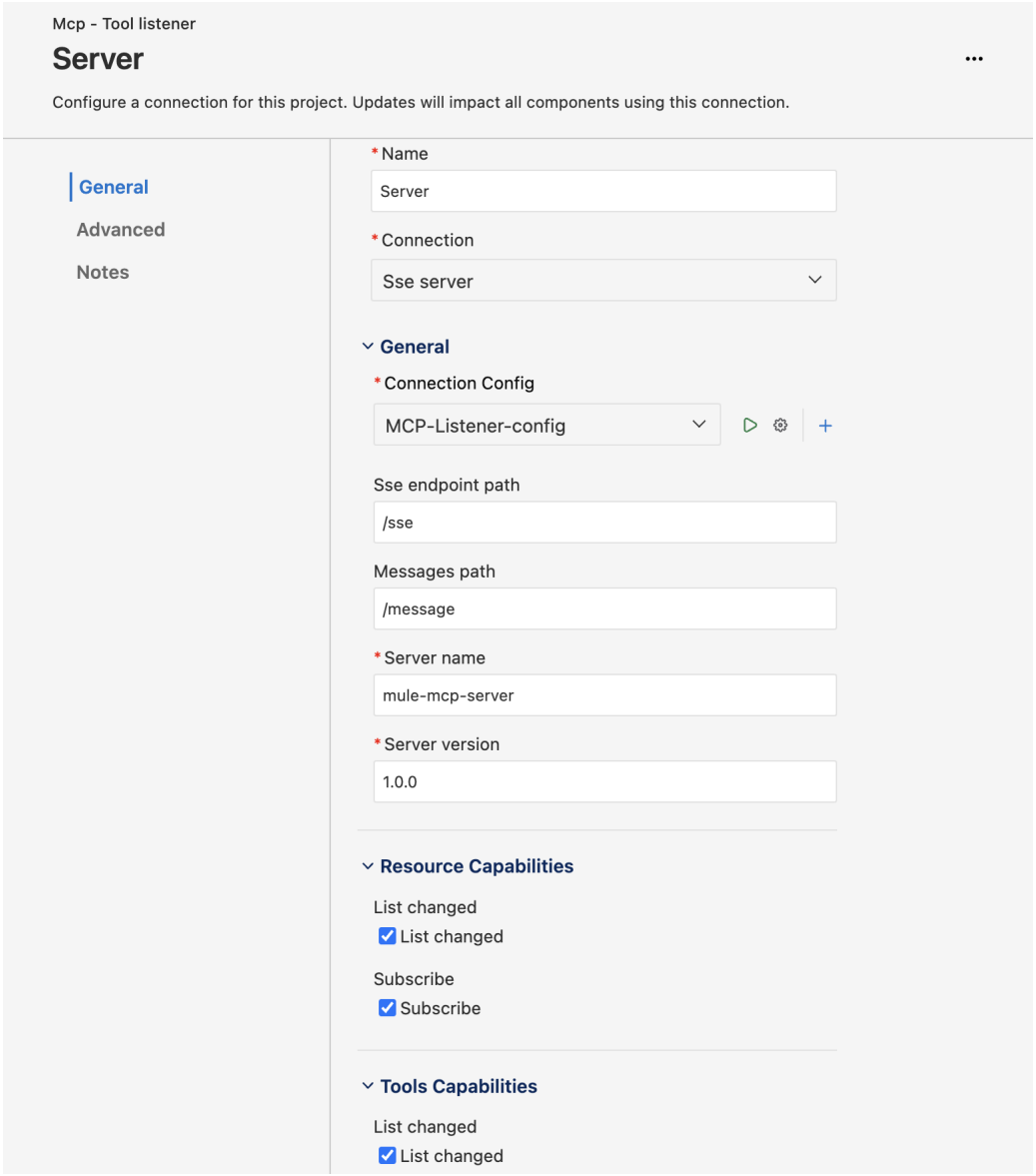
Step 3

Create a new HTTP Listener configuration using port 8083



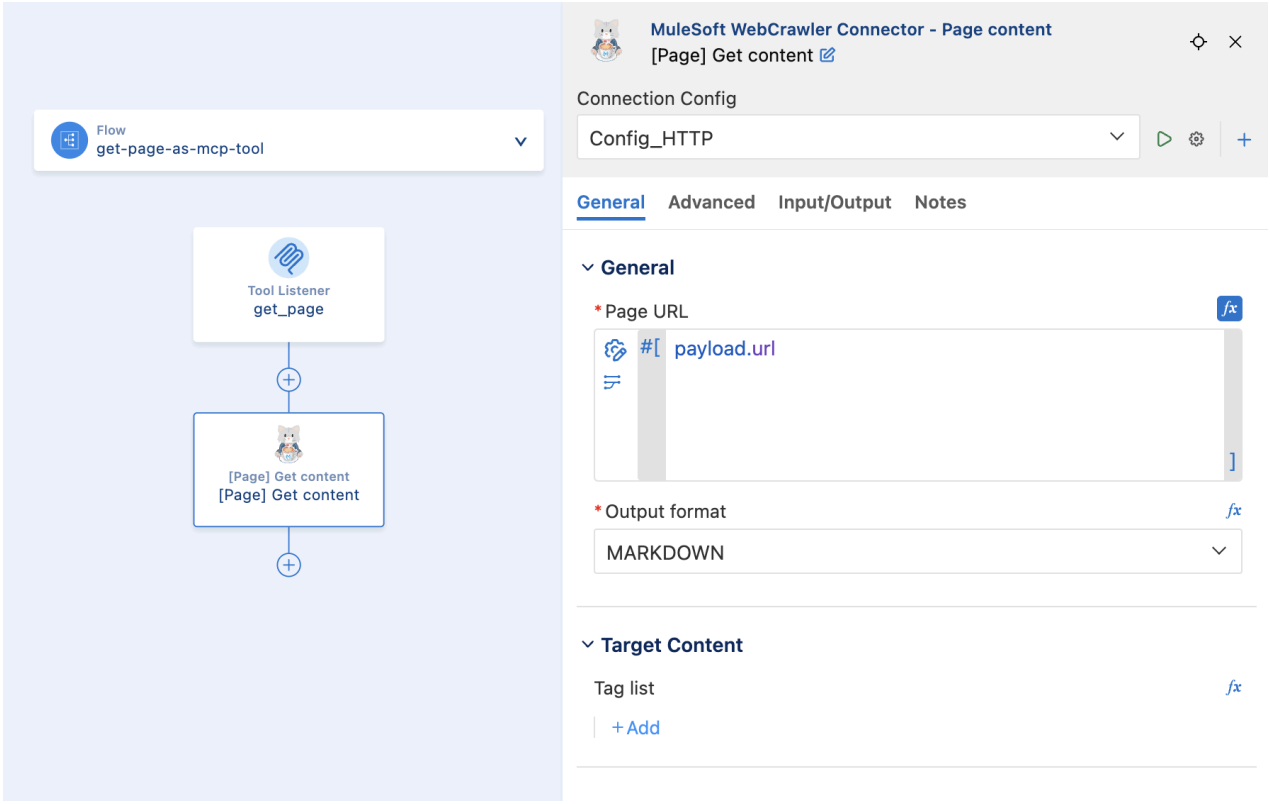
Step 4

Create a new MCP Server Configuration using Http Listener configuration created at previous step



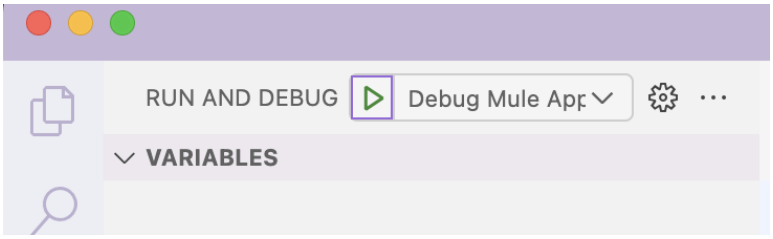
Step 5

After MCP Tool Listener add same logic implemented in exercise 1 to get page content in a static way.



Step 6

Start your application

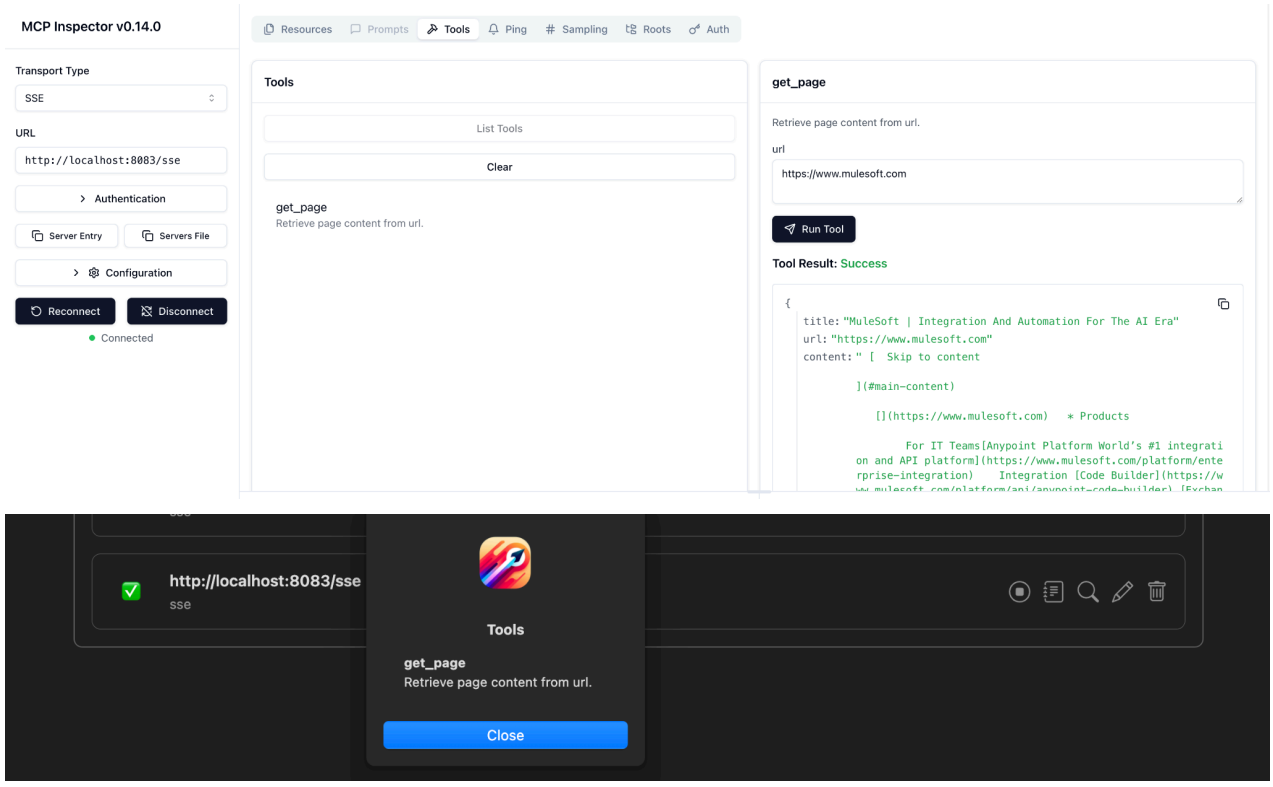


Step 7

Test your MCP Server with URL http://localhost:8083/sse using one client as Witsy or the MCP Inspector.

- MCP Inspector available at https://github.com/modelcontextprotocol/inspector

- Witsy available at https://witsyai.com/


MCP Inspector v0.14.0
Transport Type
SSE
URL
http://localhost:8083/sse
Authentication
Server Entry
Servers File
Configuration
Reconnect
Disconnect
Connected
Resources
Prompts
Tools
Ping
Sampling
Roots
Auth
Tools
List Tools
Clear
get_page
Retrieve page content from url.
get_page
Retrieve page content from url.
url
https://www.mulesoft.com
Run Tool
Tool Result: Success
{
title: "MuleSoft | Integration And Automation For The AI Era"
url: "https://www.mulesoft.com"
content: " [Skip to content
](#main-content)
[](https://www.mulesoft.com) * Products
For IT Teams[Anypoint Platform World's #1 integration and API platform](https://www.mulesoft.com/platform/enterprise-integration) Integration [Code Builder](https://www.mulesoft.com/platform/api/anypoint-code-builder) [Exchan

http://localhost:8083/sse
sse
Tools
get_page
Retrieve page content from url.
Close

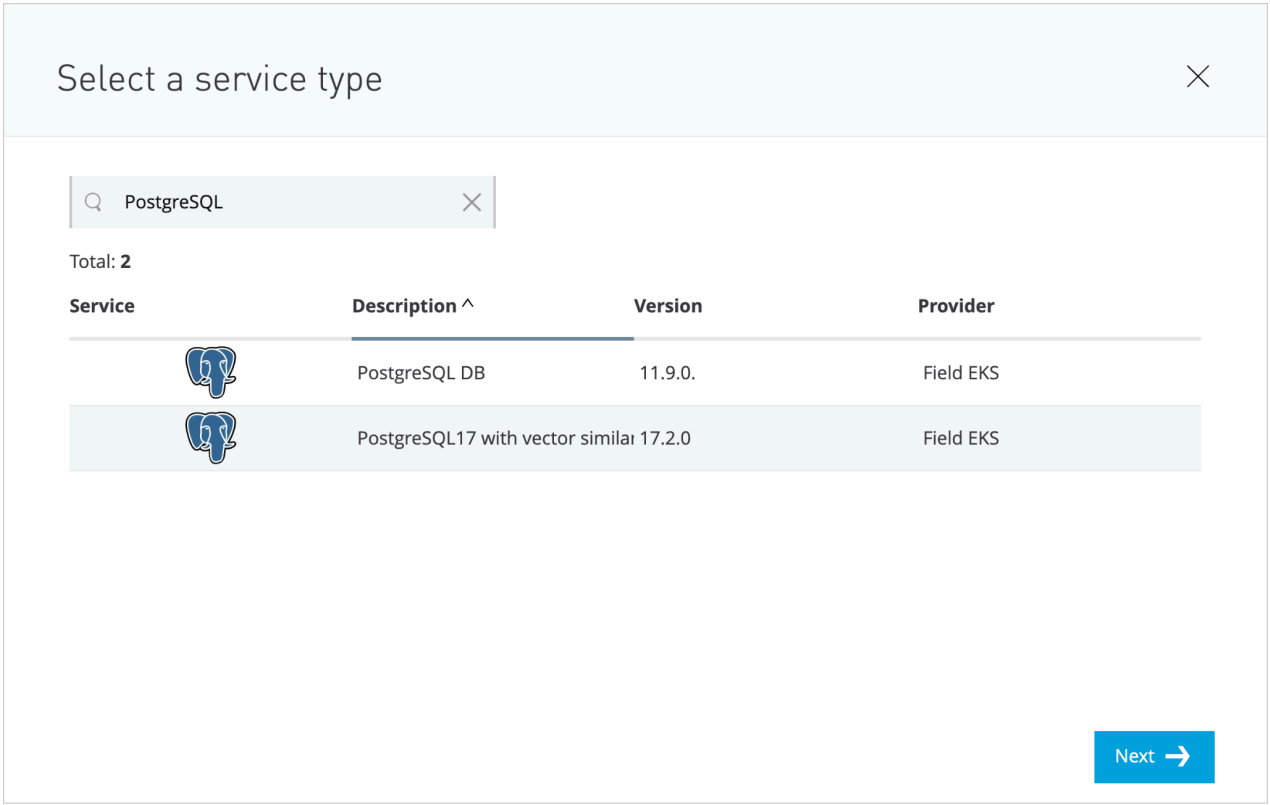
Pre-work

Setup PostgresSQL from SE Platform

```
#heroku-hands-on configurations
herokuhandson:
  host: cloud-services.demos.mulesoft.com
  port: '30389'
  username: postgres
  databaseName: sampledb
  password: TVK4pah_tze3dqe5qvx
  rootPassword: ''
```

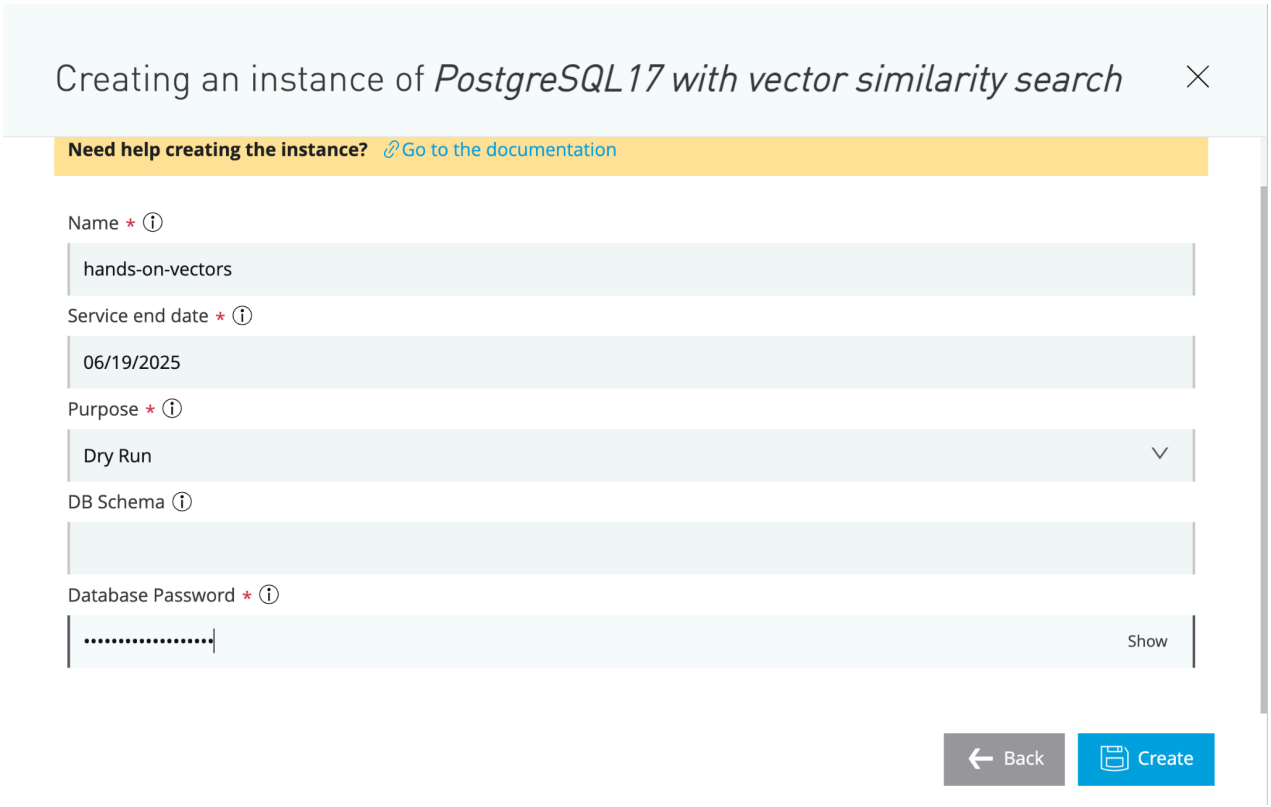
Step 1

On SE Platform create a new Cloud Service.
Select `PostgreSQL17 with vector similarity...`



Step 2

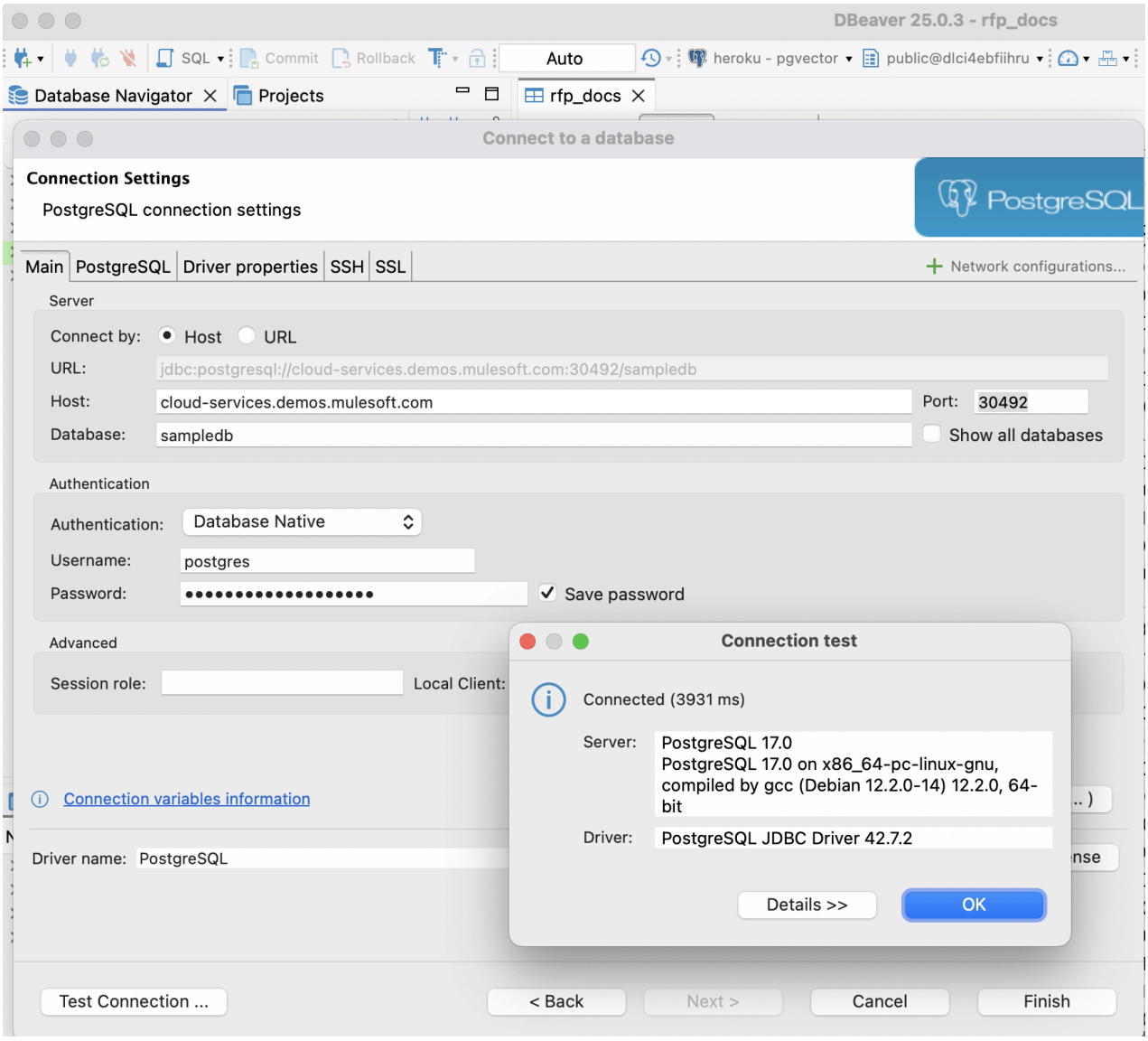
Enter required info



Step 3

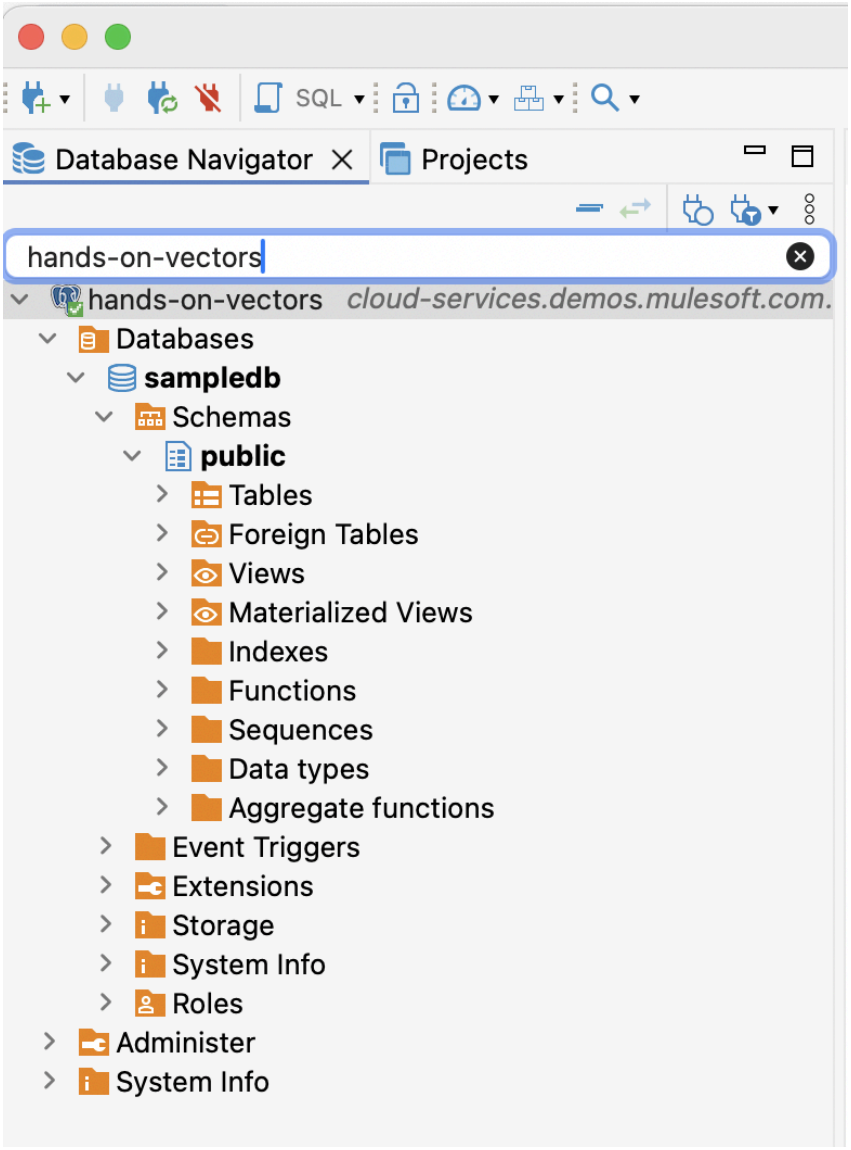
Verify connection to Store using DBeaver

https://dbeaver.io/download/



Step 4

Once connected check your are able to see database and schema.


SQL
Database Navigator
Projects
hands-on-vectors
hands-on-vectors cloud-services.demos.mulesoft.com.
Databases
sampledb
Schemas
public
Tables
Foreign Tables
Views
Materialized Views
Indexes
Functions
Sequences
Data types
Aggregate functions
Event Triggers
Extensions
Storage
System Info
Roles
Administer
System Info