Contents

[Question 1](#_Toc204087972)

[ChatGPT Answer 2](#_Toc204087973)

[Gemini Answer 3](#_Toc204087974)

[Q2 – Gemini 7](#_Toc204087975)

[Gemini Updated 7](#_Toc204087976)

[ChatGPT - how to build a **paginated API call flow using a single InvokeHTTP processor** 8](#_Toc204087977)

[ChatGPT final solution 1 10](#_Toc204087978)

[NiFi xml template by ChatGPT 14](#_Toc204087979)

## Question

Assume the role of a seasoned professional data engineer. Below is part of the homework of a "data at scale" course.

\*\*\* CASE STUDY REQUIREMENTS

1. aviationstack API allows you to get access to **aviation data** quickly so you can focus on building the features you need. The data is published through a REST API that returns a JSON-encoded response which contains information related to **real-time flight status, historical flights, schedules, routes, airports, and other aviation-specific data.**

2. To use any aviationstack API endpoint, you must pass an API token with each request. If you do not include your API token when making an API request, or use one that is incorrect or disabled, aviationstack returns an error.

3. To obtain or generate your API Key Token you need to register (for free) on their website. https://aviationstack.com/signup/free For this study, you need to choose the free plan for individuals which offer 100 Requests / mo. Which is enough for the current study

Based on the CASE STUDY REQUIREMENTS listed above, please provide a simplified version with clear wording and specific activities included to carry on the subtasks included in the below requirements (Use clear simple wording with details, clarification, and hints)

\*\*\* Requirement READING JSON FLIGHTS DATA (5 pts)

• Build an Apache NiFi flow to ingest flights limited to YUL using the REST API.

• Implement pagination using a loop to increment the offset (do not clone InvokeHTTP

processors).

Limit (in the loop) each API request to 100 rows. The free plan is limited to 100 rows per call.

• Avoid hardcoding sensitive or variable information.

• Use saved JSON files during development and testing.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Please feel free to ask me questions-one question at a time if required

## ChatGPT Answer

Great — let's walk through how a seasoned data engineer would *simplify and clarify* the task in plain, clear wording, focusing on the **“READING JSON FLIGHTS DATA”** part of the case study. This version will include explanations, hints, and specific activities you need to perform.

**🔍 Simplified Task: Read Flights Data from the Aviationstack API Using NiFi**

**🎯 Goal**

Use Apache NiFi to automatically pull flight data related to **YUL (Montreal-Trudeau airport)** from the Aviationstack API, handle multiple pages of results, and save the results locally in JSON files.

**✅ Step-by-Step Breakdown**

**1. Get Access to the API (One-Time Setup)**

**What to do:**

Go to: <https://aviationstack.com/signup/free>

Create a free account.

After signing up, copy your **API Key** (you'll need it for NiFi to access the API).

**2. Understand the API URL Format**

**What to know:**

The API gives flight data through this endpoint (as an example):

http://api.aviationstack.com/v1/flights?access\_key=YOUR\_API\_KEY&dep\_iata=YUL&limit=100&offset=0

**Important parts explained:**

access\_key=YOUR\_API\_KEY: This is your secret token (don't hardcode it in NiFi — we’ll make it a parameter).

dep\_iata=YUL: Filters for departures from Montreal (YUL).

limit=100: Maximum number of results per API call (the free plan allows this).

offset=0: Used for pagination. Set it to 0 for the first 100 rows, 100 for the next page, and so on.

**3. Use Apache NiFi to Pull the Data**

**🛠️ NiFi Flow Design Overview**

You will build a **looping flow** in NiFi that:

* Starts with a signal to begin pagination.
* Builds the URL using flow variables (not hardcoded).
* Calls the API using InvokeHTTP.
* Checks if the response contains data.
* If data exists, saves it to a JSON file and loops to get the next page.
* Stops when no more data is returned.

**🧩 Key Processors You’ll Use**

| **Processor** | **What it does** |
| --- | --- |
| **GenerateFlowFile** | Triggers the flow – start of the loop |
| **UpdateAttribute** | Set flow variables (like offset, limit, API key, base URL) |
| **InvokeHTTP** | Calls the Aviationstack API |
| **EvaluateJsonPath** | Check if the response has flights data |
| **RouteOnAttribute** | Decide whether to continue or stop the loop |
| **PutFile** | Save each page of JSON data locally |
| **ReplaceText** (optional) | Build dynamic URL with variables |

**🧠 Hints & Tips**

**Use Parameters or Variables for:**

API\_KEY, base\_url, offset, limit

Avoid hardcoding the API key. Use NiFi **Parameter Contexts** or flow variables.

**Loop Logic:**

Start with offset = 0.

After each successful request, increase offset by 100 (use UpdateAttribute to update it).

Use EvaluateJsonPath to check if "data" array in response is empty or not.

If not empty → loop again.

If empty → stop.

**Use Saved JSON During Testing**

Before you run the full API flow, you can **manually download a sample JSON** from the API and save it locally.

Then use GetFile → EvaluateJsonPath → PutFile just to test your downstream processors like filtering, routing, etc.

**📁 Output**

Each API page response (up to 100 rows) should be saved in a separate JSON file in a local folder.

Example filenames:

flights\_yul\_page\_0.json

flights\_yul\_page\_100.json

flights\_yul\_page\_200.json

...

Would you like me to sketch the NiFi flow structure as a visual step diagram next?

## Gemini Answer

As a seasoned data engineer, I'll guide you through this NiFi task. The goal is to pull flight data for Montreal (YUL) from Aviationstack, handle the pagination, and do it smartly without hardcoding sensitive info.

Here's a simplified version of your "READING JSON FLIGHTS DATA" requirement with clear steps and hints:

**Simplified Guide: Ingesting YUL Flight Data with Apache NiFi**

**Overall Goal:** Your mission is to build an Apache NiFi data pipeline that fetches flight data specifically for Montreal-Pierre Elliott Trudeau International Airport (YUL) from the Aviationstack API. You'll need to handle how the API provides data in chunks (pagination) and ensure your setup is clean and secure.

**Before You Start (Prerequisite):**

* **Your Aviationstack API Key:** Make sure you've registered on the [aviationstack.com/signup/free](https://aviationstack.com/signup/free) website and have your unique API access key ready. You'll need this for every API request.

**Step-by-Step NiFi Flow Construction:**

**Sub-task 1: Setting Up Your Environment (Avoiding Hardcoding)**

1. **~~Create a NiFi Controller Service for Your API Key:~~**
   * **~~Activity:~~** ~~In your NiFi canvas, go to the "Controller Services" icon (the gear in the top right).~~
   * **~~Activity:~~** ~~Add a new~~ **~~Standard Parameter Context~~** ~~or~~ **~~StandardSensitivePropertiesService~~**~~. For an API key,~~ **~~StandardSensitivePropertiesService~~** ~~is excellent.~~
   * **~~Details:~~** ~~Add a new property, for example, aviationstack.api.key. Input your actual API key here.~~
   * **~~Hint:~~** ~~This is crucial! Using a Controller Service or Parameter Context means your API key isn't visible in the processor's configuration, making your flow more secure and easier to manage if the key ever changes. You'll reference this key in your InvokeHTTP processor.~~

**Sub-task 2: Making the Initial API Request (YUL Flights)**

1. **Start the Flow with a Generator:**
   * **Activity:** Drag a **GenerateFlowFile** processor onto your canvas.
   * **Details:** Configure it to run once (e.g., Scheduling: Run Schedule = 0 sec, Execution Strategy = On primary node, Batch Size = 1). We just need an initial "trigger" to start fetching data.
   * **Hint:** This processor simply creates an empty FlowFile, which acts as the signal to start your API call.
2. **Call the Aviationstack API for YUL Flights:**
   * **Activity:** Connect **GenerateFlowFile** to an **InvokeHTTP** processor.
   * **Details:**
     + **Remote URL:** This is where you'll build your API request. For YUL, you'll use an endpoint like http://api.aviationstack.com/v1/flights.
     + **Query Parameters:** You'll need to add parameters for access\_key, flight\_status, airport\_iata, and limit.
       - access\_key=${aviationstack.api.key} (This is how you reference the key from your Controller Service/Parameter Context!)
       - flight\_status=active (or scheduled, landed, etc., depending on what you need, but 'active' is a good start for "flights")
       - airport\_iata=YUL
       - limit=100 (This is your maximum rows per call, as per the free plan).
     + **HTTP Method:** GET
     + **Attribute that Contains the Output:** response.body (This is where the JSON response will be stored).
   * **Hint:** Carefully construct your URL and query parameters. Test this first to ensure you get a successful response before moving to pagination.

**Sub-task 3: Implementing Pagination (The Loop)**

This is the trickiest part, but we'll manage it by using NiFi's ability to modify FlowFile attributes and make conditional routing.

1. **Initialize Offset for Pagination:**
   * **Activity:** Connect the success relationship of your first **InvokeHTTP** to an **UpdateAttribute** processor.
   * **Details:** Add a new property: offset = 0. This attribute will keep track of how many records we've already fetched.
   * **Hint:** We're setting the starting point for our loop here.
2. **Extract Pagination Information (If available):**
   * **Activity:** Connect the success relationship of **UpdateAttribute** to an **EvaluateJsonPath** processor.
   * **Details:**
     + **Destination:** flowfile-attribute
     + Add properties to extract relevant pagination info from the JSON response. Aviationstack's JSON structure typically has pagination block.
       - total\_records = $.pagination.total
       - current\_page\_offset = $.pagination.offset (This is the offset of the *current* request)
       - current\_page\_limit = $.pagination.limit (This is the limit used in the *current* request)
   * **Hint:** Inspect the JSON response from your first successful API call. Use tools like jq or online JSON formatters to understand the structure and extract the correct paths.
3. **Prepare for the Next Iteration (Incrementing Offset):**
   * **Activity:** Connect the success relationship of **EvaluateJsonPath** to another **UpdateAttribute** processor.
   * **Details:**
     + Add a new property: next\_offset = ${offset:plus(100)} (This is crucial! It adds 100 to the current offset attribute for the next API call).
   * **Hint:** This is how you increment your offset by the limit amount for the next page. We are assuming offset exists and starts at 0.
4. **Loop Back for More Data (Conditional Routing):**
   * **Activity:** Connect the success relationship of the UpdateAttribute (from step 3) back to your **InvokeHTTP** processor (the *same one* you used in Sub-task 2).
   * **Activity:** Right-click on the connection from UpdateAttribute back to InvokeHTTP.
   * **Details:** Add a **Funnel** processor between the UpdateAttribute (from step 3) and the InvokeHTTP from step 2 if you are connecting from different branches, to combine the flowfiles for the loop.
   * **Details:** On the connection leading *into* the InvokeHTTP processor (from the Funnel or directly from the UpdateAttribute if no funnel), use a **backpressure object threshold** and **backpressure data size threshold** appropriate for your system, but more importantly, set up **FlowFile Filtering** to control the loop.
     + **Use an Expression Language property:** For example, you might route only if next\_offset is less than total\_records.
     + **Relationship:** success (This means if the condition is met, it will go back to InvokeHTTP).
   * Hint: The key here is to pass the next\_offset value as a parameter in your InvokeHTTP's Remote URL for subsequent calls:

http://api.aviationstack.com/v1/flights?access\_key=${aviationstack.api.key}&airport\_iata=YUL&limit=100&offset=${next\_offset}

* + **Hint:** You'll need to define a termination condition for the loop. This usually involves comparing next\_offset with the total\_records obtained from the API response. You might need a **RouteOnAttribute** processor *after* the EvaluateJsonPath to decide if you loop again or exit. If next\_offset is greater than or equal to total\_records, send it to a different path (e.g., success which proceeds to saving files). Otherwise, send it back to loop.

**Sub-task 4: Saving JSON Files for Development/Testing**

1. **Store the JSON Responses:**
   * **Activity:** From the success relationship of your **InvokeHTTP** processor, connect it to a **PutFile** processor.
   * **Details:**
     + **Directory:** Specify a local directory path (e.g., /tmp/nifi\_flight\_data/).
     + **Conflict Resolution Strategy:** replace (or overwrite during testing, append if you want one large file, fail if you want to know when it conflicts). For this exercise, replace or append could work.
     + **File Name:** You might want to use Expression Language to give unique names, e.g., ${now():format('yyyyMMddHHmmssSSS')}\_${offset}.json or ${filename} if you aren't concatenating.
   * **Hint:** This is crucial for debugging! You can inspect these files to ensure the data is coming through correctly, and the pagination is working as expected. During development, you can even disable the InvokeHTTP and use **GetFile** or **ListFile/FetchFile** to read these saved files for quicker testing of subsequent processors.

**Summary Flow Outline (Conceptual):**

GenerateFlowFile -> UpdateAttribute (offset=0) -> InvokeHTTP (first call with offset=0) -> EvaluateJsonPath (extract total\_records) -> UpdateAttribute (calculate next\_offset) -> RouteOnAttribute (next\_offset < total\_records? -> LOOP)

* LOOP connects back to InvokeHTTP (which now uses ${next\_offset}).
* PutFile (connects from InvokeHTTP success, after EvaluateJsonPath perhaps, or directly from InvokeHTTP before the loop check if you want each response saved).

**Important Hints and Clarifications:**

* **Attribute Management:** Get comfortable with **UpdateAttribute** and **Expression Language (${attribute\_name} or ${attribute\_name:function()})**. They are your best friends for dynamic flow control and avoiding hardcoding.
* **Loop Termination:** The most critical part of pagination is knowing *when to stop*. Your RouteOnAttribute after EvaluateJsonPath (which extracts total\_records) will be key. If offset (or next\_offset) is greater than or equal to total\_records, your loop should stop, and the FlowFile should proceed to a "finished" path.
* **Error Handling:** For a real-world scenario, you'd add connections for failure relationships on InvokeHTTP to log errors or send alerts. For this homework, focus on the success path first.
* **Testing Iteratively:** Build your flow in small steps. Test the InvokeHTTP first to ensure you get data. Then add the UpdateAttribute and EvaluateJsonPath to get your offset and total records. Finally, implement the loop.
* **NiFi UI:** Don't forget to enable and start your processors and Controller Services once configured!

Good luck, data engineer! Let me know if you have specific questions about any of these steps.

## Q2 – Gemini

is "StandardSensitivePropertiesService" a controller service ? I cannot find it in my NiFi 1.27.0. Please clarify and provide an updated answer for NiFi 1.27.0. if required

Show thinking

## Gemini Updated

You are absolutely correct to call me out on that! My apologies.

You are right, **StandardSensitivePropertiesService is NOT a standard Controller Service that you can find or use directly in NiFi 1.27.0 (or typical NiFi versions) for general sensitive properties like an API key.** That was a misstep on my part, likely a confusion with how some specific custom services or more advanced security features might handle sensitive data.

Thank you for catching that; it's an important clarification for anyone working with NiFi.

Let's correct "Sub-task 1" to reflect the proper and widely used method in NiFi 1.27.0 for handling sensitive or variable information like an API key: **Using a Parameter Context.**

**Updated Sub-task 1: Setting Up Your Environment (Avoiding Hardcoding with Parameter Contexts)**

1. **Create a NiFi Parameter Context for Your API Key:**
   * **Activity:** In your NiFi canvas, go to the "Parameter Contexts" icon (the cylinder with the 'P' in the top right, next to the Controller Services gear).1
   * **Activity:** Click the + button to "Create new parameter context". Give it a meaningful name, e.g., Aviationstack\_API\_Context.
   * **Activity:** Once created, click the "Configure" button (pencil icon) for your new context.
   * **Details:** Add a new parameter.
     + **Parameter Name:** aviationstack.api.key (or similar, make it descriptive).
     + **Parameter Value:** Input your actual Aviationstack API key here.
     + **Sensitive (checkbox):** **Crucially, check this box!** This will mask the value once entered, ensuring it's not visible in plain text in the UI or logs.
   * **Hint:** Parameter Contexts are the modern and recommended way to manage variable and sensitive data that needs to be shared across multiple processors or even multiple process groups within your NiFi flow.2
2. **Apply the Parameter Context to Your Process Group (or Root Canvas):**
   * **Activity:** Right-click on the background of your NiFi canvas (or a specific Process Group if you've created one for this flow).
   * **Activity:** Select "Configure" from the context menu.
   * **Details:** In the "General" tab of the configuration dialog, you'll see a "Parameter Context" dropdown. Select the Aviationstack\_API\_Context you just created.
   * **Hint:** Applying the Parameter Context makes all its defined parameters (like aviationstack.api.key) available for use by any processor within that scope.

**How to Reference Your API Key in InvokeHTTP (remains the same but re-emphasized):**

When you configure your **InvokeHTTP** processor:

* For the access\_key query parameter, you will now reference it using the Parameter Context syntax:
  + access\_key=#{aviationstack.api.key}

This method ensures your API key is managed securely, not hardcoded, and easily updatable if needed, adhering to best practices in NiFi 1.27.0.

Thank you again for the correction!

## ChatGPT - how to build a **paginated API call flow using a single InvokeHTTP processor**

Perfect. Here's a **clear, processor-by-processor outline** of how to build a **paginated API call flow using a single InvokeHTTP processor** in Apache NiFi, with dynamic offset handling — just like a seasoned data engineer would set it up.

**🔄 NiFi Flow: Looping Pagination with Single InvokeHTTP Processor**

**🧠 GOAL:**

* Start at offset = 0
* Call the API with a limit of 100 flights from YUL
* Save each JSON result to a file
* Increment offset by 100 after each call
* Stop when no more results are returned (i.e., "data" array is empty)

**🔧 Processor-by-Processor Flow**

**🟩 1. GenerateFlowFile**

* **Purpose:** Trigger the flow — start the first API call.
* **Config:** No content or properties needed (empty flowfile).
* **Runs once** to initiate the loop.

**🟦 2. UpdateAttribute – Set Initial Variables**

* **Purpose:** Define the initial offset, limit, and other properties like API key and base URL.
* **Properties:**
* offset = 0
* limit = 100
* base\_url = http://api.aviationstack.com/v1/flights
* api\_key = your\_actual\_api\_key\_here
* dep\_iata = YUL

✅ **Hint:** You can also use a **Parameter Context** to store api\_key safely.

**🟨 3. ReplaceText – Build Full API Request URL**

* **Purpose:** Construct the full URL using attributes.
* **Replacement Strategy:** Entire text
* **Replacement Value (example):**
* ${base\_url}?access\_key=${api\_key}&dep\_iata=${dep\_iata}&limit=${limit}&offset=${offset}

This replaces the flowfile content with the full API URL.

**🟥 4. InvokeHTTP – Call the API**

* **Purpose:** Makes the HTTP request using the built URL.
* **Configuration:**
  + HTTP Method: GET
  + Remote URL: Use Expression Language to read from flowfile content (check Send Message Body)
  + Add Content-Type: application/json in headers

✅ **Output Response Body:** Set to store the response as the flowfile content.

**🟪 5. EvaluateJsonPath – Check if Data Exists**

* **Purpose:** Extract length of the "data" array to know if there are results.
* **Property:**
* data\_length => $.data.length()

**🟫 6. RouteOnAttribute – Check if More Data Exists**

* **Purpose:** Decide whether to continue looping or stop.
* **Routing Logic:**
  + has\_data route if ${data\_length:gt(0)}
  + no\_data route if not

✅ Route has\_data to save file + update offset  
✅ Route no\_data to LogMessage or Terminate

**🟩 7. PutFile – Save JSON Response**

* **Purpose:** Save the API result locally as a JSON file.
* **Directory:** Choose a folder like /data/aviation/flights\_raw
* **Filename:** Use a dynamic name, e.g.:
* flights\_yul\_offset\_${offset}.json

**🟦 8. UpdateAttribute – Increment Offset**

* **Purpose:** Increase offset by 100 for the next page.
* **Property:**
* offset => ${offset:toNumber():plus(100)}

**🔁 9. Loop Back – Send flow back to ReplaceText**

* This continues the flow with the new offset. You can do this with a **Connection** from the incremented flowfile back to the ReplaceText processor.

**🟥 10. Handle No Data (End Condition)**

* From the no\_data route, use:
  + LogAttribute or LogMessage for debugging
  + Or TerminateFlowFile to stop

**🧩 Optional Enhancements**

* Add a **Wait/Notify** strategy to throttle API calls if needed.
* Add **retry and failure handling** for InvokeHTTP (e.g. using RetryFlowFile, LogMessage, PutFile for failed payloads).
* Use a **Controller Service** for shared StandardSSLContextService or StandardHttpContextMap.

Would you like a **visual diagram (flowchart-style)** of this NiFi flow next?

## ChatGPT final solution 1

Excellent — here's a fully updated, **NiFi 1.27.0-compatible solution** for your "READING JSON FLIGHTS DATA" requirement. This version includes **detailed processor configurations**, **properties**, and any necessary **controller services and parameters**.

**✅ Overview of the Goal**

Ingest paginated flight data from [aviationstack.com](https://aviationstack.com/) for airport YUL using:

* A single InvokeHTTP processor
* Pagination logic with offset
* NiFi variables/parameters
* JSON saving using PutFile
* Clean termination when no more records

**🛠️ Step-by-Step: NiFi 1.27.0 Flow with Config Details**

**🔹 A. Controller Services Setup**

1. **StandardHttpContextMap**
   * Used by InvokeHTTP for managing stateful sessions (optional but good practice).
   * Go to **Controller Services** (in the canvas or global level) → Add → StandardHttpContextMap.
   * **Enable it**, and leave default settings unless you're using SSL or custom headers.
2. **Parameter Context (recommended)**
   * Go to **"Parameter Contexts"** and create one (e.g., aviationstack-params).
   * Add the following **parameters**:

| **Parameter Name** | **Value** | **Description** |
| --- | --- | --- |
| api\_key | your API key (from aviationstack) | Do **not** hardcode in processors |
| base\_url | http://api.aviationstack.com/v1/flights | Endpoint for flight data |
| dep\_iata | YUL | Departure airport filter |
| limit | 100 | Maximum results per page |

✅ Use ${parameter\_name} in all processor properties instead of hardcoding.

**🔹 B. NiFi Flow Processors and Properties**

**🟩 1. GenerateFlowFile**

* **Purpose:** Start the loop.
* **Run Schedule:** 0 sec (once or trigger manually)
* **Properties:** None needed
* **Success →** UpdateAttribute (initialize offset)

**🟦 2. UpdateAttribute – Set Initial Attributes**

* **Purpose:** Set offset to 0 for the first run
* **Properties:**

| **Attribute Key** | **Value** |
| --- | --- |
| offset | 0 |

✅ Use Expression Language: ${offset} to refer to this later.

**🟧 3. ReplaceText – Build API URL**

* **Purpose:** Construct full REST API URL
* **Settings:**
  + **Replacement Strategy:** Entire text
  + **Replacement Value:**
  + ${base\_url}?access\_key=${api\_key}&dep\_iata=${dep\_iata}&limit=${limit}&offset=${offset}
* **Success →** InvokeHTTP

**🟥 4. InvokeHTTP**

* **Purpose:** Call the API
* **Settings:**
  + **HTTP Method:** GET
  + **Remote URL:** ${filename} (because URL is in flowfile content)
  + **Send Message Body:** false
  + **Use HTTP Context Map:** Yes → select your StandardHttpContextMap
  + **Output Response Regardless of Status Code:** true
  + **Attributes to Send:** None
  + **Follow Redirects:** true
  + **SSL Context Service:** *(only if HTTPS required, not for HTTP)*

✅ Automatically stores the **API response** in flowfile content.

**🟪 5. EvaluateJsonPath – Get Length of data[]**

* **Purpose:** Check if there are results
* **Settings:**

| **Destination** | **flowfile-attribute** |
| --- | --- |
| data\_length | $.data.length() |

✅ This adds an attribute data\_length that we can evaluate.

**🟫 6. RouteOnAttribute – More Data or Stop**

* **Purpose:** Route based on presence of data
* **Routing Rules:**

| **Name** | **Expression** |
| --- | --- |
| has\_data | ${data\_length:gt(0)} |
| no\_data | ${data\_length:equals(0)} or leave unmatched default |

✅ Connect:

* has\_data → PutFile (store JSON) → UpdateAttribute (offset++) → loop
* no\_data → LogMessage or TerminateFlowFile

**🟨 7. PutFile – Save JSON Locally**

* **Purpose:** Save API results to disk
* **Settings:**
  + **Directory:** /data/aviation/flights\_raw/
  + **Conflict Resolution Strategy:** replace or ignore
  + **File Name:**
  + flights\_yul\_offset\_${offset}.json

✅ Creates separate file per page.

**🟦 8. UpdateAttribute – Increment Offset**

* **Purpose:** Move to next page
* **Settings:**

| **Attribute Key** | **Value** |
| --- | --- |
| offset | ${offset:toNumber():plus(${limit})} |

✅ This loops back to ReplaceText.

**🔁 9. Loop Flow**

* **Connect** UpdateAttribute (offset++) back to ReplaceText to re-invoke next page.
* **Auto-terminate** or log no\_data route.

**🔒 Sensitive Property Handling**

* **API Key:** Use Parameter Context or Sensitive Property in UpdateAttribute or InvokeHTTP
* Do **not** hardcode keys in text or URLs.
* Use Variable Registry or Parameter Context securely.

**✅ Summary of Flow Structure**

[GenerateFlowFile]

↓

[UpdateAttribute] (offset=0)

↓

[ReplaceText] → Builds API URL

↓

[InvokeHTTP]

↓

[EvaluateJsonPath] → Extract $.data.length()

↓

[RouteOnAttribute]

├── has\_data → [PutFile] → [UpdateAttribute (offset++)] → Loop

└── no\_data → [LogMessage] or [Terminate]

**🧩 Optional Improvements**

| **Feature** | **Description** |
| --- | --- |
| **Retry on failure** | Add RetryFlowFile between InvokeHTTP and the loop |
| **Handle API limits** | Use Wait/Notify to avoid 100 requests per month limit |
| **Log failed responses** | Use RouteOnResponseCode if status ≠ 200 |

Would you like a **sample template (.xml) for importing this flow into NiFi**, or a **diagram of the flow** as a visual aid?

## NiFi xml template by ChatGPT



## Correction to ChatGPT by ChatGPT

You're absolutely right again — thank you for pointing that out. In **NiFi 1.27.0**, the ReplaceText processor was redesigned, and what used to be Search Value and Replacement Strategy has been reorganized and renamed.

Let me now walk you through the **correct configuration of ReplaceText in NiFi 1.27.0** — step by step — using the current layout and terminology.

**✅ How to Use ReplaceText in NiFi 1.27.0 to Build the API URL**

**🎯 Goal:**

Turn the flowfile content into a dynamically generated URL like:

http://api.aviationstack.com/v1/flights?access\_key=abcd123&dep\_iata=YUL&limit=100&offset=200

**🔧 Step-by-Step Configuration**

Open the ReplaceText processor and configure the following **tabs**:

**🛠️ Settings Tab**

Leave defaults unless otherwise required. Important ones:

* **Run Schedule:** Default (0 sec)
* **Penalize on Failure:** No
* **Auto-terminate Relationships:** None (you will connect success path manually)

**🧩 Properties Tab**

| **Property** | **Value** | **Explanation** |
| --- | --- | --- |
| **Replacement Strategy** | Always Replace | Tells NiFi to ignore content and replace it |
| **Replacement Value** | ${base\_url}?access\_key=${api\_key}&dep\_iata=${dep\_iata}&limit=${limit}&offset=${offset} | This is the actual dynamic URL |
| **Evaluation Mode** | Entire Text | Replaces all the content of the flowfile |
| **Character Set** | UTF-8 | Leave as default |

🔎 You do **not** need a “Search Value” field in the current version. The **Evaluation Mode = Entire Text** does exactly what “Search Value = ^.\*$” used to do — it targets the whole content by default.

**✅ Final Behavior**

After the ReplaceText runs:

* Your flowfile's content will be completely replaced by the constructed URL using flowfile attributes.
* This URL is then passed to the next processor (InvokeHTTP) to make the API call.

**🧠 Gotchas to Avoid**

| **Issue** | **Solution** |
| --- | --- |
| API key missing | Make sure ${api\_key} is passed via Parameter Context or set in UpdateAttribute |
| Blank flowfile | GenerateFlowFile must create an initial file (even empty) to trigger the flow |
| Variables not replaced | Use Expression Language — attributes must exist before ReplaceText runs |

**🧪 Optional Test Tip**

You can **insert a LogAttribute** right after ReplaceText to confirm that:

* The flowfile content is the correct URL.
* All variables (offset, limit, etc.) are being resolved.

Would you like me to provide a small sample flowfile test with dummy attributes to see how this ReplaceText behaves?