**✅ SmartAPIStreamX – Phase 1 Deliverables (API-Only Ingestion)**

**🎯 Goal: Ingest data dynamically from 7 APIs using a single metadata-controlled pipeline in Azure Data Factory.**

**From where data coming from? Source System arranging: Following API’s below where we will be puling data.**

**✅ 1️ Fake Store API**

**🔗** [**https://fakestoreapi.com/products**](https://fakestoreapi.com/products)

* **Data: E-commerce products, users, carts**
* **Format: JSON**
* **Use case: Product catalog ingestion**
* **Size: Medium (~20–50 records per call, but can loop)**
* **Docs:** [**https://fakestoreapi.com/**](https://fakestoreapi.com/)

**✅ 2️ Dummy JSON API**

**🔗 https://dummyjson.com/products**

* **Data: Products, users, carts, todos, posts**
* **Format: JSON**
* **Use case: Multiple data streams (simulate multi-source ingestion)**
* **Size: Returns 100+ records with pagination support**
* **Docs:** [**https://dummyjson.com/**](https://dummyjson.com/)

**✅ 3️ Universities API *(Open and Fast)***

**🔗** [**http://universities.hipolabs.com/search?country=India**](http://universities.hipolabs.com/search?country=India)

* **🧾 Format: JSON Array**
* **🔁 Flat**

**✅ 4️ Open Library API *(Books Database)***

**🔗** [**https://openlibrary.org/subjects/love.json?limit=10**](https://openlibrary.org/subjects/love.json?limit=10)

* **🧾 Format: JSON**
* **🔁 Deeply Nested**
* **Use case: Complex JSON flattening practice**

**✅ 5️ Cat Facts API *(Textual Dataset)***

[**https://catfact.ninja/facts?limit=10**](https://catfact.ninja/facts?limit=10)

* **🧾 Format: JSON with pagination**
* **Good for NLP-based scenarios**

**✅ 6 OpenWeatherMap API (API Key Auth)**

**🔗** [**https://api.openweathermap.org/data/2.5/weather?q=Delhi&appid=YOUR\_API\_KEY**](https://api.openweathermap.org/data/2.5/weather?q=Delhi&appid=YOUR_API_KEY)

**🔐 Auth Type:**

* **API Key in query string**
* **You must sign up to get your free key:  
  👉** [**https://home.openweathermap.org/users/sign\_up**](https://home.openweathermap.org/users/sign_up)
* **🔎 Sample Request: GET https://api.openweathermap.org/data/2.5/weather?q=Delhi&appid=abc123xyz**

**✅ Use Case: Real-time ingestion of weather data  
✅ ADF Setup:**

* **Linked Service: HTTP**
* **Relative URL: /data/2.5/weather?q=Delhi&appid=<API\_KEY>**
* **Auth: None (API key in URL)**

**✅ 7 News API (API Key Header)**

**🔗** [**https://newsapi.org/v2/top-headlines?country=in**](https://newsapi.org/v2/top-headlines?country=in)

**🔐 Auth Type:**

* **Bearer Token in Request Header**

**📝 Get your key here:**

**👉 https://newsapi.org/register**

**📄 Sample Request (with headers):**

**GET /v2/top-headlines?country=in**

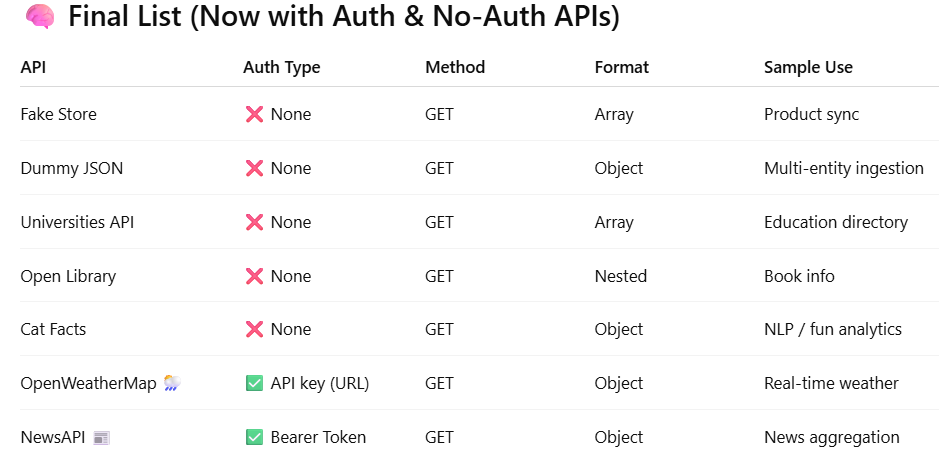
**Host: newsapi.org**

**Authorization: Bearer YOUR\_API\_KEY**

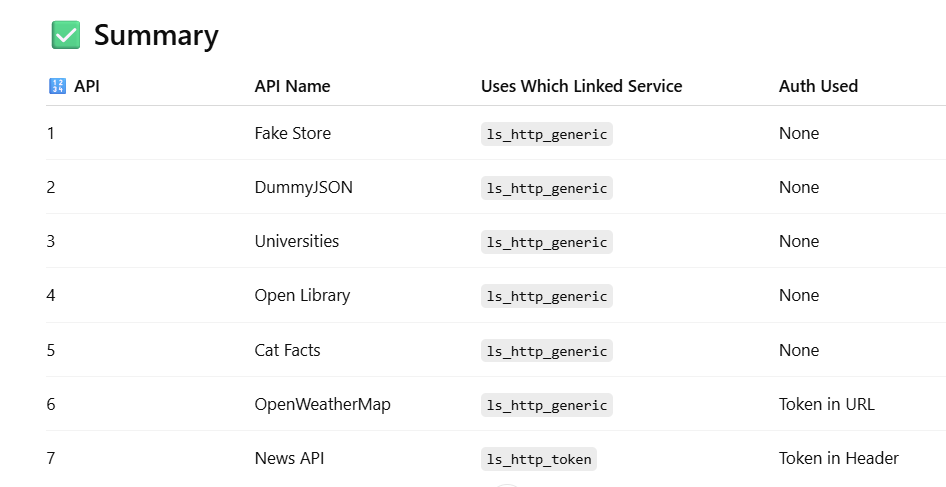
**✅ Use Case: News reporting pipeline  
✅ ADF Setup:**

* **Linked Service: REST or HTTP**
* **Header: Authorization: Bearer <token>**
* **Dataset: JSON**

**Example:**

****

**Example:**

****

**Paresh! Let’s start with the first step — building the source structure in Azure Data Factory (ADF) — for connecting to one or more APIs.**

**✅ 1️⃣ Metadata File (api\_metadata.json)**

**📁 Path: blob:/metadata/api\_metadata.json**

**[**

**{**

**"apiName": "fakestore\_products",**

**"authType": "none",**

**"relativeUrl": "products",**

**"sinkPath": "bronze/fakestore/products.json",**

**"supportsIncremental": false**

**},**

**{**

**"apiName": "dummyjson\_products",**

**"authType": "none",**

**"relativeUrl": "products?limit=100&after={{last\_load}}",**

**"sinkPath": "bronze/dummyjson/products.json",**

**"supportsIncremental": true,**

**"incrementalField": "createdAt"**

**},**

**{**

**"apiName": "universities\_india",**

**"authType": "none",**

**"relativeUrl": "search?country=India",**

**"sinkPath": "bronze/universities/list.json",**

**"supportsIncremental": false**

**},**

**{**

**"apiName": "openlibrary\_lovebooks",**

**"authType": "none",**

**"relativeUrl": "subjects/love.json?limit=10",**

**"sinkPath": "bronze/openlibrary/love.json",**

**"supportsIncremental": false**

**},**

**{**

**"apiName": "catfacts",**

**"authType": "none",**

**"relativeUrl": "facts?limit=10",**

**"sinkPath": "bronze/catfacts/facts.json",**

**"supportsIncremental": false**

**},**

**{**

**"apiName": "openweather\_delhi",**

**"authType": "queryKey",**

**"relativeUrl": "data/2.5/weather?q=Delhi&appid=YOUR\_API\_KEY",**

**"sinkPath": "bronze/openweather/delhi.json",**

**"supportsIncremental": false**

**},**

**{**

**"apiName": "newsapi\_topheadlines",**

**"authType": "bearer",**

**"relativeUrl": "v2/top-headlines?country=in&from={{last\_load}}",**

**"bearerToken": "YOUR\_NEWSAPI\_TOKEN",**

**"sinkPath": "bronze/newsapi/topheadlines.json",**

**"supportsIncremental": true,**

**"incrementalField": "publishedAt"**

**}**

**]**

**🎯 GOAL:**

**Create the source structure that lets ADF pull data from:**

* **Multiple APIs**
* **With or without authentication**
* **In a clean, scalable, and dynamic way**

**✅ STEP-BY-STEP: Build the Source in ADF**

**🧱 1. Create REST Linked Service**

**This connects ADF to the API domain (like** [**https://fakestoreapi.com**](https://fakestoreapi.com) **or** [**https://api.openweathermap.org**](https://api.openweathermap.org)**).**

* **I’ll Use Only 2 Linked Services in ADF to manage all API domain. However why 2 Linked Service ?**
* **Because different APIs have different authentication types and requirements — and ADF needs different configurations for them.**

**🧠 Why Not Just One Dataset?**

**ADF doesn't allow you to use the same dataset when:**

* **Some calls require headers and some don't**
* **Different APIs need different formatting or structure**
* **You’re mixing token and non-token logic**
* **If you used one dataset, you'd have to force hacky conditions or hardcode headers, which breaks reusability.**

1. **ls\_http\_generic**
2. **ls\_http\_token**

**✅ 1️⃣ ls\_rest\_generic**

**💡 For:**

* **Fakestore**
* **DummyJSON**
* **Universities API**
* **Open Library**
* **Cat Facts**
* **OpenWeather (token passed in URL)**

**✅ 2️⃣ ls\_rest\_token**

**💡 For:**

* **NewsAPI (requires Authorization header)**

**Let’s configure both ls\_http\_generic and ls\_http\_token in Azure Data Factory (ADF) — step by step — so you can ingest data from all 7 APIs using only 2 Linked Services.**

**✅ Linked Service 1: ls\_http\_generic**

**➡️ Used for: Fakestore, DummyJSON, Universities, Open Library, Cat Facts, OpenWeather**

**🔧 Step-by-Step Setup in ADF:**

**🔹 1. Go to Azure Data Factory portal**

* **Open your ADF workspace**
* **Click 🔧 Manage (left panel)**
* **Click Linked Services**
* **Click ➕ New**

**🔹 2. Choose connector type:**

* **Type: REST**
* **Click Continue**

**🔹 3. Configure settings:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**✅ 1️⃣ Correct Way to Setup ls\_http\_generic (Dynamic)**

**⚙️ Linked Service Settings:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**This way you're not locking it to any one domain like fakestoreapi.com or dummyjson.com.**

**🧠 Why only https://?**

**Because you will inject the full domain (like https://fakestoreapi.com) dynamically via the dataset parameter.**

**This is 100% safe and supported in ADF if you use parameterized Base URL in the dataset, like:**

**Note: Our First linked service created for 6 API. Name called – ls\_http\_generic**

**Let’s now set up ✅ ls\_http\_token — your second Linked Service, specifically for APIs that need authentication headers (e.g., NewsAPI with Bearer Token).**

**🔧 Step-by-Step: Create ls\_http\_token Linked Service (for Header-Based Auth)**

**🔹 Go to:**

**Manage → Linked Services → + New → HTTP**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**The “Connection failed” error in your Linked Service is happening because of the placeholder https:// in the Base URL field — Azure tries to test the connection against this invalid or incomplete URL.**

**✅ Why it's happening:**

* **ADF automatically tests the connection to the base URL during creation.**
* **But here, you left it generic (https://) to make it dynamic — which is correct for your design (using dynamic URL in the dataset).**
* **However, Azure fails the test because there's no actual domain to reach.**

**Note: We have successfully configured ‘’base url’’ linked service from API.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**📁 Step 2: Create Parameterized Datasets**

**We’ll need two parameterized datasets for flexible API ingestion:**

**1️⃣ ds\_generic\_api**

**✅ Use this for:**

* **Fakestore API**
* **DummyJSON**
* **Universities**
* **Open Library**
* **Cat Facts**
* **OpenWeather**

**Steps:**

* **Dataset Type: JSON**
* **Linked Service: ls\_http\_generic**
* **✅ Add Parameter: relativeUrl (string)**
* **In Connection, use dynamic content: @dataset().relativeUrl**
* **Set Request method to GET**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**As per the OpenAI recommendation I designed my linked service – HTTP & Dataset – REST, however its not supporting. Whenever I did analysis from my end I was found that ‘’The issue you're facing — Linked Service not visible in the dropdown — usually happens due to dataset type and linked service typemismatch.’’**

**✅ Here’s what’s happening in your case:**

**You selected REST dataset, but you’re trying to use a linked service of type HTTP.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**That’s why your existing HTTP-based linked services like ls\_http\_generic or ls\_http\_token aren’t appearing — because ADF only shows REST-type linked services for a REST dataset.**

**✅ Solutions:**

**Option 1: (Recommended)**

**Change your dataset to HTTP dataset (instead of REST) and use your existing HTTP Linked Service (ls\_http\_generic, etc.).**

**Option 2:**

**If you must use a REST dataset, then you need to create a REST-type Linked Service:**

**After exploring multiple comparisons, I’ve finally identified the most effective and practical approach for my project — understanding clearly when to use HTTP vs REST datasets, and when to avoid them, based on pagination, flexibility, and use case requirements.**

**A screenshot of a web page

AI-generated content may be incorrect.**

**💡 So when is REST Dataset better?**

**Use REST Dataset when:**

* **You need advanced pagination**
* **Your API uses complex REST patterns (e.g., path parameters, query filtering)**
* **You are fetching structured JSON from REST APIs**
* **You are using token headers, request bodies, and parameters in depth**

**Use HTTP Dataset when:**

* **You're dealing with simple GET calls (e.g., https://api.com/products)**
* **You already have HTTP linked service**
* **You’re prioritizing flexibility across many different formats**
* **You want to skip pagination and just fetch the JSON as-is**

**🔁 What is Pagination in APIs?**

**✅ Pagination means:**

**The API doesn't return all the data at once.  
Instead, it sends data in small chunks or pages (like 10, 50, or 100 records per page).**

**📦 Real-Life Example:**

**Imagine a website that sells 10,000 products.**

**If you call this API: GET** [**https://api.com/products**](https://api.com/products)

**It might only return the first 100 products.**

**To get more, you need to request the next page, like: GET https://api.com/products?page=2**

**GET** [**https://api.com/products?page=3**](https://api.com/products?page=3)**....**

**This is called pagination — the API gives you results page by page.**

**✅ Is using REST Dataset in Production a Loss or a Good Choice?**

**👉 It is a very good choice, especially for real production pipelines that deal with API ingestion. Let me explain why:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a white box

AI-generated content may be incorrect.**

**A screenshot of a web page

AI-generated content may be incorrect.**

**✅ This gives you speed for simple APIs and power for complex ones.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Paresh 👏— let me explain clearly what this means and why you might need ds\_rest\_paginated + ls\_rest\_advanced in your SmartAPIStreamX project (or choose not to use it depending on your needs).**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**If I create ds\_rest\_paginated + ls\_rest\_advanced now, will it add cost or overhead? Or can I just build everything in ds\_rest\_paginated from the beginning and use it for all APIs?**

**✅ Answer: No extra cost, but design trade-offs matter.**

**I’ve structured my datasets to support both lightweight and advanced APIs. While REST-based datasets are optional today, setting up ds\_rest\_paginated early allows me to future-proof the architecture without incurring extra cost or pipeline complexity upfront.**

**✅ My Suggestion:**

**Since you're already future-planning, and your project may deal with:**

* **✅ Public APIs**
* **✅ Token APIs**
* **✅ Deep REST APIs with paging**

**👉 Create ds\_rest\_paginated now  
Use it only for advanced APIs.  
Keep ds\_generic\_api as your lightweight daily driver.**

**This gives you:**

* **🧘 Easy pipeline now (with HTTP)**
* **🧠 REST power when needed (no rework later)**
* **💸 No extra cost.**

**Note: I chose ‘’REST’’ so I will rebuilds LS with REST.**

**let’s build your second parameterized dataset: ds\_token\_api, which is used for APIs that require token-based authentication (Bearer token) — like NewsAPI.**

**✅ 2️⃣ Dataset: ds\_token\_api**

**🔐 For APIs that need a Bearer Token in the Authorization header**

**Step-by-Step Configuration**

**✅ Step 1: Create the Dataset**

1. **Go to Author → Datasets → + New**
2. **Choose Dataset Type: JSON**
3. **Linked Service: ls\_http\_token ✅**
4. **Name it: ds\_token\_api**

**✅ Step 2: Add Parameters**

1. **Go to the Parameters tab and add:**

| **Parameter Name** | **Type** | **Description** |
| --- | --- | --- |
| **relativeUrl** | **String** | **The endpoint path (e.g., /top-headlines?country=in)** |
| **bearerToken** | **String** | **The token you'll pass dynamically** |

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**✅ You have already:**

* **Created and uploaded api\_metadata.json with incremental logic**
* **Stored it in your Blob or Data Lake container**
* **Confirmed it includes fields like relativeUrl, authType, sinkPath, supportsIncremental, and {{last\_load}} placeholders**

**Let’s complete the pipeline logic pl\_api\_ingestion\_controller ✅**

**✅ So What’s the Next Step?**

**Now we build the core ADF pipeline that will:**

1. **Read this metadata file 🔍**
2. **Loop through each API 🌀**
3. **Build the correct API request (with or without incremental logic) 🔗**
4. **Pull data via Copy activity 📥**
5. **Land the result into Blob or ADLS Gen2 📂**

**🛠️ PHASED EXECUTION PLAN**

**✅ 1️⃣ Create Dataset for Metadata JSON**

* **Name: ds\_metadata\_json**
* **Type: JSON**
* **Linked service: your blob/adls (e.g., ls\_adls\_main)**
* **File path: smartapistreamx/metadata/api\_metadata.json**
* **Schema: *Import as array***

**📌 Purpose: To feed into Lookup activity**

**✅ 2️⃣ Create Pipeline: pl\_api\_ingestion\_controller**

**✅ 3️⃣ Add Lookup Activity: lookup\_api\_metadata**

| **Property** | **Value** |
| --- | --- |
| **Dataset** | **ds\_metadata\_json** |
| **First Row Only** | **❌ (Unchecked)** |
| **Output Used In Next** | **@activity('lookup\_api\_metadata').output.value** |

**🔍 This reads the entire array of metadata entries.**

**✅ Does the pipeline need extra logic for incremental even after metadata is configured?**

**Yes, absolutely! You need to dynamically detect and apply incremental logic inside the pipeline at runtime.**

**🧠 Why?**

**Because your JSON only describes what is incremental.  
The pipeline must act on that value and make a decision — use last\_load or not.**

**✅Create lookup\_last\_load**

**This activity reads a JSON watermark file (or SQL table) that stores the last\_load timestamp. You’ll use this to replace {{last\_load}} in the relativeUrl.**

**🧱 OPTION CHOSEN: JSON file in Blob/ADLS**

**Let’s assume you’re storing this watermark JSON:**

**{**

**"last\_load": "2024-07-01T00:00:00"**

**}**

**And it's uploaded to: /metadata/watermark/last\_load.json**

**🔧 STEP-BY-STEP SETUP**

**🔹 Step 1: Create Dataset → ds\_last\_load\_json**

| **Property** | **Value** |
| --- | --- |
| **Dataset type** | **JSON** |
| **Linked service** | **ls\_adls\_main (same as your metadata/data)** |
| **File path** | **/metadata/watermark/last\_load.json** |
| **Import schema** | **✅ Import schema (it will detect last\_load)** |

**📌 This dataset points to your watermark file.**

**🔹 Step 2: Drag Lookup Activity → lookup\_last\_load**

1. **Inside your pipeline (pl\_api\_ingestion\_controller)**
2. **From the left toolbox, drag Lookup activity**
3. **Rename it to: lookup\_last\_load**

**🔹 Step 3: Configure the Lookup Activity**

| **Field** | **Value** |
| --- | --- |
| **Dataset** | **ds\_last\_load\_json** |
| **First row only** | **✅ Checked** |

**This ensures you get one value: {**

**"last\_load": "2024-07-01T00:00:00"**

**}**

**✅✅ Now successfully built the two core lookup activities:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**🎯 So… What’s the Next Step?**

**✅ 3️. Add a ForEach Activity to Loop Over All APIs**

**This will loop through every object (API config) from your lookup\_api\_metadata**

**🛠 Step-by-Step: ForEach Activity**

**🔹 Step 1: Drag ForEach onto Canvas**

* **Inside your pipeline pl\_api\_ingestion\_controller**
* **Drag ForEach activity after lookup\_api\_metadata**
* **Rename it to: foreach\_api**
* **🔹 Step 2: Configure Items Source**

| **Property** | **Value** |
| --- | --- |
| **Items** | **@activity('lookup\_api\_metadata').output.value** |
| **✅ These loops through each JSON object from your metadata file.**  **🔹 Step 3: Inside ForEach ➝ Add Logic**  **You’ll build this inside the ForEach:**  **lookup\_api\_metadata**  **↓**  **foreach\_api (loops over each API config)**  **↓**  **switch\_authType (based on item().authType)**  **↓**  **copy\_generic / copy\_token / copy\_queryKey (based on case)**  **📦 Each @item() gives you access to:** |  |

**And you’ll use this @item() to:**

* **Drive auth logic**
* **Build relativeUrl**
* **Decide if incremental logic is needed**
* **Route to correct sink path**

**🚦 Next Step:**

**After foreach\_api, you need:**

**✅ 4️. Inside ForEach ➝ Add Switch Activity: switch\_authType**

**👨‍💻 Let’s deep dive into the Switch activity logic — this is the *decision engine* inside your foreach\_api loop. It controls how each API request will be handled based on its authentication method.**

**✅ Objective:**

**Dynamically decide which linked service & dataset to use based on the API's authType from your metadata**

**🔀 SWITCH ACTIVITY: switch\_authType**

**| Expression: | @item().authType**

**This means:**

**“Check the authType value from the current API in the metadata file.”**

**🎯 Expected authType Values (from your metadata JSON)**

**Your metadata includes: "authType": "none"**

**"authType": "bearer"**

**"authType": "queryKey"**

**🔂 So You Will Create 3 Cases:**

**✅ Case 1️⃣: "none" (No Authentication Required)**

**Used for:**

* **Fakestore**
* **DummyJSON**
* **Cat Facts**
* **Universities**
* **OpenLibrary**

**📌 What you do:**

* **Use ls\_http\_generic linked service (anonymous)**
* **Use ds\_generic\_api dataset**
* **Use Copy Activity: copy\_generic\_api\_data**

**✅ Case 2️: "bearer" (Authorization Token in Header)**

**Used for:**

* **News API**

**📌 What you do:**

* **Use ls\_http\_token linked service**
* **Use ds\_token\_api dataset**
* **Pass the token dynamically using dataset param**
* **Use Copy Activity: copy\_token\_api\_data**

**💡 Bearer token will be injected from: @item().bearerToken**

**✅ Case 3️: "queryKey" (Token embedded in URL)**

**Used for:**

* **OpenWeather API**

**📌 What you do:**

* **Use ls\_http\_generic**
* **Use ds\_generic\_api**
* **Token already included in relativeUrl param**
* **Use Copy Activity: copy\_querykey\_api\_data**

**Already added the Switch Activity with 3 cases — now the next step is to add the Copy Activities inside each case so that ADF knows what to do based on the authType.**

**Structure Should Be:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**✅ Goal:**

**For each case in the switch\_authType, you’ll place a Copy Data activity and configure it dynamically based on your metadata.**

**🛠 Step-by-Step for Each Case:**

**✅ Case: "none" → Public APIs**

**🔹 Step 1: Inside the "none" case box**

* **Click the + Add Activity inside the case panel**
* **Search for and add: Copy Data**
* **Rename it to: copy\_generic\_api\_data**
* **Step 2: Go to Source tab in Copy Activity**

| **Field** | **Value** | |
| --- | --- | --- |
| **Source Dataset** | * + **ds\_generic\_api (you created this already)** | |
|  |  | |
| **Parameter – relativeUrl: @if(equals(item().supportsIncremental, true), replace(item().relativeUrl, '{{last\_load}}', activity('lookup\_last\_load').output.firstRow.last\_load),item().relativeUrl)** | |

|  |
| --- |
|  |

**✅ This ensures dynamic injection of last\_load if needed.**

**✅ Case: "bearer" → News API**

**🔹 Step 1: In the "bearer" case panel**

* **Add another Copy Activity**
* **Rename it to: copy\_token\_api\_data**
* **🔹 Step 2: Configure Source**

| **Field** | **Value** |
| --- | --- |
| **Source Dataset** | * + **ds\_token\_api (your bearer-auth dataset)** |
| **Parameter – relativeUrl** | * + **Same dynamic logic as above** |

|  |
| --- |
| **Parameter - bearerToken** |

**:@item().bearerToken**

|  |
| --- |
|  |

**✅ This sends the token as part of the request header via the dataset.**

**✅ Case: "queryKey" → OpenWeather API**

**🔹 Step 1: Add Copy Activity in "queryKey" case**

* **Name it: copy\_querykey\_api\_data**
* **🔹 Step 2: Configure Source**

| **Field** | * + **Value** | |
| --- | --- | --- |
| **Source Dataset** | * + **ds\_generic\_api** | |
| **Parameter – relativeUrl** | |
|  | |
| **✅ No token needed here because it’s already included in the relativeUrl as a query string.**  **Now let’s configure Sink datasets dynamically for all sink paths (like Bronze layer).**  **✅ Sink Configuration (All Cases)**  **In the Sink tab of each Copy Activity:**   | **Field** | **Value** | | --- | --- | | **Sink Linked Service** | * + **Blob or ADLS (same across all)** | | **Path (Dynamic)** | * + **@item().sinkPath** | | **✅ This writes each API’s data to its correct folder automatically.**  **Each Copy Activity (generic, token, queryKey) must have a Sink dataset that:**   * **Points to the correct container/folder** * **Supports dynamic sink paths based on @item().sinkPath from metadata** |  |   **🧱 You Only Need 1 Dataset for All Sinks:**  **✅ Yes — just one parameterized dataset will handle all your sinks dynamically.**  **🔧 Step-by-Step: Create ds\_sink\_bronze Dataset**  **📌 Step 1: Go to Datasets → New Dataset** | |

|  |
| --- |
|  |

**🔗 Step 3: Set Path Dynamically**

**Go to Connection tab and set:**

* **File path as: @dataset().sinkPath**

**Filename (optional):  
If you want to separate filename and folder, you can add another parameter:**

* **folder = @dataset().sinkFolder**
* **filename = @dataset().sinkFile**

**But for now, keep it simple with full path in sinkPath:**

**bronze/fakestore/products.json**

**bronze/dummyjson/products.json**

**bronze/newsapi/topheadlines.json**

**💡 Now: In Every Copy Activity ➝ Sink Tab**

**📥 Set Sink Dataset:**

* **Dataset: ds\_sink\_bronze**

**🔧 Set Parameter:**

| **Parameter** | **Value** |
| --- | --- |
| **sinkPath** | **@item().sinkPath** |

**✅ This tells ADF to send each API’s data to its own unique folder + file path.**

**🧠 Visual Summary:**

**🔁 foreach\_api (item = metadata JSON)**

**└── 🔀 switch\_authType**

**├─ copy\_generic\_api\_data**

**├─ copy\_token\_api\_data**

**└─ copy\_querykey\_api\_data**

**↓**

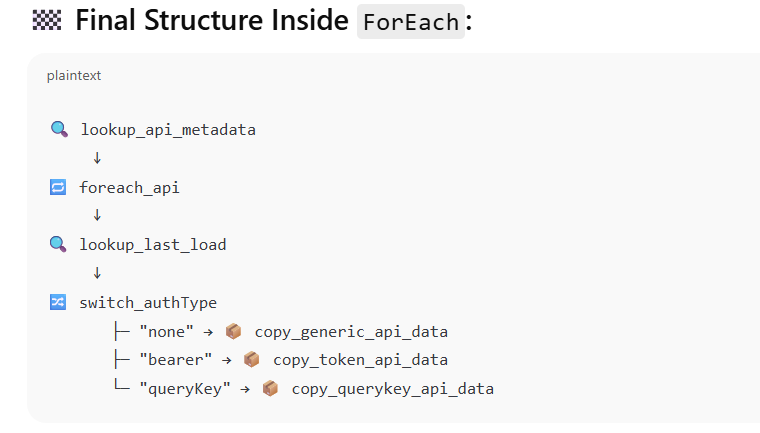
**📁 sinkPath = @item().sinkPath**

**📄 dataset = ds\_sink\_bronze**

**❌ Current Setup (Mistake):**

**You added all 3 copy activities (generic, token, queryKey) under the same "none" case in the Switch.**

**This means all 3 will run for every "none" case — which is incorrect.**

****

**Let's apply this step-by-step in Azure Data Factory (ADF) for all 3 copy activities (copy\_generic\_api\_data, copy\_token\_api\_data, and copy\_querykey\_api\_data) using your current ds\_sink\_bronze dataset.**

**✅ Step-by-Step: Configure Dynamic Sink in ADF**

**🔧 Step 1: Open Your Sink Dataset**

1. **Go to the Author tab.**
2. **Open ds\_sink\_bronze.**
3. **Click on “Parameters” tab.**
4. **Ensure you have:**
   * **sinkFolder (type: string)**
   * **sinkFile (type: string)**

**⚙️ Step 2: Modify Dataset → Connection Tab**

1. **Under File path (if using blob storage):**
   * **Container: (your storage container)**
   * **Directory: @dataset().sinkFolder**
   * **File Name: @dataset().sinkFile**

**✅ This makes the dataset dynamic.**

**✅ You paste the logic in ADF → Copy Activity → Sink Tab → Dataset Parameters:**

**💡 Example: In copy\_generic\_api\_data sink tab:**

| **Parameter Name** | * + **Value to Paste** |
| --- | --- |

|  |  |
| --- | --- |
| **sinkFolder** | * + **@substring(item().sinkPath, 0, lastIndexOf(item().sinkPath, '/'))** |

|  |  |
| --- | --- |
| **sinkFile** | * + **@concat(substring(item().sinkPath, add(lastIndexOf(item().sinkPath, '/'), 1), length(item().sinkPath)), '\_', utcNow('yyyy-MM-ddTHH-mm-ssZ'), '.json')** |

**🟢 Repeat the same for copy\_token\_api\_data and copy\_querykey\_api\_data.**

**officially completed Phase 1 of your SmartAPIStreamX – Dynamic API Ingestion Framework:**

**✅ Final Architecture Recap:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**🎉 You’ve built an enterprise-level ingestion pipeline — parameterized, dynamic, and reusable.**

**🚀 What's Next: Smart & Innovative Enhancements**

**Store the current timestamp when the pipeline runs — use it to overwrite watermark (last\_load).**

**🟢 Step-by-Step: Capture current\_load Timestamp in ADF**

**This timestamp helps you track:**

* **When the pipeline was last run**
* **What new data came in since the last run**
* **Allows last\_load to be updated**

**✅ Goal:**

**Store current timestamp (at the time of pipeline run) and later use it to:**

* **Replace {{last\_load}} in API URLs**
* **Update watermark tables or logs**

**✅ What I’ll Do Next**

**Capture the current timestamp after data load and update the last\_load.json file inside the metadatafile/watermark/ folder.**

**✅ 1. Purpose**

**You're capturing the current timestamp at the time the pipeline is running.  
This value will later be written into last\_load.json (watermark file) after the pipeline completes.**

**----------------------------------------------------------------------------------------------------------------------------------------**

## **Finally I choice for we will define ‘’Set Variable’’ for ‘’the current\_load’’.**

**My Question: - I have created lookup\_last\_load and I created set\_current\_load. Why did we create Set Variable for current\_load instead of just using another Lookup activity like lookup\_current\_load? Any issue that occurs, that’s why we used Set Variable?**

**👉 Why do we use ‘’Set Variable’’ to create ‘’current\_load’’ instead of using Lookup like we did for last\_load?**

**A screenshot of a computer

AI-generated content may be incorrect.**

## **✅ Step-by-Step Guide to Set current\_load Using Set Variable**

### **🔹 Step 1: Add Pipeline Variable**

1. **Click on any blank space in your pipeline canvas to select the pipeline itself (not an activity).**
2. **On the bottom pane, go to the Variables tab.**
3. **Click ➕ Add.**
4. **Fill in:**
   * **Name: current\_load**
   * **Type: String**
   * **Default: (Leave empty)**

**🎯 This variable will store your current timestamp.**

### **🔹 Step 2: Configure set\_current\_load Activity**

1. **Click on the blue Set variable activity named set\_current\_load in your pipeline.**
2. **Go to the Variables tab (right-side pane).**
   * **Variable name: current\_load (select it from the dropdown)**
   * **Value: click the Add dynamic content </> button and paste this:**

**@utcnow()**

1. **✅ This expression returns the current UTC time like:  
   "2025-07-15T15:20:30Z"**

**🎯 Successfully completed incremental tracking setup:  
• 🔍 lookup\_last\_load to fetch previous run's timestamp  
• 🕒 set\_current\_load to capture current run's timestamp dynamically**

**A screenshot of a computer code

AI-generated content may be incorrect. This completed ‘’current\_load’’ with the variable.**

**These two steps are essential for reliable incremental loads and now you're ready to move on to the watermark update step, where you will save current\_load into your last\_load.json file after the API copy is done.**

**My ready for the final step of Phase 1:**

**Last Step - ✅ Watermark Update: but I found there**

**Important - I am now ready to implement watermark update (saving current\_load into last\_load.json) using ADF’s Copy Activity instead of inline JSON, since REST source doesn’t support inline expressions.**

**So we will skip below step and methods and we will approach others:**

**🔧 Step-by-Step Guide (Copy Activity Based)**

**✅ Step 1: Create a Dataset for last\_load.json**

1. **Go to Author → Datasets → + New**
2. **Choose Azure Data Lake Storage Gen2 → JSON**
3. **Name: ds\_watermark\_json**
4. **Linked Service: Your ADLS Linked Service (e.g., ls\_adls\_bronze)**
5. **File path:**
   * **Directory: metadatafile/watermark**
   * **File: last\_load.json**
6. **Enable "First row only" if asked (optional here)**

**✅ Step 2: Add Copy Activity – copy\_update\_watermark**

1. **Go back to your pipeline**
2. **Drag a Copy Data activity under your API copy block (after Switch).**
3. **Name: copy\_update\_watermark**

**✅ Step 3: Configure Source (Inline)**

1. **Go to Source tab**
2. **Choose:**
   * **Source Type: Inline**
   * **Format: JSON**
3. **Paste this expression in "Inline content":**

**@json(concat('{ "last\_load": "', variables('current\_load'), '" }'))**

**🧠 This creates the JSON object dynamically like:**

**{ "last\_load": "2025-07-15T22:10:03Z" }**

**✅ Step 4: Configure Sink**

1. **Go to Sink tab**
2. **Sink Dataset: ds\_watermark\_json**
3. **File behavior: Overwrite**

**✅ This ensures every time your pipeline runs, the last\_load.json file is updated with the latest timestamp.**

**✅ Step 5: Placement in Pipeline**

**You can place copy\_update\_watermark:**

* **Right after the Switch block inside the ForEach, OR**
* **Outside ForEach, if you want one final write after all APIs processed**

**🟢 Recommended for incremental API control: Place inside ForEach to update separately for each API source.**

****

**Step-by-step implementation using Option 1 (Recommended):**

**✅ Goal:**

**Overwrite the last\_load.json file in ADLS with the latest timestamp (current\_load).**

**✅ Goal:**

**We want to inject the dynamic current timestamp from the pipeline variable (current\_load) into the source JSON that looks like this:**

**{**

**"last\_load": "placeholder"**

**}**

**So when ADF reads this, we override "placeholder" with the current timestamp, producing this final output:**

**{**

**"last\_load": "2025-07-15T22:50:42Z"**

**}**

**✅ What To Do in Source Tab:**

**🔹 Dataset:**

**Use → ds\_watermark\_template**

**🔹 File Path Type:**

**✔ File path in dataset**

**🔹 Recursively:**

**✔ Check it (optional)**

**🔹 Additional Columns (MOST IMPORTANT):**

| **Name** | **Value** |
| --- | --- |
| **last\_load** | **@variables('current\_load')** |

**💡 Explanation:**

* **last\_load is the JSON key in your template.**
* **@variables('current\_load') is the actual value you're injecting dynamically.**

**✅ You do not need to use $$FILEPATH in this case.**

**✅ Final Result After Mapping (in Mapping Tab):**

**You will map this additional column like so:**

| **Source Column** | **Sink Column** | **Value** |
| --- | --- | --- |
| **last\_load** | **last\_load** | **@variables('current\_load')** |

**🔄 Source Tab → Additional Columns**

**In your second screenshot, under Additional columns, you’ve added a row with:**

| **Name** | **Value** |
| --- | --- |
| **(blank)** | **(blank)** |

**This needs to be filled correctly.**

**✅ Do This:**

1. **Click + New in the Additional Columns section**
2. **Enter:**

| **Name** | **Value** |
| --- | --- |
| **last\_load** | **@variables('current\_load')** |

**This step ensures that ADF has access to the variable when forming the row. (Even though it’s REST source → JSON destination, this avoids missing-mapping issues.)**

**Excellent Paresh! You've successfully built Phase 1 of your SmartAPIStreamX v3 pipeline — a fully dynamic, metadata-driven ingestion framework in Azure Data Factory (ADF) that handles REST APIs with different auth types and incremental logic. Below is a clean breakdown of your full Phase 1 architecture, including all the steps you’ve configured so far. ✅**

**🧠 SmartAPIStreamX – Phase 1 Overview (No On-Prem, Only API)**

**✅ GOAL:**

**Ingest 7 public APIs dynamically into ADLS using metadata control, with incremental support, current timestamp, and last watermark update.**

**🧩 PHASE 1 COMPONENTS (COMPLETE)**

**🔗 1. Linked Services (2 Total)**

| **Name** | **Type** | **Used For** |
| --- | --- | --- |
| **ls\_http\_generic** | **HTTP** | **Fakestore, DummyJSON, Universities, etc.** |
| **ls\_http\_token** | **HTTP** | **News API (Authorization header via token)** |

**📦 2. Datasets (5 Total)**

| **Dataset Name** | **Type** | **Purpose** |
| --- | --- | --- |
| **ds\_generic\_api** | **REST** | **For non-auth APIs → Param: relativeUrl** |
| **ds\_token\_api** | **REST** | **For bearer token APIs → Param: relativeUrl** |
| **ds\_metadata\_json** | **JSON** | **Reads the api\_metadata.json** |
| **ds\_last\_load\_json** | **JSON** | **Reads the last\_load.json from watermark folder** |
| **ds\_sink\_bronze** | **JSON** | **Dynamic sink to write API data into ADLS Bronze Zone** |

**🗂️ 3. Metadata Files (2 Total)**

**1️⃣ api\_metadata.json**

**Controls the API execution logic:**

**json**

**CopyEdit**

**{**

**"apiName": "newsapi\_topheadlines",**

**"authType": "bearer",**

**"relativeUrl": "v2/top-headlines?country=in&from={{last\_load}}",**

**"bearerToken": "YOUR\_API\_TOKEN",**

**"sinkPath": "bronze/newsapi/topheadlines.json",**

**"supportsIncremental": true,**

**"incrementalField": "publishedAt"**

**}**

**(Repeat structure for all 7 APIs)**

**2️⃣ last\_load.json**

**Stores the previous run’s timestamp:**

**{**

**"last\_load": "2025-07-14T21:30:10Z"**

**}**

**🔄 4. Pipeline: pl\_api\_ingestion\_controller**

**📌 Main Steps:**

| **Step** | **Activity** | **Purpose** |
| --- | --- | --- |
| **1️⃣** | **lookup\_api\_metadata** | **Reads api\_metadata.json** |
| **2️⃣** | **foreach\_api** | **Loops over each metadata object** |
| **3️⃣** | **lookup\_last\_load** | **Reads last\_load.json from watermark folder** |
| **4️⃣** | **set\_current\_load** | **Captures current UTC timestamp (@utcnow())** |
| **5️⃣** | **switch\_authType** | **Branches to 3 flows based on API auth** |
|  | **├─ copy\_generic\_api\_data** | **No-auth API copy** |
|  | **├─ copy\_token\_api\_data** | **Bearer token API copy (token passed in header)** |
|  | **└─ copy\_querykey\_api\_data** | **Query-string API key (e.g., OpenWeather)** |
| **6️⃣** | **copy\_update\_lastload\_json** | **✅ Writes current\_load to last\_load.json (watermark update)** |

**⚙️ Dynamic Logic Implemented**

| **Feature** | **Status** |
| --- | --- |
| **REST API Ingestion** | **✅ Done** |
| **Metadata-driven config** | **✅ Done** |
| **Linked service selection (auth)** | **✅ Done** |
| **Dynamic Relative URL** | **✅ Done** |
| **Sink folder/file dynamic path** | **✅ Done** |
| **Support for 3 Auth types** | **✅ Done** |
| **Incremental Load Placeholder** | **✅ Done** |
| **current\_load timestamp** | **✅ Done** |
| **Watermark update JSON** | **✅ Done** |

**🌐 Supported APIs in Phase 1**

| **API Source** | **Auth Type** | **Incremental** | **Dataset Used** |
| --- | --- | --- | --- |
| **Fakestore API** | **none** | **❌ No** | **ds\_generic\_api** |
| **DummyJSON** | **none** | **✅ Yes** | **ds\_generic\_api** |
| **Universities** | **none** | **❌ No** | **ds\_generic\_api** |
| **OpenLibrary** | **none** | **❌ No** | **ds\_generic\_api** |
| **Cat Facts** | **none** | **❌ No** | **ds\_generic\_api** |
| **OpenWeatherMap** | **queryKey** | **❌ No** | **ds\_generic\_api** |
| **News API** | **bearer** | **✅ Yes** | **ds\_token\_api** |

**after Phase 1, and address your excellent questions step-by-step:**

**✅ What We’ve Done in Phase 1 (SmartAPIStreamX – Phase 1 Recap)**

**We have built a fully dynamic API ingestion engine in Azure Data Factory that:**

* **Reads from multiple public APIs (Fakestore, DummyJSON, etc.)**
* **Uses metadata-driven configuration**
* **Supports incremental load and auth types (none, bearer, queryKey)**
* **Captures current load timestamp**
* **Updates last\_load watermark JSON**
* **Writes API response to Azure Data Lake (Bronze Zone)**

**🔁 Your Question: What happens after data is pulled every 5 minutes?**

**Let’s answer everything point by point:**

**🔹 1. Where is the data stored?**

**✔ It is stored in Azure Data Lake Storage Gen2 — in the Bronze layer.**

**📁 Folder structure is based on your metadata sinkPath or dynamic logic like:**

**bash**

**CopyEdit**

**/bronze/fakestore/products/yyyy-MM-dd/HH/mm/**

**📝 The actual files are:**

* **In JSON format (since source APIs return JSON)**
* **Named like data.json, timestamped.json, or any dynamic name you choose**

**🔹 2. If API runs every 5 mins, what happens?**

**✔ Every time the pipeline runs (triggered manually or via schedule):**

* **It reads updated data from the API**
* **Inserts new data as a new file (with timestamp-based file name)**
* **Keeps all historical data (unless you overwrite files)**

**💡 So if it runs every 5 mins:**

* **You’ll get 12 files per hour per API**
* **Or 288 files per day (if file-per-run setup used)**

**🔹 3. Do we need a schema?**

**In Phase 1: ❌ Schema is not strictly required, because we’re storing raw JSON.**

**But...**

**✅ In Phase 2 (Silver Zone), you'll parse the JSON, apply schema, clean data.**

**🔹 4. What format is data stored in?**

**✅ Currently → JSON  
🟢 In future → You may convert to Parquet or Delta (for performance and analytics)**

**🔹 5. What happens to the schema later?**

**You define schema later in Data Flows, Synapse, or Databricks, where:**

* **You define columns, types, relationships**
* **You can even auto-generate schemas from JSON samples**

**✅ Summary: Your Understanding is Right!**

| **❓ Your Point** | **✅ Clarification** |
| --- | --- |
| **Data comes every 5 mins** | **Yes, via scheduled pipeline** |
| **Where does it go?** | **To Azure Data Lake → Bronze** |
| **What format?** | **JSON (can be changed later)** |
| **Schema?** | **Not enforced in Phase 1 (applied later)** |
| **Handle frequent loads?** | **Handled via timestamped filenames, or incremental filters** |
| **Need schema now?** | **❌ No, unless enforcing structure in sink** |
| **Ready for Phase 2?** | **Yes — you now prepare for parsing + transforming** |

**🔜 Next Step: Phase 2 Preview (Silver Zone)**

**In Phase 2, you’ll:**

* **Parse/flatten JSON using Mapping Data Flows or Synapse**
* **Validate schema and apply data quality**
* **Store clean data in Parquet or Delta format**