Retrieving Data Part 1 - APIs

Learning objectives

By the end we, should be able to:

- understand the purpose and high-level functioning of REST APIs;
- know of alternative solutions to APIs for data retrieval;
- describe what IoT devices are and their common architectural components;
- list types of data sources that can be commonly ingested; and
- perform a simple exercise in extracting data from an API.

1. Introduction to REST APIs

What is REST?

Having a good understanding of REST and its use cases is very helpful, regardless of if you are a client-facing software developer, a data engineer, or anyone wanting access to information on the World Wide Web.

REST APIs are a common way in which software applications communicate. By understanding how they work, we can successfully build fully integrated software applications and systems that can potentially influence the lives of millions of people.

REST is an abbreviation for *REpresentational State Transfer*. It is a web standards-based architecture that uses Hypertext Transfer Protocol (HTTP). REST was first introduced by Roy Fielding in his 2000 dissertation which is widely considered the "REST bible".

REST architecture provides access to resources through a REST server, where a REST client can then access and modify the resources. Each resource has its own uniquely identified URI or global ID. Some of the common types of resource representations include text, JSON, and XML. JSON is the most popular REST resource representation.

Before REST became the standard for transferring data, we would have used the SOAP protocol. SOAP stands for *Simple Object Access Protocol*, and the main idea behind SOAP was to ensure that programs built on different platforms and programming languages could easily exchange data. You may find yourself encountering SOAP one day while working with a legacy system. Here are some key differences between REST and SOAP:

- REST is an architectural pattern; SOAP is a protocol.
- REST uses less bandwidth compared to SOAP.
- SOAP only works with XML formats, while REST can work with text, XML, HTML, and JSON formats.
- SOAP cannot make use of REST, but REST can make use of SOAP.

Pattern vs. protocol

Typically in technology, a *pattern* is a widely accepted template for a common scenario or problem. A *protocol* is a fixed set of rules that dictate how two or more software entities should interact. From this definition, it's important to understand that there is no official standard for RESTful web APIs, while there is one for SOAP-based web services. This is because REST has a set of architectural rules that are *recommended* for a system to qualify as RESTful, and within these rules, REST can make use of standard protocols such as HTTP, JSON, URI, and XML. SOAP, on the other hand, is required to use either XML or HTTP and is a set protocol for information exchange.

We also mentioned that SOAP cannot make use of REST, but REST can make use of SOAP. For similar reasons to the above, this is true because REST is a set of architectural guidelines, but SOAP has a fixed protocol. We could use SOAP as a means to exchange information but use RESTful architectural constraints. As long as this system does not violate REST architectural rules it can qualify as RESTful even though it makes use of SOAP to exchange information. The opposite is not true, though; we cannot use SOAP alone and say that the system is RESTful.

REST architectural constraints

For REST APIs to perform optimally and stay within the definition of a REST architecture, some constraints should be adhered to:

1. **Client-server architecture** â€“ The client (front-end) is responsible for the user interface and the server is responsible for the back-end and the storage of data. Thus, the client and server are independent of each other and can be replaced.
2. **Stateless** â€“ No data from the client can be stored on the server side. All session state data must be stored on the client side.
3. **Cacheable** â€“ Clients should be able to (temporarily) cache server responses to improve performance.

More detailed information on the constraints is availablehere.

Methods available with REST APIs

Within the REST architecture, the following four HTTP methods see widespread usage:

- **GET**: Provides read-only access to a specific resource.
- **POST**: This method is used to create a new resource.
- **DELETE**: This method is used to remove a resource.
- **PUT**: This method can either be used to create a new resource or, more commonly, to update an existing resource.

To help us understand how these methods work, the following image represents the system overview and information flow used when a user interacts via an API.

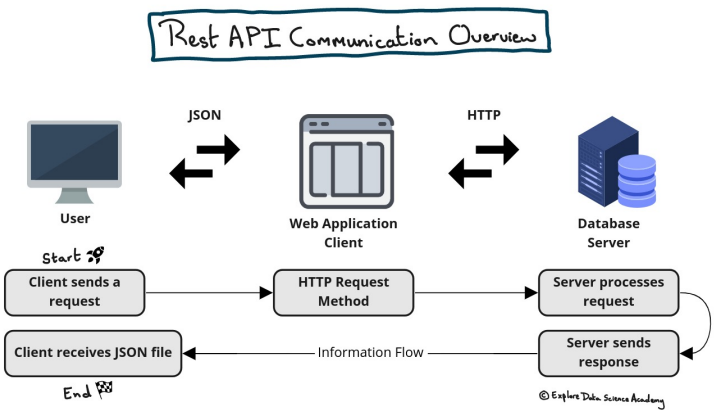

Figure 1. The general flow of information when interacting with a REST API through HTTP request methods.

In this example, a client web app is sending HTTP requests to get, update, delete, and create information on a database server. Let's consider what each of these methods would look like in some detail below.

GET request

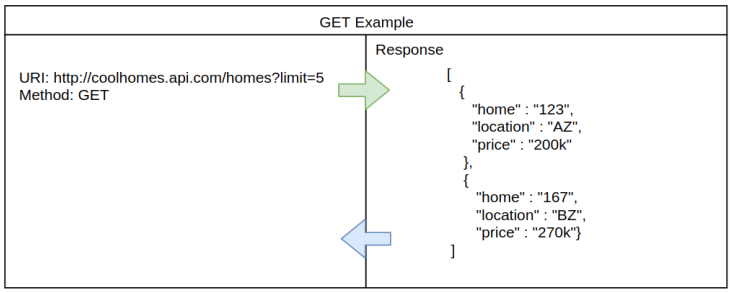

Figure 2. The flow of information between a client application and server using an HTTP GET request.

In the example above, the client sends a URI (Uniform Resource Identifier) within a GET request method to retrieve information about homes 123 and 167 from the fictitious `coolhomes` API.

POST request

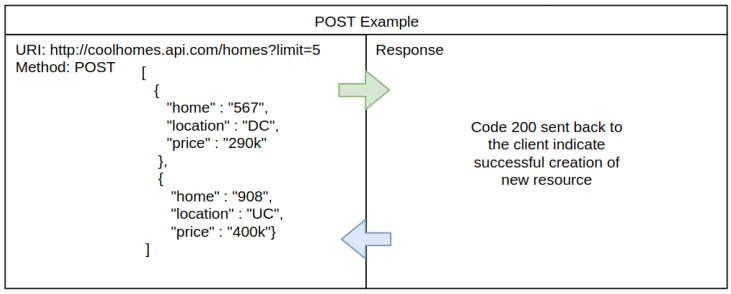

Figure 3. The flow of information between a client application and server using an HTTP POST request.

In the above POST request example, the client seeks to create two new resources on the server. To do so, it sends the URI and details for these new resources along with a POST method. In this case, we create homes 567 and 908. The server responds with a code `200` to indicate the request was successful.

DELETE request

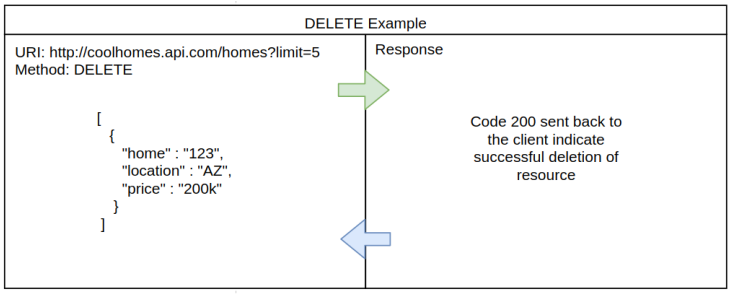

Figure 4. The flow of information between a client application and server using an HTTP DELETE request.

In this DELETE request example, the client deletes home 123. A code `200` is returned to the client to indicate the request was successful.

PUT request

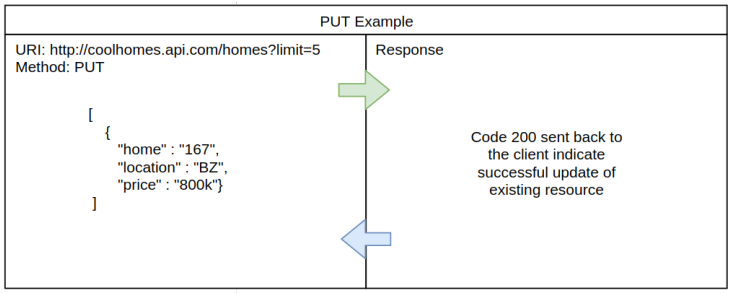

Figure 5. The flow of information between a client application and server using an HTTP PUT request.

In this PUT request example, the client updates home 167 with a new price. As seen before, this entails sending the appropriate URI along with a PUT request method. Here the server also sends back a code `200` to indicate a successful request.

Alternatives to REST

So much is said about REST APIs, you can be forgiven for thinking that there are no other software methods for sending and receiving data. This is not the case. Some common alternatives to REST APIs are:

- GraphQL
- Falcor
- gRPC
- Apache Thrift

GraphQL is probably the most preferred alternative to REST APIs. If you are interested you can find more information about the properties of different types of APIs here.

2. IoT devices and ingestion architectures

Now that we have touched on some of the methods available for data sharing, we can delve into IoT devices and ingestion architectures. When designing a system, we need to keep in mind where our data may be coming from and what type of architectural pattern will make this process most efficient.

What are IoT devices?

IoT, or *Internet of Things*, describes a network of physical objects that have sensors, software, and other technologies embedded within them. IoT devices can take many shapes and perform

many different functions. An IoT device can be defined as any device that is connected to the internet and can send and/or receive data. With a growing volume of both devices and the data they exchange, the aim is to ensure that interconnection and exchange over the internet are simple and efficient. This growth is a primary contributor to the massive increase in data available for data engineers, software engineers, and data scientists today.

Some examples of IoT devices include:

- traditional PCs and laptops;
- mobile phones;
- intelligent appliances;
- heart monitoring implants;
- pressure and flow measuring devices inside piping; and
- sewer-level depth measuring devices inside manholes.

IoT devices typically send data through tightly constrained high-latency environments. For example, a sensor in a manhole may rely on a cellular signal for communication, which is bound to vary in strength and won't necessarily be the fastest.

In other cases the data may be sent through low-latency environments, requiring that millions of devices can connect correctly and data be ingested rapidly. An example of this may be on a petrochemical plant with sensors monitoring a reactor's temperature, pressure, etc. In this case, the sensors would be connected to the plant's network with low latency. Here the sensors must send their data immediately (i.e. in real-time) so that a control engineer can monitor the reactor conditions.

Proper planning and architectural design are required so that the highly constrained system will work correctly.

Ingestion architectures

The traditional 3-tier model

When designing a traditional software architecture model, it's important to understand that the final forms of a solution are highly dependent on business and application requirements. Here, we'll simplify the traditional architecture into the canonical 3-tier model that includes a database, the application, and presentation of this application. Figure 6 illustrates this oversimplification of traditional architectural design.

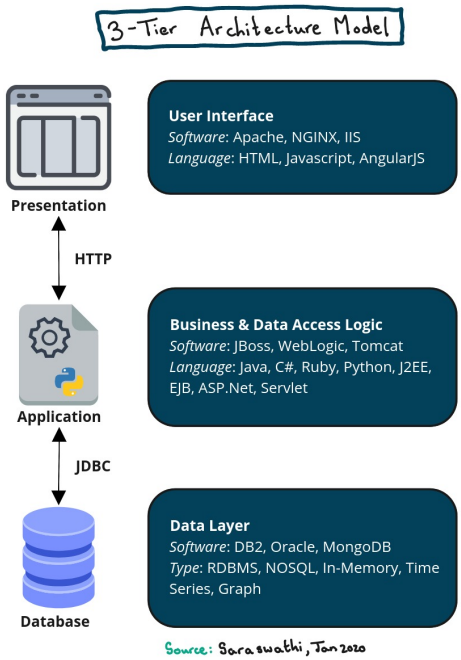

Figure 6. A representation if the traditional 3-tier architectural model used in building software applications. This model is incapable of scaling to the needs of big data without incurring significant costs to an organisation.

Looking at Figure 6 in more detail, we can see that the database connects to the application via Java Database Connectivity (JDBC). A JDBC is a software component that enables the communication between an application and a database via a Java API.

Next, we can see that the back-end of the application communicates with the front-end (presentation) via HTTP requests. These would be the API calls given to retrieve specific data via business logic rules written in the back-end.

Along with each section of this simplified architectural model, we can see the associated software and languages commonly used to implement this type of system.

Confronting big data challenges

In dealing with big data, the primary constraints of the traditional architecture are those surrounding the database. Big data components are introduced in cases where traditional database systems cannot handle the volume of data, as it is either too large or its required read/write transformations too complicated.

So, how do we decide when we need big data components? The threshold at which an organisation enters the realm of big data depends on the competence of its technology practitioners, internal users, and its current systems and tools. Another factor is operating budget, with wealthier organisations having the ability to simply scale their current compute or storage resources without investing in big data technologies. That said, it is good practice to build a system that does not require the brute-force approach of increasing storage and compute resources indefinitely.

Integrating IoT devices: Event-driven architectures

A central theme when forming IoT solutions is the use of event-driven architectures. An event-driven architecture is built around the assumption that multiple processes can be triggered at the same time, and the architecture should be able to handle these processes running simultaneously.

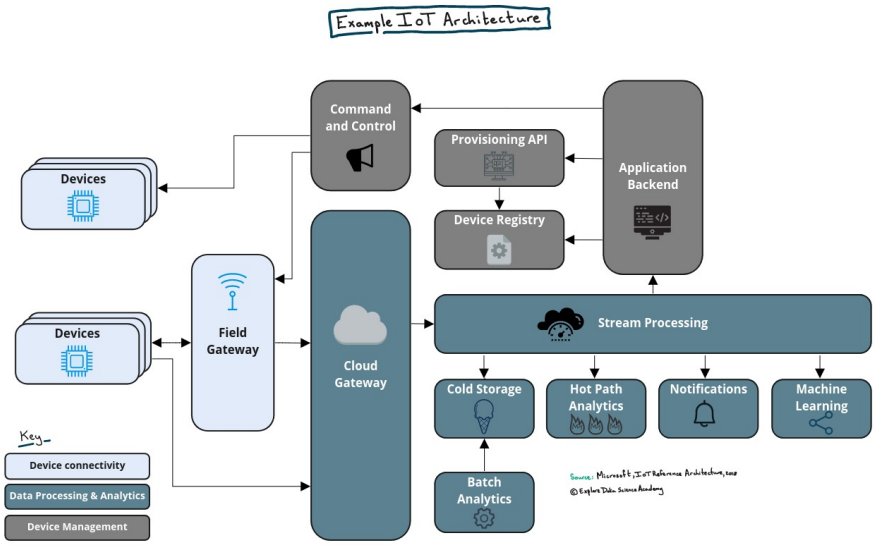

Figure 7. A representation of the various architectural components typically used in an event-driven IoT system.

In Figure 7, the **cloud gateway** is responsible for ingesting device events at the cloud boundary. It uses a reliable low-latency messaging system.

Event data may be sent directly to the cloud gateway, but often it gets sent through a **field gateway** beforehand. This is a specialised device or piece of software that is usually located with the IoT device and receives the events to be forwarded to the cloud gateway. The field gateway may also be responsible for doing pre-processing on the data, such as filtering, aggregation, or protocol transformation.

Once the event data has been ingested it will go through one or more **stream processors**. This will route the data to storage, analytics, or further processing.

Some common types of processing include but are not limited to the following:

- **Writing the event data to cold storage**, after which batch analytics may be performed. Cold storage is cheaper but not as easily accessible when compared to hot storage. Its usage is a good option for data that do not need to be accessed frequently.
- **Hot path analytics**, which refers to analysing data in near real-time or through streaming. This would be necessary for operational decisions such as detecting anomalies, recognising patterns over rolling time frames, or triggering an alert system if certain conditions are met.
- **Non-electric messaging** from IoT devices such as an alarm or a notification.
- **Machine learning**, where data are fed into a model that, for example, forecasts water demand in a water supply network.

The grey boxes in Figure 7 are not directly related to IoT event streaming but are included for a comprehensive description of the IoT architecture.

- The **device registry** is a database or location where all the data about the devices are kept. This includes the device IDs as well as metadata such as location or date of installation.
- The **provisioning API** is an external interface that is used for provisioning and registering new devices.
- In some cases, it may be necessary to be able to **command and control** messages that need to be sent to registered IoT devices.

Internet gateways

In reflecting on our learnings around event-driven and IoT architectures, one fundamental component that we need to focus on is the internet gateway. Consider how, in an IoT environment, hundreds or even millions of devices may seek to send their data into the system. Such data need to be correctly handled, routed, translated, and screened upon reception. These tasks are the responsibilities of an internet gateway.

An internet gateway is a network node that can connect two networks using different internet protocols for communicating, acting like a pitstop for data that is on its way to/from another network. Internet gateways can take many different forms depending on the combination of software and hardware being used to form a system, as well as if they are set up in a home environment or a cloud server.

By definition, internet gateways are located at the edge of a network and frequently incorporate firewalls. A firewall is made up of rules to include or exclude certain IP addresses that will keep out unwanted traffic and only allow traffic specific to an application or network environment.

In a **home environment**, an internet gateway usually takes the form of the internet service provider (ISP) with the hardware being your modem/router. In this case, your internet gateway provides access to the entire internet (unless you're in North Korea) by allowing you to connect to your ISP's network.

In the case of your internet gateway being a computer server or **hosted by a cloud service provider**, the internet gateway will act as a firewall and a proxy server. The proxy server will make sure that the physical server can handle your online data requests.

Let's look at an example of an internet gateway using AWS.

Figure 8 shows an example of an AWS Virtual Private Cloud (VPC). Here we can see the VPC has two availability zones â€“ one private and the other public. The router determines where network traffic will be directed, and the internet gateway only allows certain internet traffic into our system based on its configured rules.

Generally, the internet gateway would be configured while creating the VPC, where we would create rules to include or exclude certain IP addresses.

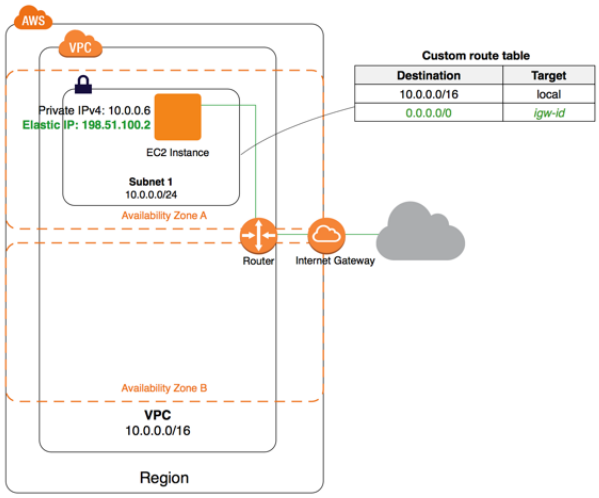

Figure 8. An internet gateway forms an interface between a private network and the public internet.

Edge computing

Another aspect of our architecture design that is helpful to understand, and which relates to both IoT and general ingestion systems, is edge computing.

> **Edge computing:** A distributed computing paradigm that brings computation and data storage closer to the location where it is needed, to improve response times and save bandwidth.
> - Science Direct

Edge computing has particular relevance in scenarios where users are located far from where our application is hosted.

Say, for example, we have a cloud-hosted web application and the data centre region is specified as Berlin, Germany, but our users are in Belgium and Poland. Here, we would make use of servers on the edge of the network (in other words, physically near to the edges of Germany) to bring our application and its supporting resources closer to the users. This helps to improve performance and user experience for these *edge* users. Performance improvements might include speeding up the display of real-time ingested data or sending user-inputted data to a central location.

Figure 9. A geographic overview of Germany and its interior regions. Source:Lonely Planet

The field of edge computing originates from content delivery networks (CDNs) that were created in the late 1990s to serve video and web content to servers that were physically located closer to users.

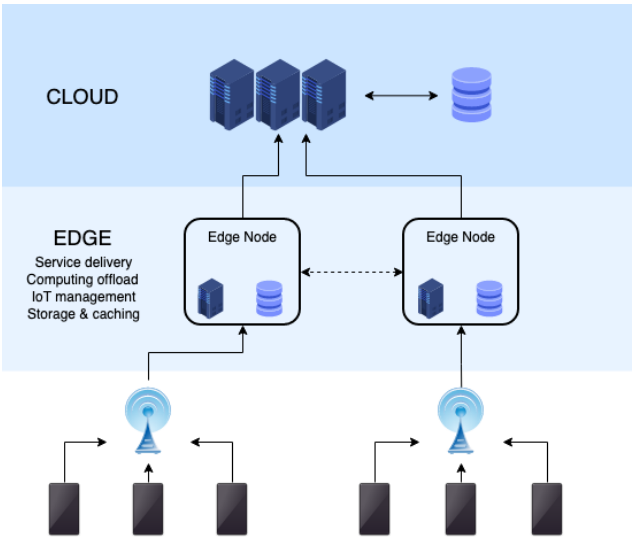

Figure 10. A representation of the hierarchical nature of edge computing. Image by Anonymous @Wikipedia

Figure 10 shows a typical example of edge computing in operation. Starting from the bottom of the graphic, we have multiple smartphones (representing users on our application) connecting to various cell towers. One level up we have the edge servers, followed by the cloud servers.

In our example, the **edge servers** would contain user-specific information to help speed up the delivery of information they are trying to get from our mobile application. Importantly, the edge servers would be located closer to the cell towers than the cloud servers.

The **cloud servers** themselves will contain the bulk of our application's back-end and data storage, such as historical data or data that are not accessed as frequently by the users.

3. Forms of ingested data

Having covered various types of data ingestion methods and related architectural components, let's progress by considering some examples of data that we may need to ingest using data pipelines.

Logs

A log file contains information about an event that happens regularly. This might be in the form of one file per event or multiple events in a single log file.

Let's consider an example where we create log files to track the history of all the API requests made to a website that we own. A single log record is shown below, giving the details of the API call. In this case, the log record starts with the Client IP. A schema has also been provided.

A single log record

```
29.48.17.65 - - [21/Dec/2015:10:19:53 -0800] "GET /apps/cart.jsp?appID=8345 HTTP/1.0" 200 5040 "http://www.mcdermott.com/" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/5360 (KHTML, like Gecko) Chrome/13.0.853.0 Safari/5360"
```

Schema alignments

- Client IP : 43.23.12.78
- [Date Time] : [21/Jan/2021:11:24:5 -0930]
- "Request URI HTTP Version" : "GET /apps/cart.jsp?appID=9034 HTTP/1.0"
- Status Code : 200
- Bytes : 6040
- Referrer : "http://www.facebook.com/"
- User Agent : "Chrome/5.0 (X11; Mac OS x86_64) AppleWebKit/5360 (KHTML, like Gecko) Chrome/13.0.853.0 Safari/5360"

Field breakdown

- *Client IP* : Client internet protocol address.
- *[Date Time]* : Timestamp of the request.
- *"Request URI HTTP Version"* : API method, the URI (identifier), and the HTTP version.
- *Status Code* : The code returned from the server; 200 indicates a successful interaction.
- *Bytes* : The number of bytes transferred.
- *Referrer* : The originating address of the request.
- *User Agent* : The browser and operating system from where the request comes from.

A complete dump of this example log can be foundhere. Data Source: (StreamSets Inc, 2018)

Devices

As mentioned previously, a wide range of IoT device types currently exists, with almost anything that is connected to the internet being classified as an IoT device. The data format and type received from an IoT device are each dependent on the function of the device. One may receive a .jpeg or .mp4 file from a drone that is monitoring crop damage. Or, for example, a.json file from a device monitoring a pump's operational uptime may look like this:

```
{
  "Links":{},
  "Items":[
     {
      "PumpId":1909,
      "RunTime":1265,
      "Units":"seconds"
     }
    ],
    "UnitsAbbreviation":""
}
```

Looking at the fields of this file, we can see quite a few extra keys that are not necessarily of interest. This is a common occurrence with IoT devices, where the actual data that we need is nested within other keys. In this example, we can see that the keys of interest are PumpId, RunTime, and Units. Having received this data, the next step would be to extract the values associated with the keys we are interested in.

> As a simple exercise, try and perform this extraction task using Python or a command line tool of your choice.

Collecting data via application APIs

An application such as a website may display important data that need to be processed. In many cases, this application will be hosted in some sort of cloud server where an API enables access to its data and/or database. Depending on how complete the API is, we can usually access any data that would logically need to be made available. For example, if we're collecting pump run-time data, it makes sense to have a method that allows for this data to be retrieved from the application's database.

Another example of data that may need to be processed from an application is payment data. This data need to be picked up and processed immediately for records to be kept consistent and accurate. We may encounter data from an application in the form of .json, .xml, or even log files.

Once the data have been successfully pulled using the application API, it can undergo further processing or be saved directly to a data lake and picked up at a later stage.

---

4. Exercise: Connecting to an API

Introduction

Let's get our hands dirty by connecting to a live API and running some queries to extract data. We are going to use theAccuWeather website application and create an account on their developer website to gain appropriate credentials to access the site's data.

Dependencies

We will use Python and two extra packages to query the API and extract data:

- urllib library
- json library

Outcomes

To complete this exercise, you'll need to follow the steps provided below, which includes modifying the associated Python script to retrieve data from the AccuWeather API for a location of your choice. In doing this, you should get a list in a format similar to the example file we provide called api_output.md.

As an extra challenge, once you have successfully received a response from the AccuWeather API, try and see if you can extract specific elements from the JSON response such as the Temperature.

Exercise instructions

1. Head over to AccuWeather's developer website and create a free account.
2. Once you have set up your account, you will need to create an application to get API credentials to connect to the AccuWeather system. From the AccuWeather API landing page, log in and then click "My Apps".
3. Click "Add a new App" and fill in the relevant details. Since this is not going to be an actual application, don't worry too much about the options you select. Keep them as the defaults and give your application a name. Select Python as the programming language of development.
4. Once you have added your app, open it by clicking on its name within the "My Apps" dashboard. To retrieve your API key, select the "Keys" tab, and copy the value associated with the "API Key" field. This key will be used as the identifier to give our Python script access to the system.
5. Find the location that you are interested in getting the data for by searching for ithere.
6. Once you have searched for and loaded a location, the location ID (which you'll use in later steps in conjunction with your API credentials) can be found as the numbers appended to the end of the AccuWeather URL.

   For example, if you were searching for the current weather in Cape Town, you'd be directed to the following URL, with a corresponding location ID of 306633:

   https://www.accuweather.com/en/za/cape-town/306633/weather-forecast/306633
7. Move on to the api_exercise.py script found here. This script provides a practical walk-through of the code needed to pull data from AccuWeather via its API. Reference documentation for AccuWeather's API can be found here.
8. Complete the instructions within the exercise script to retrieve JSON-formatted data representing the current weather conditions at the location ID retrieved in step 5.
9. Once you've successfully run the script, follow the exercise challenge by manipulating the decoded API data payload to extract various data fields. If you're feeling ambitious, you can further challenge yourself by:
   - writing a workflow that calls the API and writes specific fields in the received data payload to a.csv or .txt format; and
   - automating this workflow to run every two hours as a background task.

Conclusion

We introduced methods for ingesting datasets. We started by introducing REST APIs, their history, and common commands used in their operation. Following this, we briefly discussed some common alternatives to REST and provided resources for further exploration. We then discussed IoT devices and their associated architectures, where concepts such as internet gateways, edge computing, and resulting log data were introduced. Lastly, we set up a basic script to query a weather API and extract interesting data.

References

REST APIs

- Alternatives for REST APIs
- A brief introduction to REST
- A helpful comparison between REST and SOAP
- How to use an API with Python

IoT and Ingestion Architectures

- Four architecture choices for application development in the digital age
- Big data architecture components
- Ingesting log files into ElasticSearch