Managing events and APIs from a unified environment: 300-level live demo script

Introduction

It's becoming increasingly common to need to expose data both synchronously, using APIs, and asynchronously, using events. This enables us to cater to the unique needs and characteristics of the different consumers of the data. Some consumers need to retrieve or change the data at will - and will therefore require APIs. Others are more focused on just being notified of changes in the data in a timely manner, which is a good use case for events. Even a single consumer may need to retrieve data in different styles for different scenarios. As a data provider, however, you shouldn't need two separate systems to achieve this. In this demo we'll look at how events can be shared and governed in the same way as APIs.

The scenario focuses on an airline's flight information system that makes data available over an API to airport displays such as 'Flight Boards' showing departures and arrival times. The airline also makes this data available to other applications via API management. In our example, we will look at an airline mobile app currently using the flight information API. It has an opportunity to commercialize flight-delay events as part of a co-marketing agreement with a major restaurant in the airport. However, it can only do this if it can receive the events in real-time. We will show how Apache Kafka events can also be surfaced in the same catalogue as the APIs such that they can be discovered and used by the airline mobile app.

Let's get started!

1 - Existing API-driven information system - without events

1.1 - Explore the Flight Information Manager UI

Narration
Let's first take a look at the flight information system itself so we understand the data we're dealing with. The user interface for the flight information system is known as the Flight Information Manager. It shows us the current flights, and also provides us with a mechanism to add a delay to the anticipated departure time of a flight.

Action 1.1.1
- Show the Flight Information Manager UI, as highlighted in the narration above.

Narration
The Flight Information Manager uses a REST API available on the Flight Information Data Store. This API is only directly available to the Flight Information Manager user interface since it has the ability to perform sensitive functions, such as delaying the arrival time of a flight.

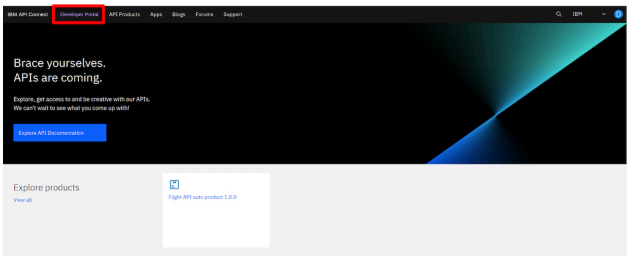
1.2 - Explore the existing REST API

Narration
There are other systems, such as the Flight Boards around the airport that also need access to flight data but do not need access to the whole API. For example, the Flight Board should not have access to the API function to 'delay' a flight. An API management capability (IBM API Connect) is used to expose selected data and functions from the API in a Developer Portal. The developers of other applications can browse the various APIs available at the airport, and self-subscribe to use them in their applications. This means that their usage of the API can be tracked, controlled (e.g. rate limited) and indeed revoked. Let's take a quick look at the exposed REST API for flight information.

Action 1.2.1
- Go to the API management Portal and show the exposed REST API for flight information, noting that there is only a GET operation.

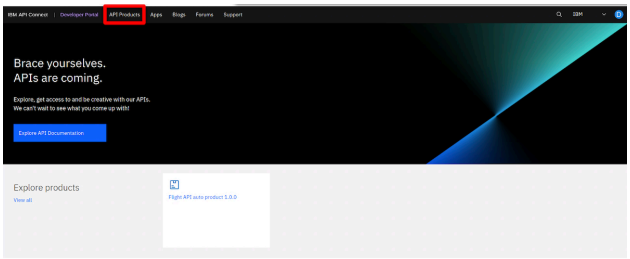
Action 1.2.2

- Show the IBM API Connect **Developer Portal** tab in the browser.



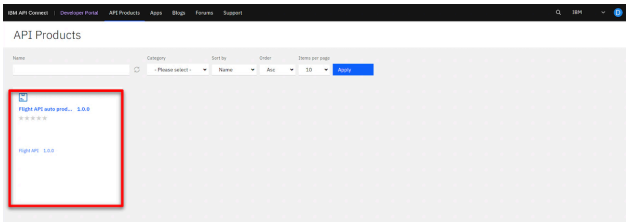
Action 1.2.3

- Click **API Products.**



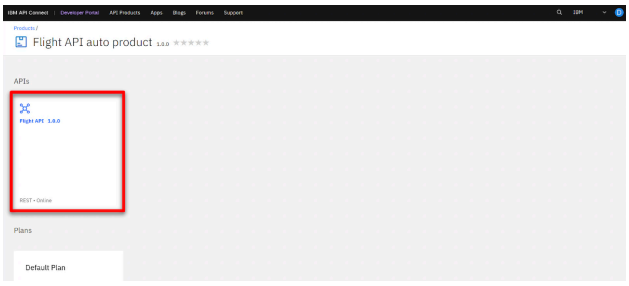
Action 1.2.4

- Click **Flight API**.



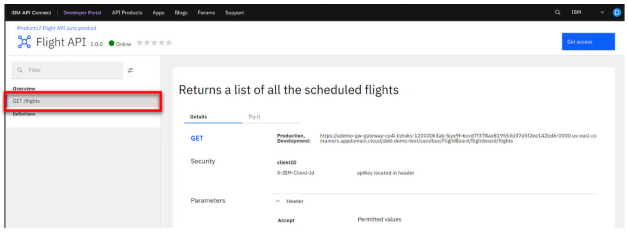
Action 1.2.5

- Click **Flight API 1.0.0**.



Action 1.2.6

- Click the **GET /flights** operation.



Narration

Note that we've been able to explore this API right down to what operations are available, and even example data that would be returned. However, we cannot make calls on this API because we have not yet requested access to it. API Connect ensures only known consumers of the API can use it.

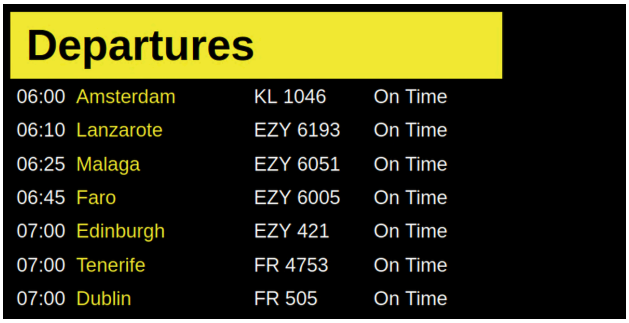
1.3 - Display the flight board

Narration

The Flight Board is one of the consumers of this flight information API. It retrieves the data every five minutes to get the latest flight information.

Action 1.3.1

- Bring up the Flight Board UI.



1.4 - Explore the update rate of the Flight Board

Narration

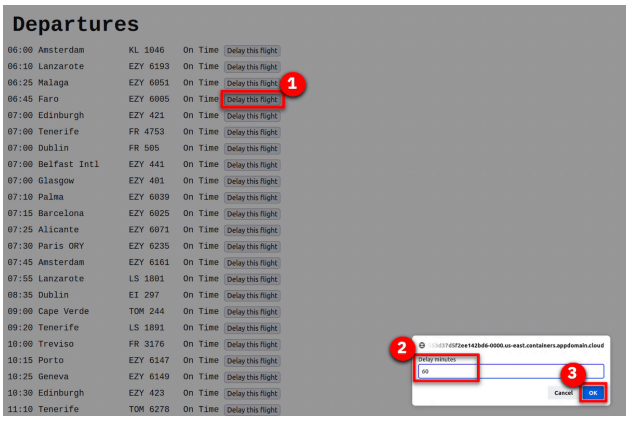
Note that if we delay a flight using the Flight Information Manager, although it shows the new time immediately in that user interface, the Flight Board only picks up the change when it next polls the API, several minutes later.

Action 1.4.1

- Show both the Flight Board (ie, the display in the airport, shown on the left) and the Flight Information Manager (ie, the master flight information record, shown on the right) on screen at the same time. Both screens are showing the same information for now.

Action 1.4.2

- On the Flight Information Manager, click on one of the **Delay this flight** (1) buttons, enter **60** (2) for the minutes to be delayed, and click **OK** (3).



Action 1.4.3

- Show that the flight departure time has been updated on the Flight Information Manager only.

Action 1.4.4

- Remember that the board updates every five minutes. If required, wait to show that the Flight Board is updated.

Narration
Clearly, we could reduce the poll interval to something that might be more responsive, but as we reduce this interval, we are increasing the load on the Flight Information Data Store. Note that there could be many other systems also requiring this information in a timely fashion (eg., taxi companies, ground crew systems, airlines). If they were all to poll with short time intervals, the Flight Information Data Store would quickly become overwhelmed. The Flight Information Data Store could of course implement performance measures such as caching, but it would end up having to cater to constantly increasing demands of consumers, in terms of both performance, and availability.

NARRATION TIP
This is a good time to discuss how common this situation is across scenarios in other industries. APIs are a commonly used way to provide data, but do not make it easy to receive data changes in a timely way. Examples might include fraudulent payment attempts, exceptional stock price movements, fire or other hazard alerts, refrigerator temperature thresholds etc.

2 - Using events to deliver real-time information

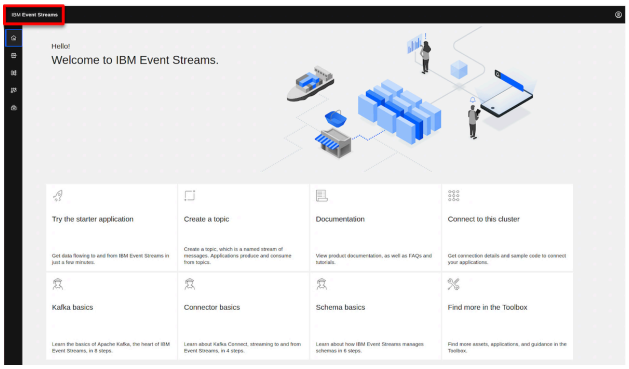
2.1 - Introduce Events, Kafka, and IBM Event Streams

Narration

Modern applications are turning to Apache Kafka to distribute information using a publish/subscribe pattern. In Kafka's terminology, a topic is a stream of events related to the same subject. Events are published to specific Kafka topics, then consumers decide (via subscription) which topics they would like to receive events from. Publish/subscribe capabilities are certainly not new and messaging products have provided this pattern for decades. However, Apache Kafka has risen to popularity due to its ability to retain an event history, making it well suited to certain application patterns. IBM Cloud Pak for Integration provides both messaging (IBM MQ) and also a production strength implementation of Apache Kafka (IBM Event Streams). IBM Event Streams provides a Kubernetes operator that enables rapid deployment of Apache Kafka clusters, including a graphical user interface that simplifies familiarization.

Action 2.1.1

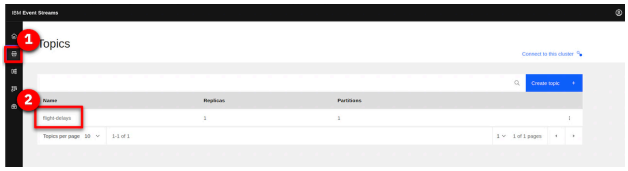
- Show the **IBM Event Streams** tab in the browser.



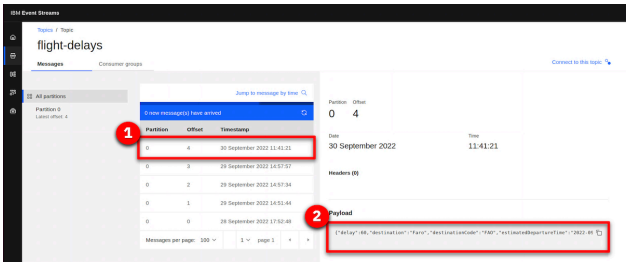
2.2 - View events on Kafka topic

Narration
For the simplicity of the demo, we've already configured our flight information system to publish events to a Kafka topic when we delay a flight.

Action 2.2.1
- Navigate to **Topics** (1). Click **flight-delays** (2).



Action 2.2.2
- The events already emitted will be displayed. Select the top event (1) and show this corresponds to the flight you previously delayed (2).



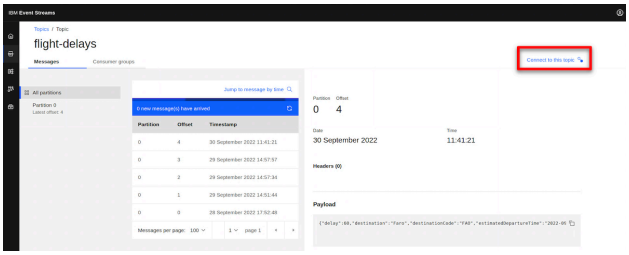
Action 2.2.3
- Show that as you delay additional flights these display in the event stream.

Narration
Note that the events for each delay we create are present in the event history, and there is a very clear representation of their sequential order based on the 'Offset' value. These will remain here, regardless of whether subscribers read them or not, and this is one of the key differences between Apache Kafka and messaging capabilities such as IBM MQ. Events can only be removed administratively, either by archiving those older than a certain time period, or a more sophisticated mechanism known as 'log compaction' which is useful when at least one event for each data record must be preserved.

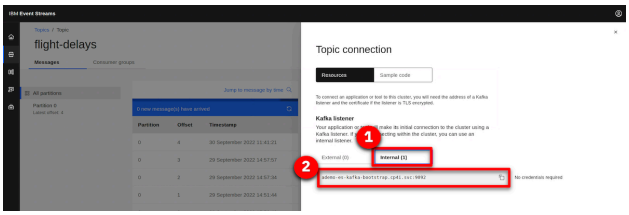
2.3 - Connect an application to the Kafka cluster

Narration
Applications consuming events from a Kafka cluster need to know the location of the bootstrap Kafka broker for initiation of their connection. You will notice that that IBM Event Streams only provides ‘internal’ connections. That means you can only connect from within the OpenShift cluster. This connection can be used, for example, by the flight information system to publish events.

Action 2.3.1
- Click **Connect to this topic**.



Action 2.3.2
- Click **Internal (1)** (1), then copy the **endpoint information** (2). We will use this later when configuring IBM Event Endpoint Manager. Note that we do not need to download any certificates because the demo Kafka cluster has security switched off for simplicity.



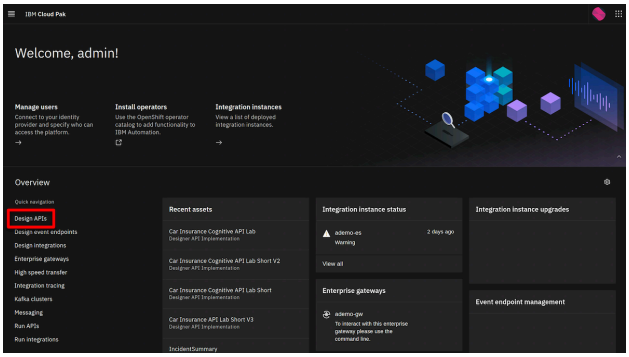
Narration
The consumers of flight delay events (such as the Flight Board and later, the airline mobile application) will be external to the OpenShift cluster. We could do basic external exposure by configuring an ‘external’ listener in the screens we’ve just explored, but this provides only limited control when we are exposing Kafka to consumers beyond our immediate sphere. We are going to look at a much more powerful way of exposing a topic, using IBM Event Endpoint Manager. Since this is in the same OpenShift cluster as this Kafka cluster, the internal listener will be sufficient as IBM Event Endpoint Manager will do the external exposure for us, then route to the internal connection.

3 - Making events available to external consumers

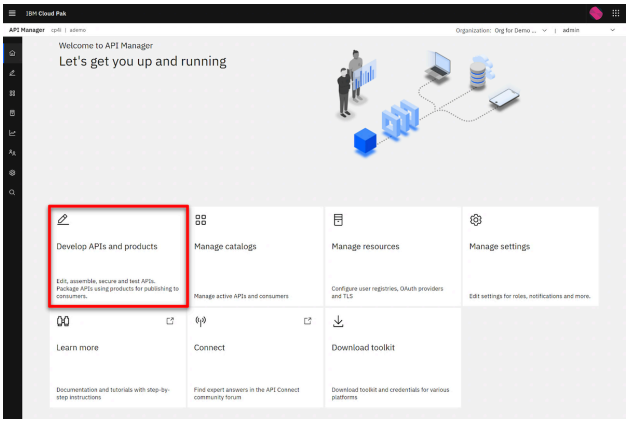
3.1 - Manage the Async API

Narration
IBM Event Endpoint Manager performs exactly the same role for asynchronous interfaces as IBM API Connect does for synchronous APIs. It allows us, for example, to make a selection of Kafka topics available in a catalogue and display them in a portal such that a developer can easily discover and subscribe to use them. It then protects the actual Kafka cluster by controlling access to it via a gateway. Let's add our Kafka topic to the IBM Event Endpoint Manager catalogue.

Action 3.1.1
- Open the Cloud Pak for Integration Platform Navigator tab and click **Design APIs**.

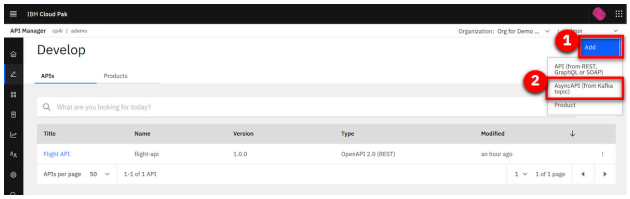


Action 3.1.2
- Click **Develop APIs and products**.



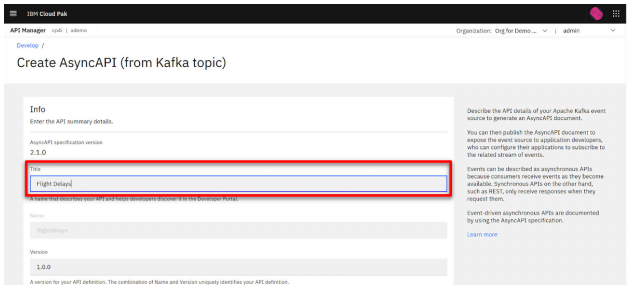
Action 3.1.3

- Select **Add** (1) in the top right, and select **AsyncAPI (from Kafka topic)** (2).



Action 3.1.4

- Enter **Flight Delays** in the Title field.



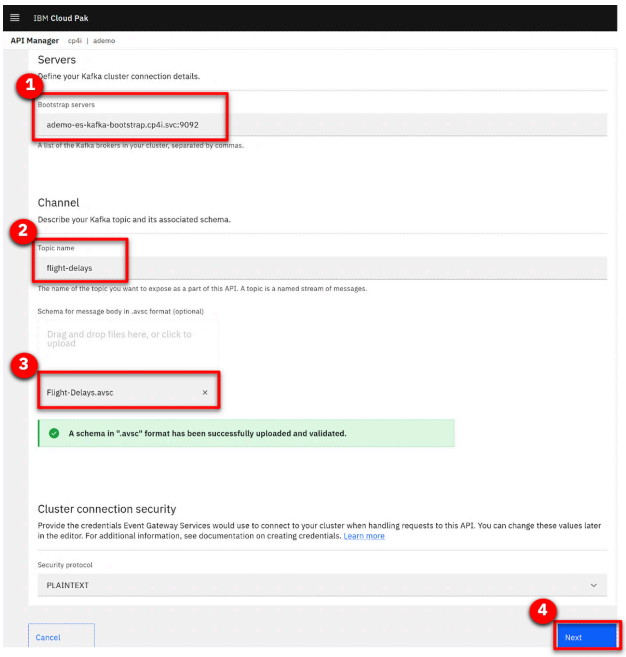
Narration

IBM Event Endpoint Management can provide access to topics on multiple Kafka clusters via a single ‘event gateway’. Consumers therefore only have to connect to one bootstrap server (the event gateway) with one set of credentials regardless of which Kafka cluster their topics reside on.

As we add the topic to the catalogue for external exposure, we also choose to provide a schema for the event data. This ensures that all consumers of the events know what data model to expect from this topic.

Action 3.1.5

- Enter the Kafka listener URL from the earlier step in the **Bootstrap servers** field (1) in the Servers section. In the Channel field enter the Topic name **flight-delays** (2). Upload the **Flight-Delay.avsc** (3) schema that we downloaded from here during the Preparation steps. Click **Next** (4).

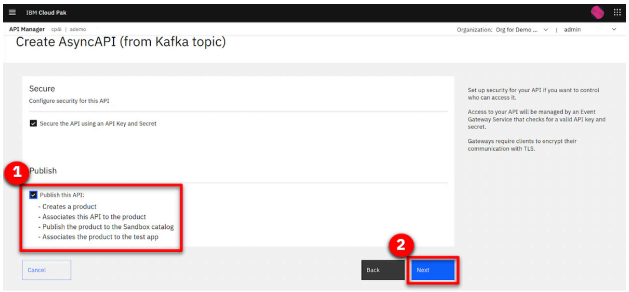


Narration

We will choose to publish the topic straight away, which will automatically create a Product and publish that within the Sandbox catalog. By default it will secure the topic in the same way we would a synchronous API, using an API Key and Secret. This enables us to hide what might be a more complex security model used on your Kafka cluster (such as mTLS). Note that AsyncAPIs exposed by the event gateway are always protected with TLS communication, even if the underlying Kafka broker is not, as is the case in this demo.

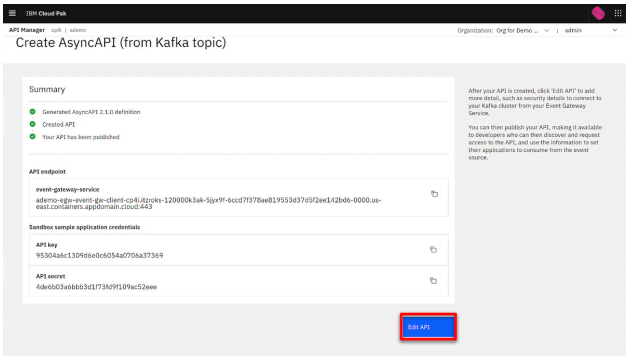
Action 3.1.6

- Check **Publish this API** (1) and click **Next** (2).



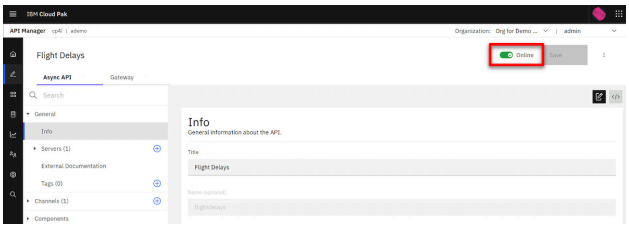
Narration
The next page shows the API Key and Secret that are needed to access the Async API from your Kafka consumer. These will take the place of the username and password in your consumer properties.

Action 3.1.7
- Click **Edit API** to continue.



Narration
The next page is where you can enter a vast array of optional detail about the API, that will ultimately be published to the catalog. However, we do not need to add any further details at this stage.

Action 3.1.8
- The next screen is the Edit API screen, but you do not need to do anything here. Simply ensure that the **online switch** is green, indicating the API is online.



Narration
Managing granular access to topics would normally require a Kafka administrator to create and maintain multiple access control lists (ACLs). IBM Event Endpoint Manager enables consumers to self-administer their own access to topics. The Kafka administrator need only set up access between the Kafka cluster and the event gateway. From that point onward, the person deciding which topics the consumers can subscribe to no longer needs to be a Kafka specialist.

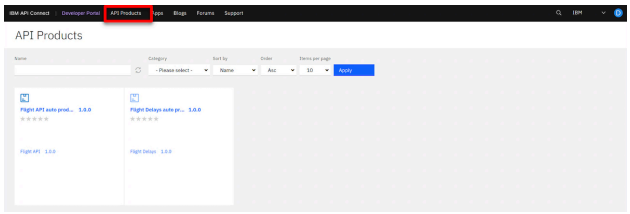
4 - Consuming real-time events in an application

4.1 - Discover the AsyncAPI from an app developer's perspective

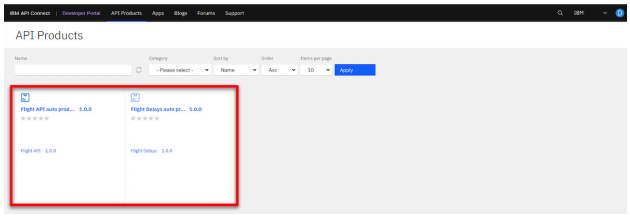
Narration
Let's get back to our airport scenario. There is a restaurant at the airport that is struggling to fill its tables. They have discovered that the first thing people do when their plane is badly delayed is to go find somewhere to eat, so they see this as an excellent marketing opportunity. They are prepared to invest in advertising, but it needs to be extremely timely if it's going to capture this particular opportunity as people make decisions within seconds when they see their flight delayed. They need to get a notification to customers immediately, perhaps before the delay is even posted on the departure boards. They negotiate with the airline asking if they can place an advert on the mobile application alongside the delay notification, but they are only prepared to pay for an advert if it is delivered within a handful of seconds of the delay being published by the flight information system.

The developers of the airline app investigate the design of the current delay notification in their app. It uses an API to retrieve flight delays from the flight information system, but the API only polls once every five minutes, to avoid undue pressure on the back end database. They come to the Portal to see if there are alternative ways to find out about delays. They find the REST API for flight information that they're currently using. They then discover there is also an AsyncAPI for 'Flight Delays' which would enable them to be immediately notified, so they explore how they could incorporate retrieving events from Kafka into their mobile application.

Action 4.1.1
- Go to the API Connect Developer Portal tab and click on **API Products**. Note that if you are already on the API Products page, you may need to refresh the browser page as you just added a new AsyncAPI product.

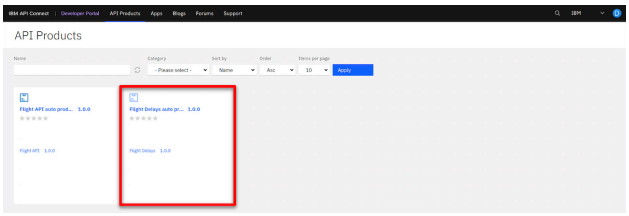


Action 4.1.2
- Explain the two entries, as highlighted in the narration above.

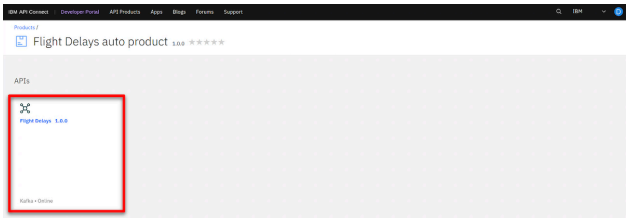


Narration
Note that APIs are gathered together into ‘Products’. You could gather multiple topics under one Product, and they could even originate from different Kafka clusters. Access is then defined at the Product level, so you can easily subscribe to a whole suite of events or APIs in one go. Simple products are automatically created for development purposes and that’s what we will use in this demonstration.

Action 4.1.3
- Click **Flight Delays auto product** (the automatically-generated product we created earlier).

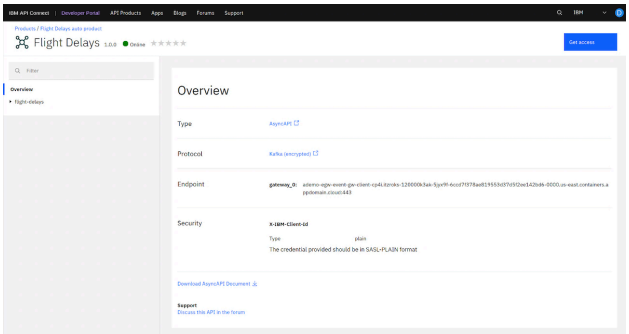


Action 4.1.4
- Click **Flight Delays.**



Narration
This next page will show documentation we provided when we added this API to the catalogue. From this page you will find the values needed for connectivity. Note that the bootstrap server is that of the event gateway, not that of the original Kafka cluster. You will also find some example code samples to simplify application development.

Action 4.1.5
- Explain the screen, as highlighted in the narration above.



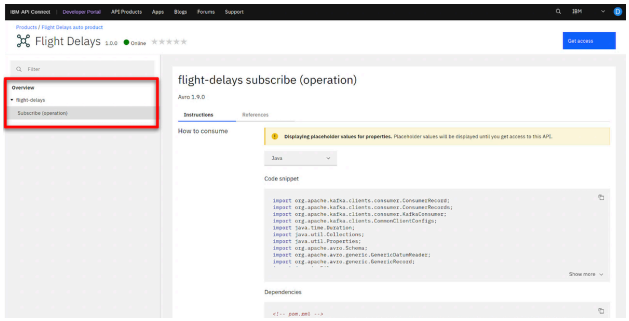
4.2 - Subscribe to the AsyncAPI

Narration

From the detail we've found so far on the API, it looks ideal for providing the notifications for our restaurant mobile app, so we decide to collate the connection details and subscribe to the AsyncAPI. Just to reinforce what's happening here – these are the connection details to the event gateway provided by IBM Event Endpoint Management, not to the actual underlying Kafka cluster. However, the event gateway 'looks' just like a Kafka cluster to the consuming application.

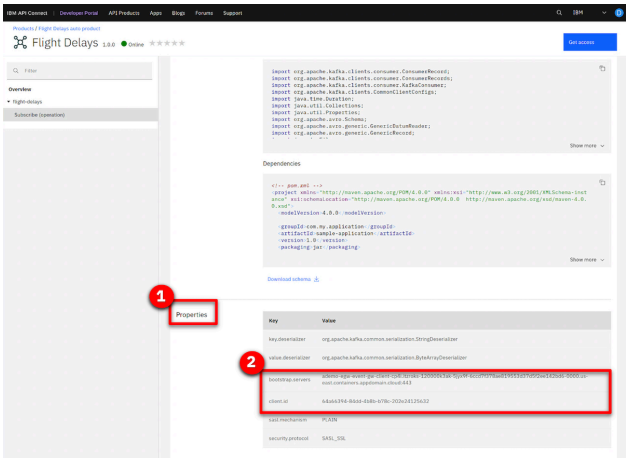
Action 4.2.1

- Click the arrow next to flight delays in the overview on the left, then select **Subscribe (operation)**.



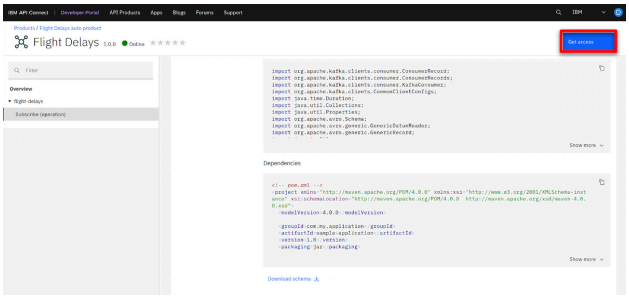
Action 4.2.2

- Scroll down on the right hand side to the **Properties** section (1) and obtain the value for **client.id** and **bootstrap.servers** (2).



Action 4.2.3

- Click **Get access** in the top left. If for any reason you are not logged into the Portal, this is the point at which you will be forced to log in.

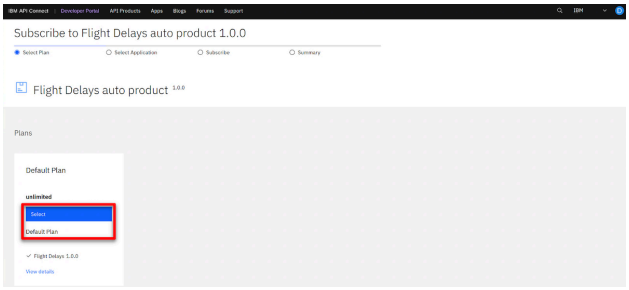


Narration

Products are made available to subscribers through ‘Plans’. This gives us the opportunity to have different policies depending on the plan that is chosen. If APIs are monetized APIs, each plan might have a different cost, or some plans might be only available to selected user groups.

Action 4.2.4

- Select the **Default** pricing plan to continue.



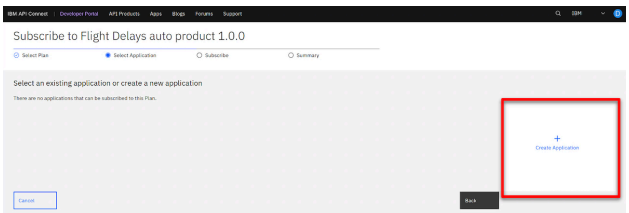
Narration

As a developer, you may be writing several applications, each using different APIs. From a security perspective you might not want all applications you work with to have access to all of the APIs that you have subscribed to. For this reason, you can create an ‘Application’ within IBM Event Endpoint Manager for each of your applications, and individually subscribe them only to the Plans on the Products they need. Each application will get its own unique Key and Secret such that it can be identified when attempting to access the APIs.

At this point you will need to create an ‘Application’ in the portal. This is simply an identity or set of credentials that your actual application will use.

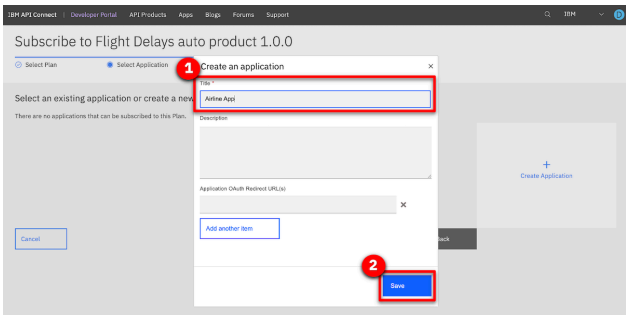
Action 4.2.5

- Click **Create Application** on the right.



Action 4.2.6

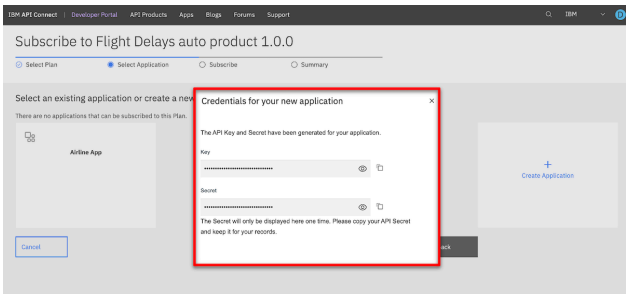
- Name the application **Airline App** (1) and click **Save** (2).



Action 4.2.7

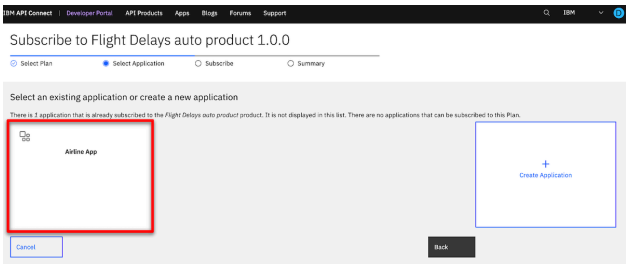
- You will now see a dialog showing the **Key** and **Secret** for this application to access the API.

NOTE: It is very important that you save these, as they cannot be retrieved later.



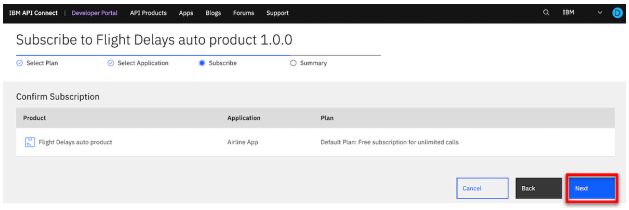
Action 4.2.8

- Close this dialog then select your newly created application on the left.



Action 4.2.9

- Review the details then click **Next**. Your subscription will be complete.



4.3 - Consume the events

Narration
We now have all the credentials we need for our Airline App to subscribe to the Kafka topic via IBM Event Endpoint Manager's gateway. All we need now is an application to consume the events. We're not going to create the actual airline mobile app. We're just going to give you an example, built using the sample code from the Portal, for what the server side of the mobile application would be doing in order to pick up the events.

Recall that the IBM Event Endpoint Manager gateway always enforces TLS encryption on external requests, so our consumer will need to have a copy of the gateway's public TLS certificate. We will download this certificate and store it somewhere that the consuming application can retrieve at runtime.

,

Action 4.3.1

- Install the server certificate from the event gateway endpoint ready for the client application. There is a script to do this:

```
./install-certificate.sh
```

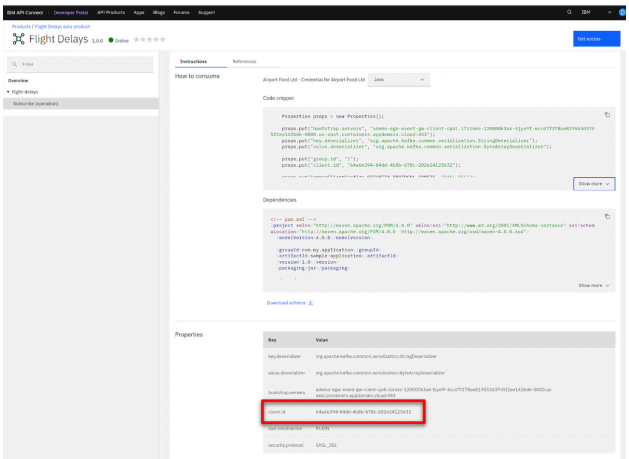
```
callum@callum-Virtual-Machine:~/git/eem-demo/scripts$ ./install-certificate.sh
Downloading certificate
depth=0 CN = ademo-gw-gateway-cp4i
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 CN = ademo-gw-gateway-cp4i
verify error:num=21:unable to verify the first certificate
verify return:1
depth=0 CN = ademo-gw-gateway-cp4i
verify return:1
DONE

Creating keystore
Certificate was added to keystore
secret/eem-client-app created
```

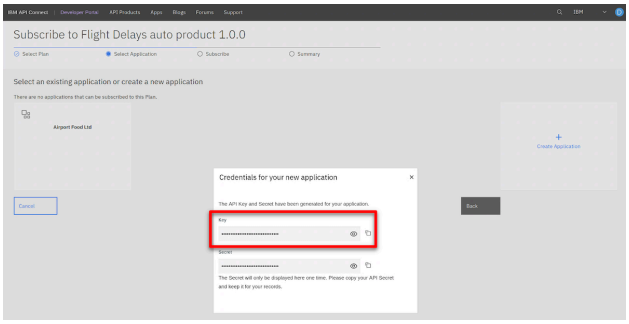
Narration
The example in our demo is simply a java consumer that simulates the way the Airline App would receive Kafka events, such that we can see them arriving in real time.

Action 4.3.2

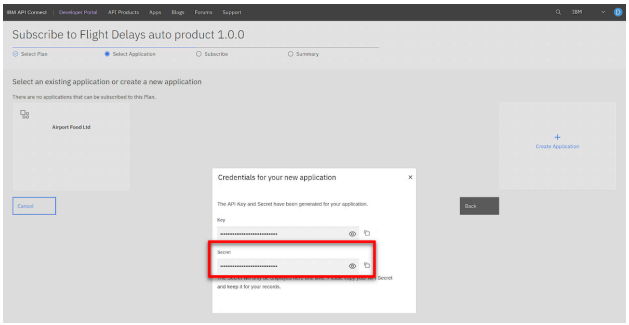
- The script that runs the consumer code itself is called ./setup-client-app.sh and is one of the files you downloaded in the beginning. This takes five arguments:
- 1) **Target Namespace:** this is normally cp4i
- 2) **Kafka Client Id:** this is retrieved from the Developer Portal in the properties section



- 3) **Gateway Username:** this is the API Key that you copied in this step.



- 4) **Gateway Password:** this is the API Secret also shown in the screenshot above.



- 5) **Flight Number:** this is the flight number you want to monitor. This needs to be within quotes, e.g. “EZY 6005”

Action 4.3.3

- Run the following command, replacing each of the parameters:

./setup-client-app.sh \<TargetNamespace> \<KafkaClientId> \<GatewayUsername> \<GatewayPassword> \<FlightNumber>

Here is an example. Remember, this needs to be customized with your own values for each of the parameters:

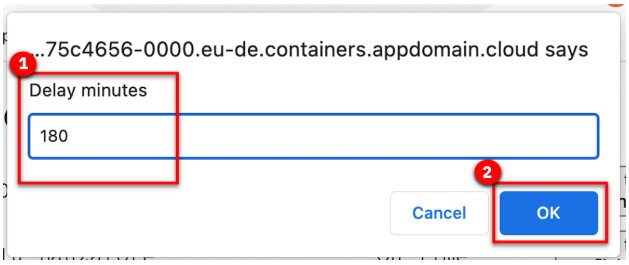
./setup-client-app.sh cp4i 64a66394-84dd-4b8b-b78c-202e24125632 8515b170f10b6937c4ba89f1c6eaf325 746f7e1f89415723d00366acd0574410 “EZY 6005”

This will automatically build a container and run it on Red Hat OpenShift.

NOTE:
When entering the flight number, be sure to type the quotation marks directly in your Terminal window. Copying and pasting from another text editor may result in no delays being reported.

Action 4.3.4

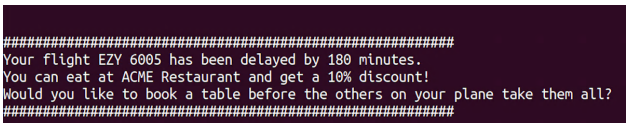
- In the Flight Information Manager, click the button to delay the flight you entered in the consumer app. Enter **180** minutes (1), then click **OK** (2).



Action 4.3.5

- Now return to the terminal window where the script was run, and you will see a message.

  “Your flight EZY 6005 has been delayed by 180 minutes. You can eat at ACME Restaurant and get a 10% discount! Would you like to book a table before the others on your plane take them all?”



Narration

We see the following message, which of course in real life would appear on the airline mobile app alongside the delay notification.

“Your flight EZY 6005 has been delayed by 180 minutes. You can eat at ACME Restaurant and get a 10% discount! Would you like to book a table before the others on your plane take them all?”

Notice that it takes a while before the flight board shows the delay, since it is still using API polling to get flight information.

Summary

In the demo we saw an example of a use case that was better served by listening to Kafka events than by polling synchronous APIs. We were able to react instantly to an event happening rather than wait a full polling interval. Furthermore, we did that without putting any additional load on the back-end system, as the Kafka stream provided full decoupling and is completely independent of the flight delay information. It is clear that there are many advantages to exposing a back end's capabilities not just using a synchronous API, but also with a source of events. Even a single consumer is likely to need to retrieve data in different styles for different scenarios.

A key focus of the demo was to show how events could be shared, and governed in the same way as APIs, extending API management into event endpoint management. The topics were discoverable in a catalogue from which the consumer could self-subscribe to receive events. Access to the events was then securely exposed using IBM's unique event gateway that transparently routes the Kafka protocol to the underlying Kafka brokers.

Thank you for attending today's presentation.