How to Build a Simple Event-Driven Application with Apache Kafka on Heroku

Harnessing the power of real-time data processing? Look no further than event-driven architectures (EDA). This approach excels at handling high-volume data streams, making it perfect for applications that require constant updates and responsiveness. This guide delves into building a basic event-driven application leveraging Apache Kafka on Heroku, a cloud platform beloved by developers.

What is Apache Kafka?

Apache Kafka acts as a distributed streaming platform, enabling applications to publish, subscribe to, and store streams of records (events). It boasts impressive scalability and fault tolerance, making it ideal for mission-critical applications.

Why use Kafka with Heroku?

Heroku simplifies application deployment and management, allowing developers to focus on core functionalities. Here's why it shines for Kafka deployments:

- **Effortless Scalability:** Heroku automatically scales your application as event volume surges, ensuring smooth performance.
- **Simplified Management:** Heroku handles server management and infrastructure, freeing you from those burdens.
- **Familiar Workflows:** Heroku integrates seamlessly with Git, allowing for hassle-free deployment using familiar tools.
- **Rich Add-On Ecosystem:** Heroku offers a vast marketplace of add-ons, enabling integration with databases, monitoring tools, and more.

Getting Started: Prerequisites

Before embarking on this journey, ensure you have the following:

- A Heroku account (Free tier works for getting started)
- Heroku CLI installed (Follow instructions on Heroku's website)
- A code editor or IDE of your choice

Planning Your Application

Before diving into code, take a moment to plan your application. Here are some key considerations:

- **Events:** Define the events your application will produce and consume (e.g., user registration, order confirmation).
- **Topics:** Create Kafka topics (categories) for each event type, ensuring organised data flow.
- **Producers:** Designate the components that will publish events to Kafka topics (e.g., a user registration service).
- **Consumers:** Identify the applications that will subscribe to and process events from Kafka topics (e.g., an order processing system).

Setting up a Kafka Cluster on Heroku

Heroku offers Kafka as an add-on service, making it easy to integrate with your applications. Here's how to provision it:

1. **Login to your Heroku account**:

   ```
   heroku login
   ```

2. **Create a new Heroku app**:

   ```
   heroku create kafka-app
   ```

   Replace "kafka-app" with your desired app name.

3. **Add Kafka Add-On to Your App**:

   ```
   heroku addons:create heroku-kafka:basic-0 -a kafka-app
   ```

   For our example, the basic plan suffices. You can choose a different plan if needed.

4. **Manual Installation (Optional)**:

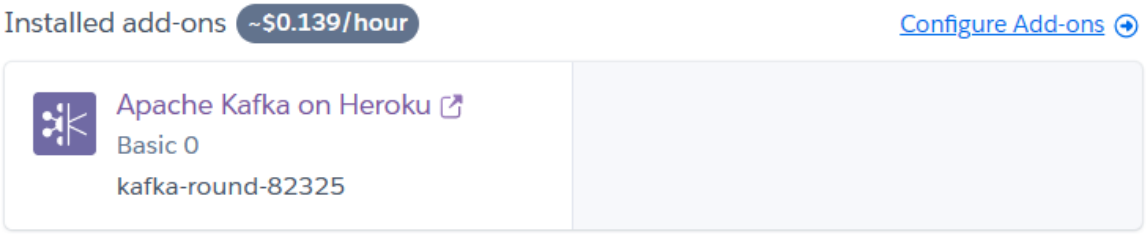
   If the above command doesn’t work, you can manually install the add-on from your dashboard:

   - Go to your application dashboard on Heroku.
   - Go to the "Resources" tab.

- Click on the "Find more add-ons" button.
- Search for "Kafka" and select the desired plan to install.

5. **Confirmation**:

   After adding the Kafka add-on, it should appear on your Heroku application dashboard, like this:



Viewing Kafka Cluster Details

To view your Kafka cluster details, use the following command:

```
heroku addons:info kafka -a kafka-app
```

You should receive a command line output similar to the content below:

```
=== kafka-round-82325

Attachments:  kafka-app::KAFKA

Installed at: Sun Mar 17 2024 03:23:13 GMT+0530 (India Standard
Time)

Max Price:    $100/month

Owning app:   kafka-app

Plan:         heroku-kafka:basic-0

Price:        ~$0.139/hour

State:        created
```

This output provides essential details about your Kafka cluster, including its name, installation date, maximum price, owning app, plan, price per hour, and current state.

Building your application

Now that your Kafka cluster is up and running, it's time to code your application. We'll use Python for this example, but the concepts translate to other languages.

Install the `kafka-python` library:

You can use either of the following commands to install the required library

```
pip install kafka
```

```
pip install kafka-python
```

1. Imports and Logging:

Imports:

- `logging`: Used for logging information and errors to a file or console.
- `ssl`: Used for secure communication with Kafka using SSL certificates.
- `KafkaProducer` and `KafkaConsumer`: Classes from the `kafka` library for interacting with Kafka as a producer and consumer, respectively.

```
import logging
import ssl
from kafka import KafkaProducer, KafkaConsumer
from kafka.errors import KafkaError
```

Logging Setup:

- Configures logging to print messages with timestamps, levels (INFO, ERROR), and message content.

```
# Set up logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s -
%(levelname)s - %(message)s')
```

2. Kafka Configuration:

- **KAFKA_URL:** Stores a comma-separated list of Kafka broker URLs, using the `kafka+ssl` scheme for secure connections.
- **KAFKA_PREFIX:** Specifies a prefix to be added to topic names.
- **Certificate Paths:**
    - `KAFKA_TRUSTED_CERT_PATH`: Path to the trusted CA certificate for SSL verification.

- `KAFKA_CLIENT_CERT_PATH`: Path to the client certificate for authentication.
- `KAFKA_CLIENT_CERT_KEY_PATH`: Path to the client certificate key.

```
# Kafka configuration
KAFKA_URL = "YOUR_KAFKA_URL"
KAFKA_PREFIX = "licking-62471."

# Certificate paths
KAFKA_TRUSTED_CERT_PATH = "trusted_ca_cert.pem"
KAFKA_CLIENT_CERT_PATH = "client_cert.pem"
KAFKA_CLIENT_CERT_KEY_PATH = "client_cert_key.pem"
```

You can find all of the configuration values by using `heroku config --app your_application_name`

For example: `heroku config --app kafka-app`

You can either hardcode the values or store the values in separate in the root directory itself.

3. Certificate Validation and SSL Context:

- **validate_cert_paths():** Ensures all required certificate paths are provided, raising an error otherwise.
- **create_ssl_context():** Creates an SSL context for secure communication:
  - Loads the trusted CA certificate.
  - Loads the client certificate and key.
  - Disables hostname verification (not recommended for production).

```
# Validate certificate paths

def validate_cert_paths():

    if not all([KAFKA_TRUSTED_CERT_PATH, KAFKA_CLIENT_CERT_PATH,
KAFKA_CLIENT_CERT_KEY_PATH]):

        raise ValueError("All certificate paths must be provided")



validate_cert_paths()



def create_ssl_context():

    ssl_context =
ssl.create_default_context(cafile=KAFKA_TRUSTED_CERT_PATH)

    ssl_context.load_cert_chain(certfile=KAFKA_CLIENT_CERT_PATH,
keyfile=KAFKA_CLIENT_CERT_KEY_PATH)

    ssl_context.check_hostname = False  # Disable hostname
verification

    return ssl_context
```

4. Kafka Producer Setup:

- **Extracts bootstrap servers:** Parses the `KAFKA_URL` string to extract individual broker hostnames and ports for connecting to Kafka.
- **Creates producer:**
    - Uses the extracted `bootstrap_servers` for establishing connections.
    - Specifies `security_protocol="SSL"` for secure communication.
    - Provides the created `ssl_context` for SSL configuration.
    - Sets a serializer to convert message values to UTF-8 encoded bytes for transmission.

```
# Create Kafka producer

ssl_context = create_ssl_context()
```

```
bootstrap_servers = []

for server_url in KAFKA_URL.split(","):

    host, port = server_url.split("://")[1].split(":")

    bootstrap_servers.append(f"{host}:{port}")



producer = KafkaProducer(

    bootstrap_servers=bootstrap_servers,

    security_protocol="SSL",  # Assuming SSL is required

    ssl_context=ssl_context,

    value_serializer=lambda v: v.encode('utf-8')

)
```

5. Producer Function:

- **produce_message(topic, message):**
    - Sends the provided message to the specified topic using the producer.
    - Logs a success message if successful.
    - Logs an error message if an exception occurs.

```
# Producer function

def produce_message(topic, message):

    try:

        producer.send(topic, message)

        logging.info(f"Produced message: {message}")

    except KafkaError as e:

        logging.error(f"Error producing message: {e}")
```

6. Kafka Consumer Setup:

- **Creates consumer:**
  - Uses the same `bootstrap_servers` as the producer.
  - Sets `auto_offset_reset='earliest'` to start consuming messages from the beginning of the topic.
  - Specifies `security_protocol="SSL"` and provides the `ssl_context` for secure communication.
  - Sets a deserializer to convert received message values from bytes back to UTF-8 strings.

```
consumer = KafkaConsumer(

    bootstrap_servers=bootstrap_servers,  # Split the URL by comma
(",")

    auto_offset_reset='earliest',

    security_protocol="SSL",

    ssl_context=ssl_context,

    value_deserializer=lambda m: m.decode('utf-8')

)
```

7. Consumer Function:

- **consume_messages(topic):**
  - Subscribes the consumer to the specified topic.
  - Continuously polls for messages:
    - Logs the value of each consumed message.
    - Logs an error message if an exception occurs.

```
# Consumer function

def consume_messages(topic):

    try:

        consumer.subscribe([topic])
```

```
        for message in consumer:
            logging.info(f"Consumed message: {message.value}")
    except KafkaError as e:
        logging.error(f"Error consuming messages: {e}")
```

8. Main Execution:

- If the script is run directly (not imported as a module):
    - Constructs a topic name using the `KAFKA_PREFIX`.
    - Calls `produce_message` to send a test message.
    - Calls `consume_messages` to start consuming messages from the topic

```
if __name__ == "__main__":
    topic = f"{KAFKA_PREFIX}my-topic"
    produce_message(topic, "Hello, Kafka!")
    consume_messages(topic)
```

Important Notes:

- Replace the hardcoded credentials with a secure authentication mechanism for production use.
- Heorku uses environment variables for storing information securely, you can use the `heroku config:set KEY=YOUR_VALUE -a app_name` command for setting up your credentials.
- Consider error handling and logging for more robust application behaviour.

Deploying Your Application to Heroku

With your Python scripts ready, leverage Heroku for deployment:

1. Initialise a Git repository in your project directory:

```
git init
```

2. Add your code and any dependencies (requirements.txt) to the Git repository.

   ```
   pip freeze > requirements.txt
   ```

3. Create a new Heroku app (if not already created):

   ```
   heroku create kafka-app (replace with your desired app name)
   ```

4. Configure a Procfile to specify how Heroku should run your applications:

   ```
   web: gunicorn app:app  # Replace 'consumer' with your module name if it differs
   ```

5. Deploy your code to Heroku:

   ```
   git add .

   git commit -m "First deployment"

   heroku git:remote -a kafka-app

   git push heroku master
   ```

Testing and deployment

After deploying your Kafka application, you can test it by running the producer script. If everything works well initially, the logs will show positive signs:

- The producer successfully connects to all three brokers in the Kafka cluster, establishing communication.
- It identifies the version of each broker, ensuring compatibility.
- Most importantly, the producer subscribes to a specific topic named "licking-62471.my-topic," indicating it can interact with the Kafka message flow as intended.
- These initial steps in the logs suggest a successful connection and setup for your Kafka producer.

```
2024-03-26 01:32:37,105 - INFO - <BrokerConnection node_id=bootstrap-1 host=ec2-100-25-107-37.compute-1.amazonaws.com:9096 <connecting> [IPv4 ('100.25.107.37', 9096
)]>: connecting to ec2-100-25-107-37.compute-1.amazonaws.com:9096 [('100.25.107.37', 9096) IPv4]
2024-03-26 01:32:37,105 - INFO - Probing node bootstrap-1 broker version
2024-03-26 01:32:38,316 - INFO - Set configuration api_version=(2, 5, 0) to skip auto check_version requests on startup
2024-03-26 01:32:38,418 - INFO - <BrokerConnection node_id=bootstrap-7 host=ec2-3-231-110-127.compute-1.amazonaws.com:9096 <connecting> [IPv4 ('3.231.110.127', 9096
)]>: connecting to ec2-3-231-110-127.compute-1.amazonaws.com:9096 [('3.231.110.127', 9096) IPv4]
2024-03-26 01:32:38,316 - INFO - Set configuration api_version=(2, 5, 0) to skip auto check_version requests on startup
2024-03-26 01:32:38,418 - INFO - <BrokerConnection node_id=bootstrap-7 host=ec2-3-231-110-127.compute-1.amazonaws.com:9096 <connecting> [IPv4 ('3.231.110.127', 9096
)]>: connecting to ec2-3-231-110-127.compute-1.amazonaws.com:9096 [('3.231.110.127', 9096) IPv4]
2024-03-26 01:32:38,419 - INFO - Probing node bootstrap-7 broker version
2024-03-26 01:32:39,217 - INFO - <BrokerConnection node_id=bootstrap-7 host=ec2-3-231-110-127.compute-1.amazonaws.com:9096 <handshake> [IPv4 ('3.231.110.127', 9096)
]>: Connection complete.
2024-03-26 01:32:39,599 - INFO - Broker version identified as 2.5.0
2024-03-26 01:32:39,600 - INFO - Set configuration api_version=(2, 5, 0) to skip auto check_version requests on startup
2024-03-26 01:32:39,600 - WARNING - group_id is None: disabling auto-commit.
2024-03-26 01:32:39,717 - INFO - <BrokerConnection node_id=7 host=ec2-3-208-232-40.compute-1.amazonaws.com:9096 <connecting> [IPv4 ('3.208.232.40', 9096)]>: connect
ing to ec2-3-208-232-40.compute-1.amazonaws.com:9096 [('3.208.232.40', 9096) IPv4]
2024-03-26 01:32:40,548 - INFO - <BrokerConnection node_id=7 host=ec2-3-208-232-40.compute-1.amazonaws.com:9096 <handshake> [IPv4 ('3.208.232.40', 9096)]>: Connecti
on complete.
2024-03-26 01:32:40,548 - INFO - <BrokerConnection node_id=bootstrap-1 host=ec2-100-25-107-37.compute-1.amazonaws.com:9096 <connected> [IPv4 ('100.25.107.37', 9096)
]>: Closing connection.
2024-03-26 01:33:39,620 - ERROR - Error producing message: KafkaTimeoutError: Failed to update metadata after 60.0 secs.
2024-03-26 01:33:39,620 - INFO - Updating subscribed topics to: ['licking-62471.my-topic']
```

Conclusion

This guide equips you with the knowledge to construct a basic event-driven application, leveraging the features of Apache Kafka available on Heroku. Here we've explored the core concepts of event-driven architectures, grasped the benefits of using Kafka along with Heroku, and implemented a sample application that produces and consumes sensor data.

Remember to prioritise security in production environments by replacing hardcoded credentials and incorporating robust error handling mechanisms. For any new information, feel free to refer to the official [Apache Kafka on Heroku documentation.]

As you venture further, you'll explore advanced features like consumer groups, message partitioning, and data serialisation formats to tailor your event-driven applications to your specific needs.