Get started

This page tells you how to get started with the Casino Platform Event Streaming and consume your first message.

On this page:

Prerequisites

To consume messages from the Event Streaming, ensure you have a Kafka client library SDK. Conduktor.io lists an extensive range of solutions for various programming languages.

Contact technical account manager

As soon as you conclude that Event Streaming is a valuable asset to your project, do the following:

1. Prepare a list of IPs of your servers that will connect to Kafka.
   The service is only accessible from allowlisted IP addresses. Requests from other IPs will be blocked.
2. Contact your account manager, attaching the list of IPs.
   DevOps engineers will prepare security certificates and consumer groups and forward these data to your account manager.

When everything is settled, your account manager will send you the following data:

- Certificates required to access topics and consumer groups:
  - Two `.crt` files
  - One `.key` file
- Name of your consumer group. Generally, it's your project name.
- A list of topics you can consume from.
- IPs of Kafka brokers.

After you receive all the data, you can authorize and test consumption.

Consume messages

Depending on your workflow, you can consume messages from a terminal or using a programming language. The example below shows how to connect to Kafka broker and consume messages using ruby-kafka.

Consume messages in Ruby

1. Create a Kafka connection instance with broker IPs, certificates, and client id.

```
# gem 'ruby-kafka'

connection = Kafka.new(
  ["BROKER_1_IP", "BROKER_2_IP", "BROKER_3_IP"],
  ssl_ca_cert: File.read('KAFKA_CA_PATH'),
  ssl_client_cert: File.read('KAFKA_CLIENT_PATH'),
  ssl_client_cert_key: File.read('KAFKA_CLIENT_KEY'),
  client_id: 'YOUR_CLIENT_ID'
)
```

   Where:

   - `{BROKER_1_IP}`, `{BROKER_2_ip}`, and `{BROKER_3_IP}` — IP addresses of the Kafka brokers that you received from your TAM.
   - `{KAFKA_CA}` — path to the `kafka-ca.crt` file.

     If your system can't process the `.crt` files, you can convert them to `.pem` by running the following command in the terminal:

     ```
     openssl x509 -in cert.crt -out cert.pem
     ```

     Ensure you refer to the converted files in the Kafka connection instance.
   - `{KAFKA_CLIENT}` — path to the `kafka-client.crt` file.
   - `{KAFKA_CLIENT_KEY}` — path to the `kafka-client.key` file.
   - `{YOUR_CLIENT_ID}` — unique identifier for your project. It's used for logging, debugging, and auditing purposes. The field is optional, yet we highly recommend including it.

2. Connect to the consumer.

```
# Consumers with the same group id will form a Consumer Group
together.
consumer = connection.consumer(
  group_id: "{CONSUMER_GROUP}",
  offset_commit_interval: 5 # offsets are commited every 5 seconds
)
```

Where:

- `{CONSUMER_GROUP}` — name of the consumer group you received from your TAM.

3. Subscribe to one of the topics.

```
# It's possible to subscribe to multiple topics by calling
`subscribe` repeatedly.
consumer.subscribe("{TOPIC_NAME}")
```

Where:

- `{TOPIC_NAME}` — topic name. For example, `es.{casino_name}.balance_transaction`.

4. Consume messages. The following code logs topic, partition, offset, key, and the message body.

```
# This will loop indefinitely, yielding each message in turn
consumer.each_message do |message|
  puts message.topic, message.partition
  puts message.offset, message.key, message.value
end
```

See the full source code of the example:

Full code example

```
# gem 'ruby-kafka'

connection = Kafka.new(
  ["kafka1://11.11.111.11:9093", "kafka://22.222.222.22:9093"],
  ssl_ca_cert: File.read('kafka-ca.crt'),
  ssl_client_cert: File.read('kafka-client.crt'),
  ssl_client_cert_key: File.read('kafka-client.key'),
  client_id: 'my_client_name'
)

# Consumers with the same group id will form a Consumer Group together.
consumer = connection.consumer(
  group_id: "issued_group_name", # casino_name
  offset_commit_interval: 5 # offsets are commited every 5 seconds
)

# It's possible to subscribe to multiple topics by calling `subscribe`
repeatedly.
consumer.subscribe("topic1")

# Stop the consumer when the SIGTERM signal is sent to the process.
# It's better to shut down gracefully than to kill the process.
trap("TERM") { consumer.stop }

# This will loop indefinitely, yielding each message in turn.
consumer.each_message do |message|
  puts message.topic, message.partition
  puts message.offset, message.key, message.value
end
```

Consume messages in Go

Here's example of message consumption using Go.

Go example

```
package main

import (
    "context"
    "os"
    "os/signal"
    "sync"
    "syscall"
    "time"

    "crypto/tls"
    "crypto/x509"
    "fmt"
    "github.com/Shopify/sarama"
    log "github.com/sirupsen/logrus"
    "io/ioutil"
)

func NewTLSConfigFromFile(clientCertFile, clientKeyFile, caCertFile
string) (*tls.Config, error) {
    // Load client cert
    cert, err := tls.LoadX509KeyPair(clientCertFile, clientKeyFile)
    if err != nil {
        return nil, fmt.Errorf("failed to load client cert: %w", err)
    }

    // Load CA cert
    caCert, err := ioutil.ReadFile(caCertFile)
    if err != nil {
        return nil, fmt.Errorf("failed to load CA cert: %w", err)
    }

    cfg := newTLSConfig(cert, caCert)

    return cfg, nil
}

func NewTLSConfigFromBytes(clientCert, clientKey, caCert []byte) (*tls.
Config, error) {
    cert, err := tls.X509KeyPair(clientCert, clientKey)
    if err != nil {
        return nil, fmt.Errorf("failed to build key-pair cert: %w", err)
    }

    cfg := newTLSConfig(cert, caCert)

    return cfg, nil
}

func newTLSConfig(tlsCert tls.Certificate, caCert []byte) *tls.Config {
    cfg := tls.Config{}

    cfg.Certificates = []tls.Certificate{tlsCert}
    cfg.InsecureSkipVerify = true

    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)
    cfg.RootCAs = caCertPool

    return &cfg
}

type KafkaConfig struct {
    ClientID           string        `env:"KAFKA_CLIENT_ID"`
    Brokers            []string      `env:"KAFKA_BROKERS"`
```

```
    GroupID            string        `env:"KAFKA_GROUP_ID"`
    AutoOffsetReset    string        `env:"KAFKA_AUTO_OFFSET_RESET"`
    AutoCommitInterval time.Duration `env:"KAFKA_AUTO_COMMIT_INTERVAL"`
    SSLEnabled         bool          `env:"KAFKA_SSL_ENABLED"`
    SSLClientCert      string        `env:"KAFKA_SSL_CLIENT_CERT"`
    SSLClientKey       string        `env:"KAFKA_SSL_CLIENT_KEY"`
    SSLCACert          string        `env:"KAFKA_SSL_CA_CERT"`
    Topics             []string
    Ready              chan bool
}

func NewKafka() *KafkaConfig {
    return &KafkaConfig{
        ClientID: "my_client_name",
        Brokers: []string{
            "localhost:9092",
        },
        GroupID:            "issued_group_name",
        AutoOffsetReset:    "latest",
        AutoCommitInterval: 10 * time.Second,
        Topics:             []string{"topic1", "topic2"},
        Ready:              make(chan bool),
    }
}

func (k *KafkaConfig) Connect() func() {
    kafkaCfg := sarama.NewConfig()
    kafkaCfg.ClientID = k.ClientID
    switch k.AutoOffsetReset {
    case "largest", "latest", "newest":
        kafkaCfg.Consumer.Offsets.Initial = sarama.OffsetNewest
    case "smallest", "earliest", "oldest":
        kafkaCfg.Consumer.Offsets.Initial = sarama.OffsetOldest
    }
    kafkaCfg.Consumer.Offsets.AutoCommit.Enable = true
    kafkaCfg.Consumer.Offsets.AutoCommit.Interval = k.AutoCommitInterval
    if k.SSLEnabled {
        tlsCfg, err := NewTLSConfigFromFile(
            k.SSLClientCert,
            k.SSLClientKey,
            k.SSLCACert,
        )
        if err != nil {
            log.Errorf("failed to init TLS config: %v", err)
        }

        kafkaCfg.Net.TLS.Enable = true
        kafkaCfg.Net.TLS.Config = tlsCfg
    }

    client, err := sarama.NewClient(
        k.Brokers,
        kafkaCfg,
    )
    if err != nil {
        log.Errorf("failed to init kafka client: %v", err)
    }

    consumerGroup, err := sarama.NewConsumerGroupFromClient(k.GroupID,
client)
    if err != nil {
        log.Errorf("failed to init kafka consumer group: %v", err)
    }

    ctx, cancel := context.WithCancel(context.Background())
    wg := &sync.WaitGroup{}
    wg.Add(1)
    go func() {
        defer wg.Done()
        for {
            if err := consumerGroup.Consume(ctx, k.Topics, k); err != nil {
```

```
                // When setup fails, error will be returned here
                log.Errorf("Error from consumer: %v", err)
                return
            }
            // check if context was cancelled, signaling that the consumer
should stop
            if ctx.Err() != nil {
                log.Println(ctx.Err())
                return
            }
            k.Ready = make(chan bool)
        }
    }()
    <-k.Ready
    log.Infoln("Sarama consumer up and running!...")
    // Ensure that the messages in the channel are consumed when the
system exits
    return func() {
        log.Info("kafka close")
        cancel()
        wg.Wait()
        if err = consumerGroup.Close(); err != nil {
            log.Errorf("Error closing client: %v", err)
        }
    }
}

// Setup is run at the beginning of a new session, before ConsumeClaim
func (k *KafkaConfig) Setup(session sarama.ConsumerGroupSession) error {
    log.Info("setup")
    // session.ResetOffset("topic1", 0, 13, "")
    log.Info(session.Claims())
    // Mark the consumer as Ready
    close(k.Ready)
    return nil
}

// Cleanup is run at the end of a session, once all ConsumeClaim
goroutines have exited
func (k *KafkaConfig) Cleanup(sarama.ConsumerGroupSession) error {
    log.Info("cleanup")
    return nil
}

// ConsumeClaim must start a consumer loop of ConsumerGroupClaim's
Messages().
func (k *KafkaConfig) ConsumeClaim(session sarama.ConsumerGroupSession,
claim sarama.ConsumerGroupClaim) error {

    // NOTE:
    // Do not move the code below to a goroutine.
    // The `ConsumeClaim` itself is called within a goroutine, see:
    // <https://github.com/Shopify/sarama/blob/master/consumer_group.
go#L27-L29>
    // Specific consumption news
    for message := range claim.Messages() {
        log.Infof("[topic:%s] [partiton:%d] [offset:%d] [value:%s] [time:%
v]",
            message.Topic, message.Partition, message.Offset, string
(message.Value), message.Timestamp)
        // Update displacement
        session.MarkMessage(message, "")
    }
    return nil
}

func main() {
    k := NewKafka()
    c := k.Connect()

    sigterm := make(chan os.Signal, 1)
```

```
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Warnln("terminating: via signal")
    }
    c()
}
```

Offset management

Consumer offset lets you resume processing messages after the consumption stops. For example, the event consumption can crash due to a network abruption. While the consumer is unresponsive, Event Streaming can still produce events. If other members of the consumer groups don't know the last offset committed before the crash, they start processing messages from the newest message. That's where a data gap can occur.
To avoid data loss, you must manually commit offsets in code or configure auto-commits based on particular periods. In this case, every consumer will checkpoint its position, so other consumer group members can resume from the last commit when another member crashes.

Via the `auto.offset.reset` setting (depending on the solution you use, the setting name varies but the main idea remains) of the Kafka consumer, you can define whether to consume from the beginning of a topic partition or to consume new messages only if there's no initial offset for the consumer group.

In the Go example above, the following block is responsible for this:

```
switch k.AutoOffsetReset {
    case "largest", "latest", "newest":
        kafkaCfg.Consumer.Offsets.Initial = sarama.OffsetNewest
    case "smallest", "earliest", "oldest":
        kafkaCfg.Consumer.Offsets.Initial = sarama.OffsetOldest
}
```

We recommend setting the `Offset0ldest` value to consume from the beginning of the topic. This way you'll read all messages and not only those from the connection moment. It's especially crucial for the Exp ort topic.

Important

It's crucial to commit offsets in the production environment. As for the test environments, it's up to you to commit offsets there or not.

Enable auto-committing

As a rule, third-party services automatically commit offsets. Depending on your tool, enabling auto-commits varies. Check an example of enabling auto-commits in Go:

```
kafkaCfg.Consumer.Offsets.AutoCommit.Enable = true
kafkaCfg.Consumer.Offsets.AutoCommit.Interval = k.AutoCommitInterval
```

If you use another solution, refer to its documentation for more details.

Please rate this page: