

# Kafka for .NET Developers

Ian Cooper

x/Hachyderm: ICooper

# Who are you?

- Software Developer for more than 25 years
  - Stuff I care about: Messaging, EDA, Microservices, TDD, XP, OO, RDD & DDD, Code that Fits in My Head, C#
  - Places I have worked: DTI, Reuters, Sungard, Beazley, Huddle, Just Eat Takeaway
- No smart folks
  - Just the folks in this room



## Welcome to Brighter

This project is a Command Processor & Dispatcher implementation with support for task queues that can be used as a lightweight library.

It can be used for implementing [Ports and Adapters](#) and [CQRS \(PDF\)](#) architectural styles in .NET.

It can also be used in microservices architectures for decoupled communication between the services

[GET STARTED](#)

## Links

The Sample Code: <https://github.com/iancooper/KafkaForDotNetDevs>

Brighter: <https://github.com/BrighterCommand/Brighter>

This Presentation: <https://github.com/iancooper/Presentations>

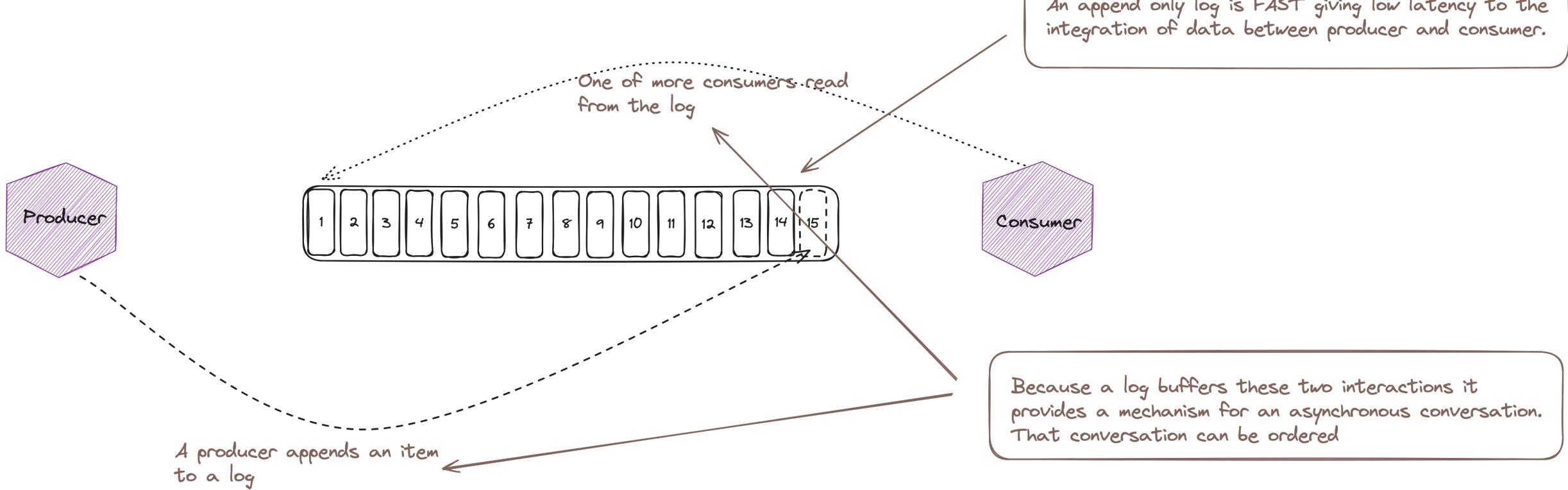
# Agenda

1. Kafka
2. Managing Kafka
3. Kafka and C#
4. Under the Hood
5. Reliable Producer
6. Reliable Consumer
7. Serializers, De-serializers & Schema Registry
6. Q&A

# Kafka

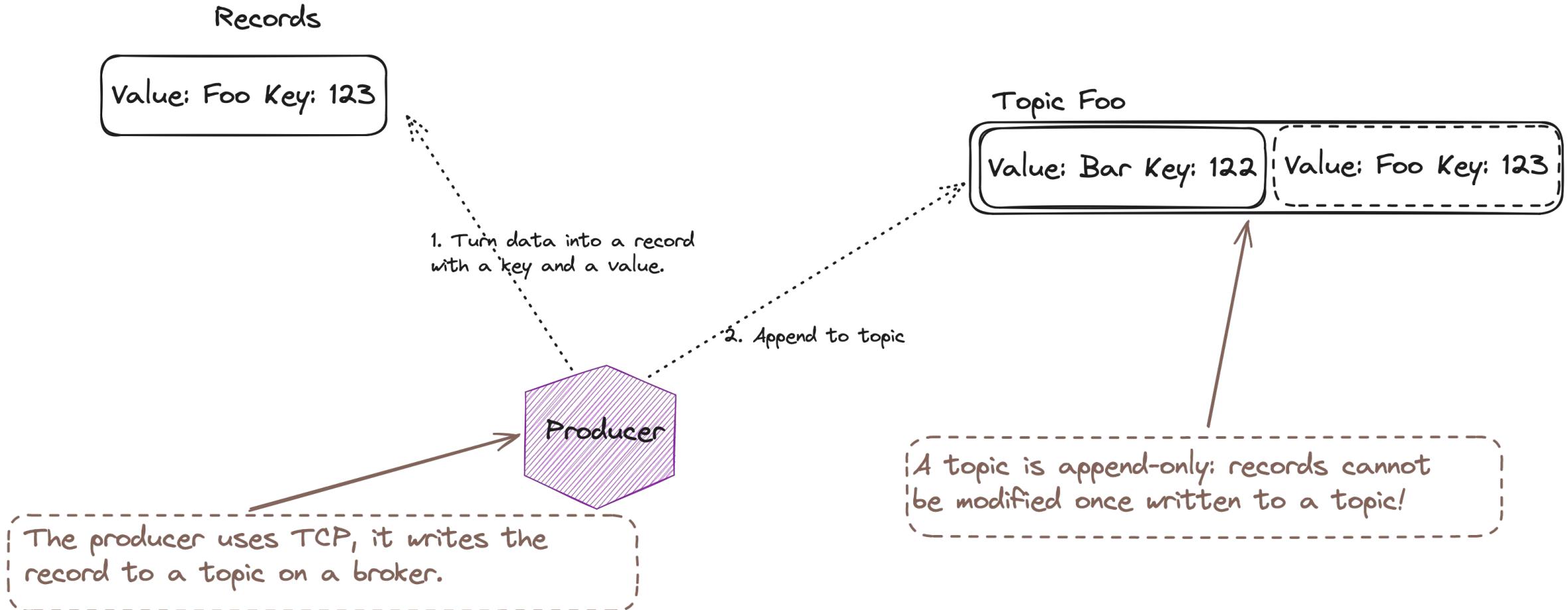
Kafka was created at LinkedIn by Jay Kreps, Neha Narkhede, Jun Rao in 2011.  
it became an Apache OSS project in October 2012

Original use case was data engineering, capturing data into LinkedIn's Data Warehouse from diverse sources so that it could be used for later analysis.

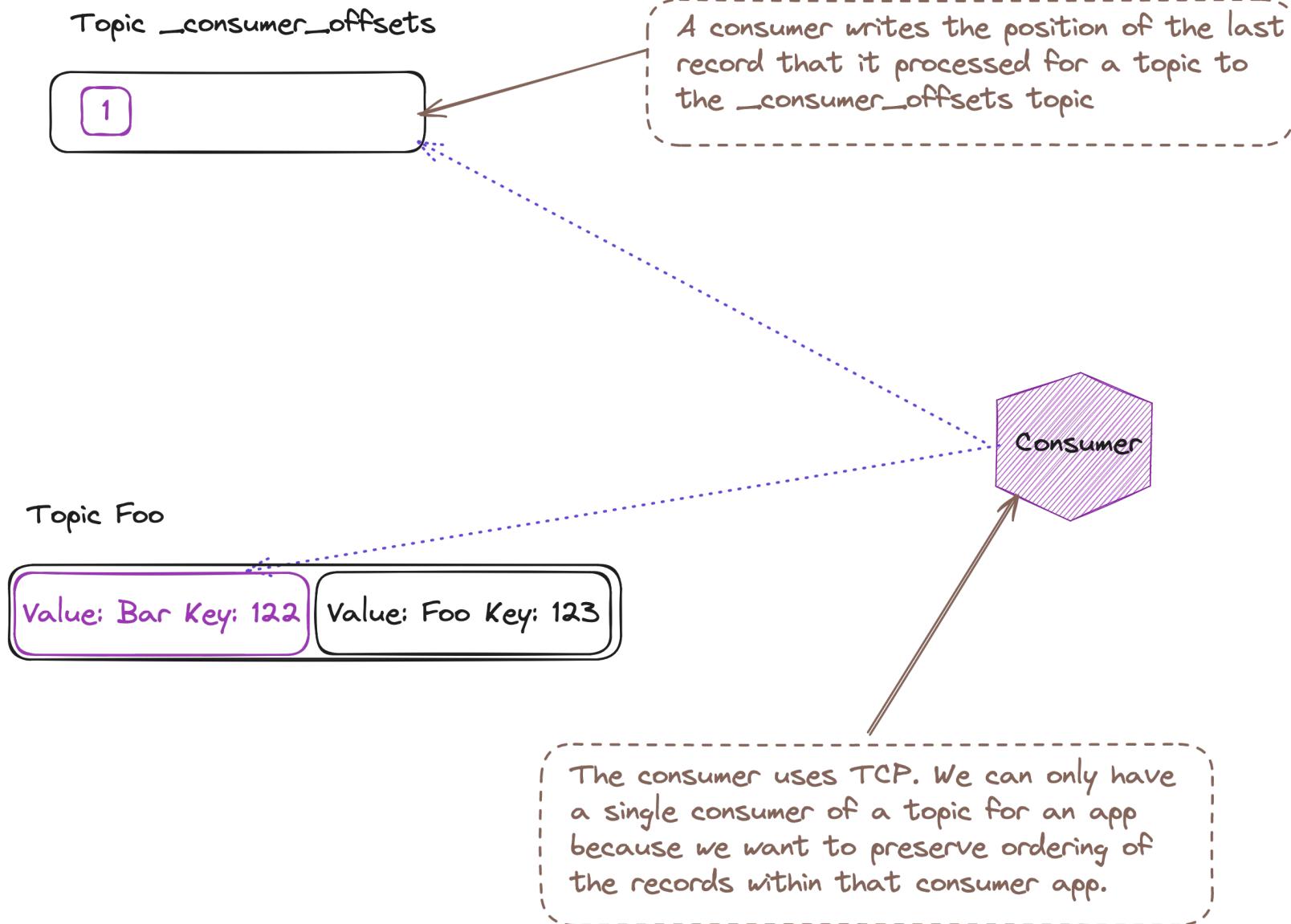


<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>

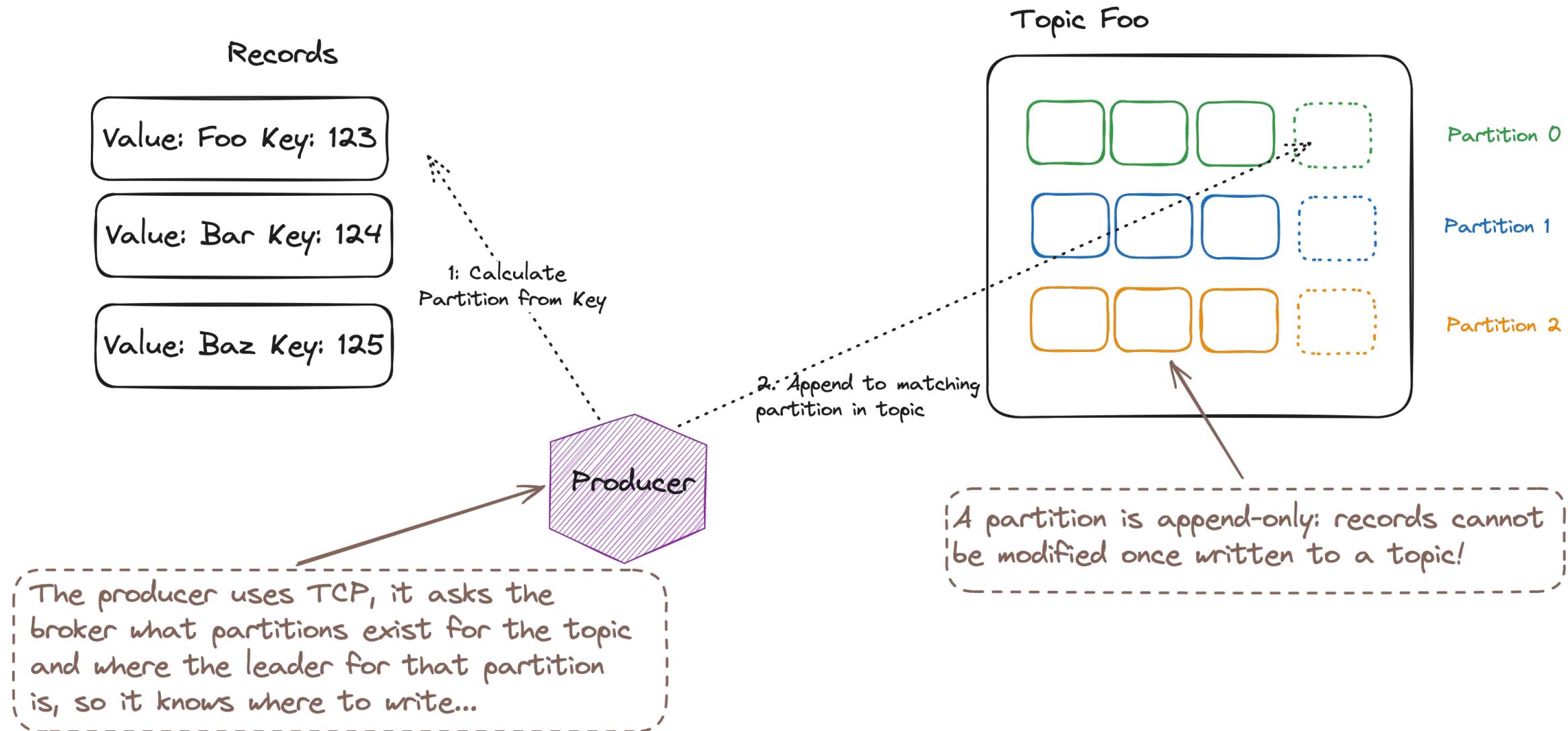
# Kafka Topics



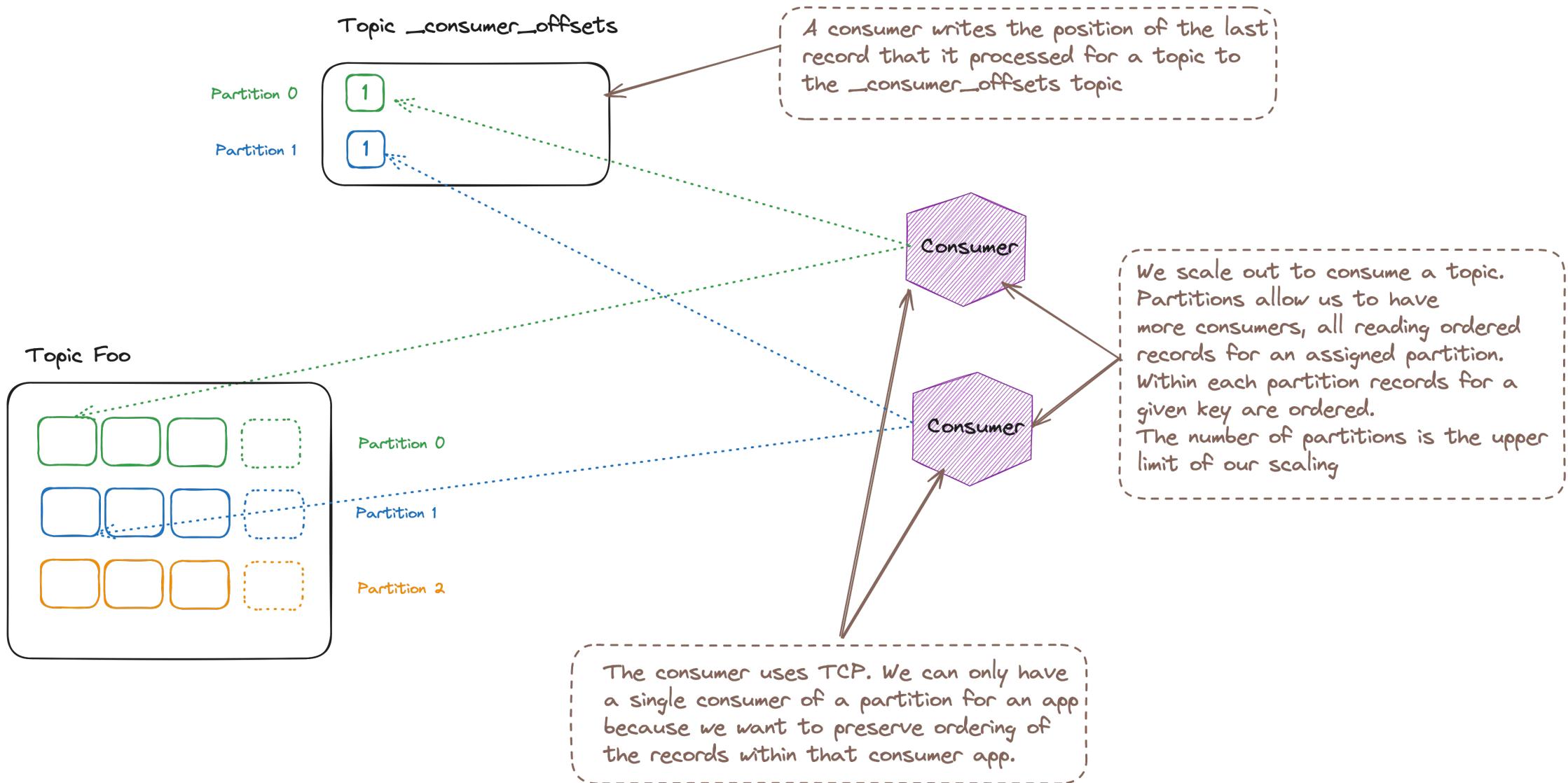
## Kafka Topics



# Kafka Topics

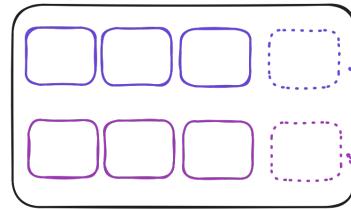


## Kafka Topics



## Kafka Topics

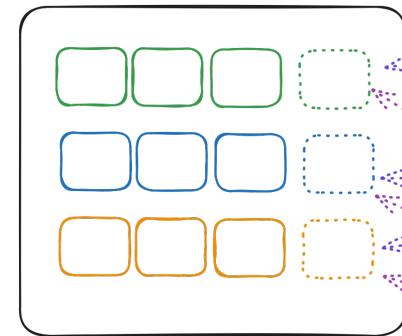
Topic \_consumer\_offsets



Partition 0

Partition 1

Topic Foo



Partition 0

Partition 1

Partition 2

Group: Alpha

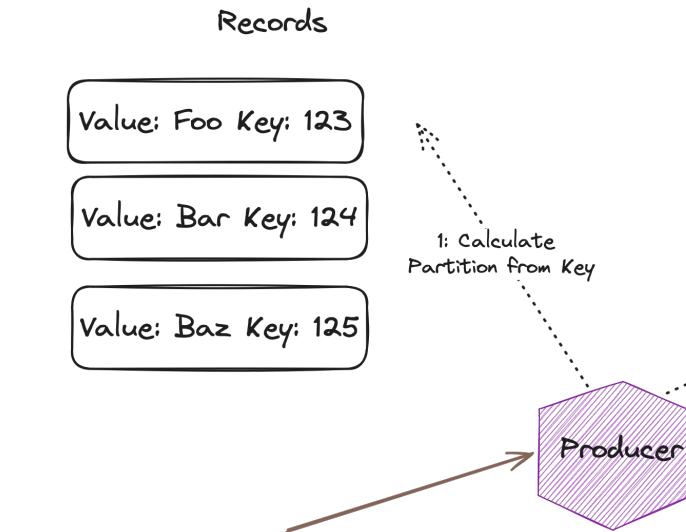


Group: Beta



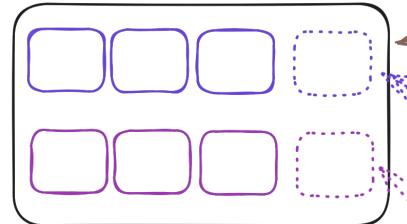
The consumer belongs to a group. Within a group (representing an application) only one consumer can read from a partition on a topic.

## Kafka Topics



The producer uses TCP, it asks the broker what partitions exist for the topic and where the leader for that partition is, so it knows where to write...

Topic \_\_consumer\_offsets

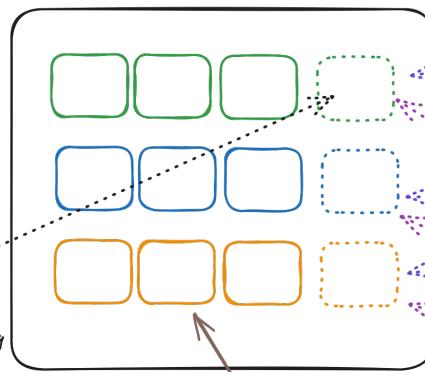


Partition 0

Partition 1

The \_\_consumer\_offsets topic has a partition per consumer group; a consumer writes the position of the last record that it wrote for a partition to the partition on the \_\_consumer\_offsets topic for its group

Topic Foo



Partition 0

Partition 1

Partition 2

A partition is append-only: records cannot be modified once written to a topic!

Group: Alpha



Group: Beta



The consumer uses TCP. The partitions for a topic are allocated to a single reader in each group.

# Managing Kafka

# CLI

## Kafka CLI

Suite of tools located in the /bin directory where you installed Kafka:

`kafka-topics.sh` => create or delete or topic

`kafka-console-producer.sh` => produce records to a topic

`kafka-console-consumer.sh` => consume records from a topic

`kafka-get-offsets.sh` => retrieve topic/partition offsets

...many more

<https://docs.confluent.io/kafka/operations-tools/kafka-tools.html>

## KCat

A generic, non-JVM, producer and consumer for Kafka.

Producer Mode:

Read messages from stdin (or a file)

`kcat -b localhost:9092 -t my_topic -P test`

Consumer Mode:

Read messages from a topic and writes to stdout

`kcat -b localhost:9092 -t my-topic`

<https://github.com/edenhill/kcat>

# UIs

The screenshot shows the UI for Apache Kafka version 56fa824 v0.7.1. The left sidebar has a 'local' dropdown set to 'Brokers'. The main area is titled 'Brokers' and displays 'Uptime' and 'Partitions' metrics. Under 'Uptime', it shows 1 Broker Count, 1 Active Controller, and Version 3.2-IV0. Under 'Partitions', it shows 93 of 93 Online, 0 URP, 93 of 93 In Sync Replicas, and 0 Out Of Sync Replicas. Below these are detailed tables for Broker ID 1, showing Disk usage (2 KB, 93 segment(s)), Partitions skew (-), Leaders (93), Leader skew (-), Online partitions (93), Port (29092), and Host (kafka0).

Broker ID	Disk usage	Partitions skew	Leaders	Leader skew	Online partitions	Port	Host
1	2 KB, 93 segment(s)	-	93	-	93	29092	kafka0

UI for Apache Kafka: <https://github.com/provectus/kafka-ui/tree/master>

Other OSS UIs:

AKHQ  
Kafdrop  
Lenses  
CMACK

Paid UIs:

Confluent CC  
Conduktor

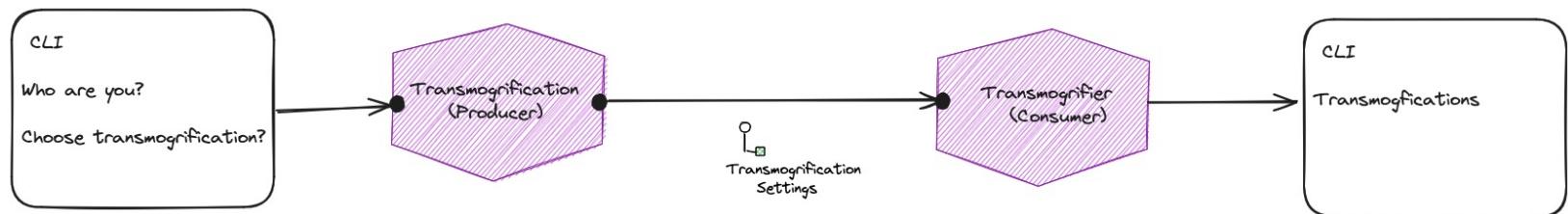
# Kafka and C#

# Building a Transmogrifier

YOU STEP INTO THIS CHAMBER,  
SET THE APPROPRIATE DIALS,  
AND IT TURNS YOU INTO  
WHATEVER YOU'D LIKE TO BE.



Calvin and Hobbes – Bill Watterson



# Producing a Message in C#

The topic we want to produce to.

```
1 using Confluent.Kafka;
2
3 namespace Transmogrification;
4
5 public class Dispatcher(string topic, Dictionary<string, string> producerConfig) : IDisposable
6 {
7     private readonly IProducer<string, string> _producer = new ProducerBuilder<string, string>(producerConfig).Build();
8
9     public void Transmogrify(TransmogrificationSettings settings)
10    {
11        _producer.Produce(
12            topic,
13            new Message<string, string>
14            {
15                Key = settings.Name,
16                Value = settings.Transformation
17            }
18        );
19        _producer.Flush(TimeSpan.FromSeconds(10));
20    }
21
22    public void Dispose()
23    {
24        _producer.Dispose();
25    }
26 }
```

Produce a message to a topic; build message from our data

```
1 var producerConfig = new Dictionary<string, string>
2 {
3     { "bootstrap.servers", "localhost:9092" }
4 };
```

Create a producer from the config; note that ProducerBuilder<string, string> means:  
- key: string  
- value: string

# Consuming a Message in C#

The topic we want to  
consume from.

```
1 using Confluent.Kafka;
2 using Spectre.Consumer;
3
4 namespace Transmogrifier;
5
6 public class MessagePump(string topic, Dictionary<string, string> consumerConfig)
7 {
8     private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig).Build();
9
10    public async Task Run<TDataType>(
11        Func<Message<string, string>, TDataType> translator,
12        Action<TDataType> handler,
13        CancellationToken cancellationToken = default)
14    {
15        ...
16    }
17 }
```

Generic, so pass in:  
- translator  
- handler

```
1 var consumerConfig = new Dictionary<string, string>()
2 {
3     { "bootstrap.servers", "localhost:9092" },
4     { "group.id", "transmogrification-consumer" },
5 };
```

Create a consumer from the config; note  
that IConsumer<string, string> means:  
- key: string  
- value: string

# Consuming a Message in C#



Subscribe to topic

```
1     try
2     {
3         _consumer.Subscribe(topic);
4
5         while (true)
6         {
7             var consumeResult = _consumer.Consume(cancellationToken);
8
9             if (consumeResult.IsPartitionEOF)
10            {
11                await Task.Delay(1000, cancellationToken);
12                continue;
13            }
14
15            var dataType = translator(consumeResult.Message);
16
17            handler(dataType);
18        }
19    }
20    // error handling
21    finally
22    {
23        _consumer.Close();
24    }
```

Message Pump is an endless loop

Read next result; SDK is buffering for us. Yield if nothing in buffer. Break if cancelled.

Translate the message

Handle the message

Tidy

# Consuming a Message in C#

The topic we want to consume from.

Generic, so pass in:  
- translator  
- handler

Subscribe to topic

Message Pump is an endless loop

Read next result; SDK is buffering for us. Yield if nothing in buffer. Break if cancelled.

```
var consumerConfig = new Dictionary<string, string>()
{
    { "bootstrap.servers", "localhost:9092" },
    { "group.id", "transmogrification-consumer" },
};
```

```
1 using Confluent.Kafka;
2 using Spectre.Console;
3
4 namespace Kafka.Consumer;
5
6 public class MessagePump(string topic, Dictionary<string, string> consumerConfig)
7 {
8     private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig).Build();
9
10    public async Task Run<TDataType>(
11        Func<Message<string, string>, TDataType> translator,
12        Action<TDataType> handler,
13        CancellationToken cancellationToken = default)
14    {
15        try
16        {
17            _consumer.Subscribe(topic);
18
19            while (true)
20            {
21                var consumeResult = _consumer.Consume(cancellationToken);
22
23                if (consumeResult?.IsPartitionEOF)
24                {
25                    await Task.Delay(1000, cancellationToken);
26                    continue;
27                }
28
29                var dataType = translator(consumeResult.Message);
30
31                handler(dataType);
32            }
33        catch(ConsumeException e)
34        {
35            AnsiConsole.WriteLine(e);
36        }
37        catch (OperationCanceledException)
38        {
39            //Pump was cancelled, do nothing
40        }
41        finally
42        {
43            _consumer.Close();
44        }
45    }
46}
```

Create a consumer from the config; note that `IConsumer<string, string>` means:  
- key: string  
- value: string

Translate the message

Handle the message

Tidy

# Using the C# Consumer



```
1 using Spectre.Console;
2 using Transmogrifier;
3
4 const string topic = "transmogrification";
5
6 var consumerConfig = new Dictionary<string, string>()
7 {
8     { "bootstrap.servers", "localhost:9092" },
9     { "group.id", "transmogrification-consumer" },
10    { "auto.offset.reset", "earliest" }
11 };
12
13 CancellationTokenSource cts = new CancellationTokenSource();
14 Console.CancelKeyPress += (_, e) => {
15     e.Cancel = true; // prevent the process from terminating.
16     cts.Cancel();
17 };
```

The topic we want to consume from.

Configure:  
- where is Kafka?  
- what is our group?

# Using the C# Consumer



```
1 var box = new Box();
2 if (box.StartTransformation())
3 {
4     box.BeginTransforming();
5
6     var messagePump = new MessagePump(topic, consumerConfig);
7     await messagePump.Run(
8         (message) => new Transmogrification(message.Key, message.Value),
9         (transmogrification) =>
10    {
11        //Output the results using a spectre console table
12        var table = new Table();
13        table.AddColumn("Name");
14        table.AddColumn("Transformation");
15        table.AddColumn("Result");
16
17        table.AddRow(transmogrification.From, transmogrification.To, "Success");
18
19        AnsiConsole.Write(table);
20    },
21    cts.Token);
22
23    box.EndTransforming();
24 }
```

Translate the message to a data type



Handle the message



# Using the C# Consumer

The topic we want to consume from.

Configure:  
- where is Kafka?  
- what is our group?

```
1 using Spectre.Console;
2 using Transmogrifier;
3
4 const string topic = "transmogrification";
5
6 var consumerConfig = new Dictionary<string, string>()
7 {
8     { "bootstrap.servers", "localhost:9092" },
9     { "group.id", "transmogrification-consumer" },
10 };
11
12 CancellationTokenSource cts = new CancellationTokenSource();
13 Console.CancelKeyPress += (_, e) => {
14     e.Cancel = true; // prevent the process from terminating.
15     cts.Cancel();
16 };
17
18 var box = new Box();
19 if (box.StartTransformation())
20 {
21     box.BeginTransforming();
22
23     var messagePump = new MessagePump(topic, consumerConfig);
24     await messagePump.Run(
25         (message) => new Transmogrification(message.Key, message.Value),
26         (transmogrification) =>
27     {
28         //Output the results using a spectre console table
29         var table = new Table();
30         table.AddColumn("Name");
31         table.AddColumn("Transformation");
32         table.AddColumn("Result");
33
34         table.AddRow(transmogrification.From, transmogrification.To, "Success");
35
36         AnsiConsole.Write(table);
37
38     },
39     cts.Token);
40
41     box.EndTransforming();
42 }
```

Translate the message to a data type

Handle the data type

# Demo Two: Produce and Consume A Message

YOU STEP INTO THIS CHAMBER,  
SET THE APPROPRIATE DIALS,  
AND IT TURNS YOU INTO  
WHATEVER YOU'D LIKE TO BE.



Calvin and Hobbes – Bill Watterson

```
1 using Confluent.Kafka;
2
3 namespace Transmogrification;
4
5 public class Dispatcher(string topic, Dictionary<string, string> producerConfig) : IDisposable
6 {
7     private readonly IProducer<string, string> _producer = new ProducerBuilder<string, string>(producerConfig).Build();
8
9     public void Transmogrify(TransmogrificationSettings settings)
10    {
11        _producer.Produce(
12            topic,
13            new Message<string, string>
14            {
15                Key = settings.Name,
16                Value = settings.Transformation
17            }
18        );
19        _producer.Flush(TimeSpan.FromSeconds(10));
20    }
21
22    public void Dispose()
23    {
24        _producer.Dispose();
25    }
26 }
```

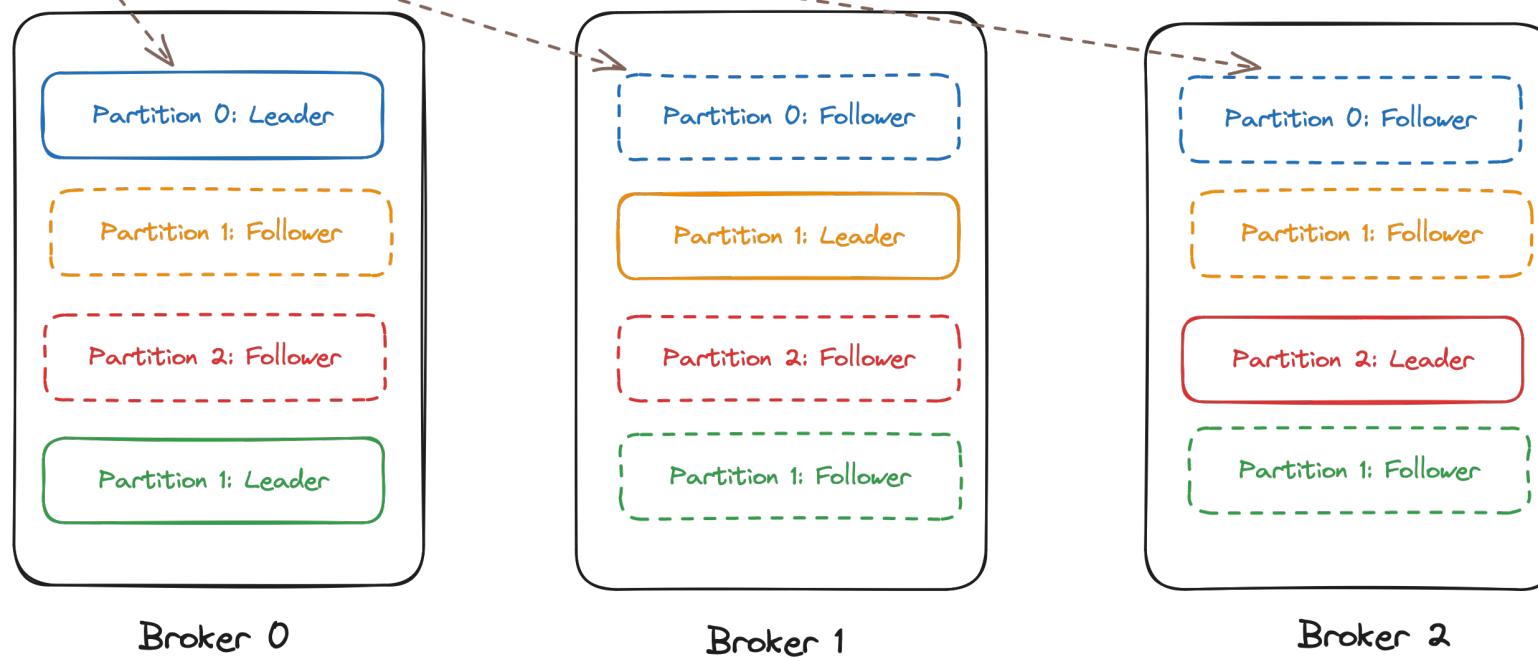
```
1 using Confluent.Kafka;
2 using Spectre.Console;
3
4 namespace Transmogrifier;
5
6 public class MessagePump(string topic, Dictionary<string, string> consumerConfig)
7 {
8     private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig).Build();
9
10    public async Task Run<TDataType>(
11        Func<Message<string, string>, TDataType> translator,
12        Action<TDataType> handler,
13        CancellationToken cancellationToken = default)
14    {
15        try
16        {
17            _consumer.Subscribe(topic);
18
19            while (true)
20            {
21                var consumeResult = _consumer.Consume(cancellationToken);
22
23                if (consumeResult.IsPartitionEOF)
24                {
25                    await Task.Delay(1000, cancellationToken);
26                    continue;
27                }
28
29                var dataType = translator(consumeResult.Message);
30
31                handler(dataType);
32            }
33        }
34        catch(ConsumeException e)
35        {
36            AnsiConsole.WriteLine(e);
37        }
38        catch (OperationCanceledException)
39        {
39            //Pump was cancelled, do nothing
40        }
41    }
42
43    finally
44    {
45        _consumer.Close();
46    }
47 }
```

# Under the Hood

## Kafka Clusters and Partitions

For availability a partition has a leader, and followers on other nodes in the cluster.

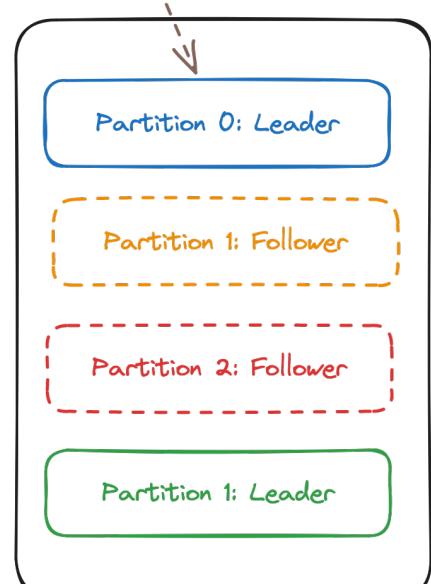
We only ever write to the leader. Before 2.3 read as well; after we can read from the closest replica instead.



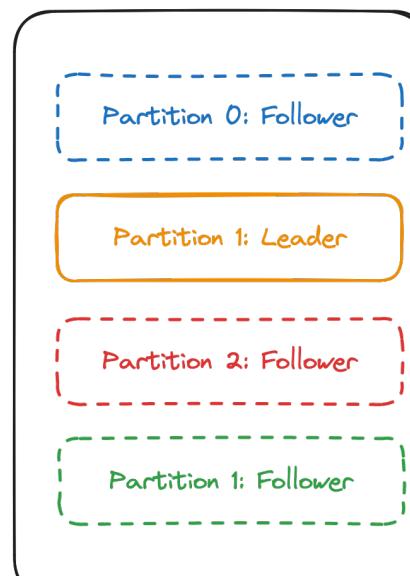
## Bootstrap Server

The bootstrap server is just one of the servers that we can query in order to obtain metadata about the location of partitions in the cluster.

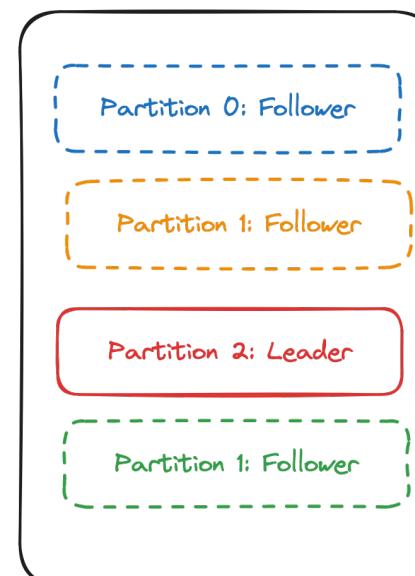
Given that I only write to the leader, how do I know where the leader is?



Broker 0



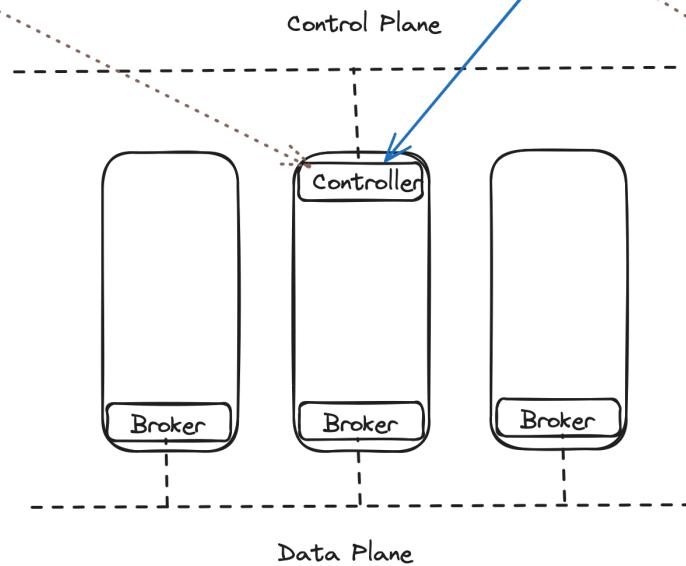
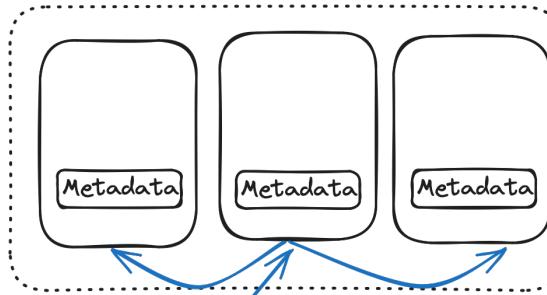
Broker 1



Broker 2

## Zookeeper Ensemble

The controller is a Kafka broker that holds the responsibility for electing the leader for a partition across the brokers; in the event that a broker is lost, it will replace any leaders on that broker by choosing new leaders from amongst the remaining brokers.

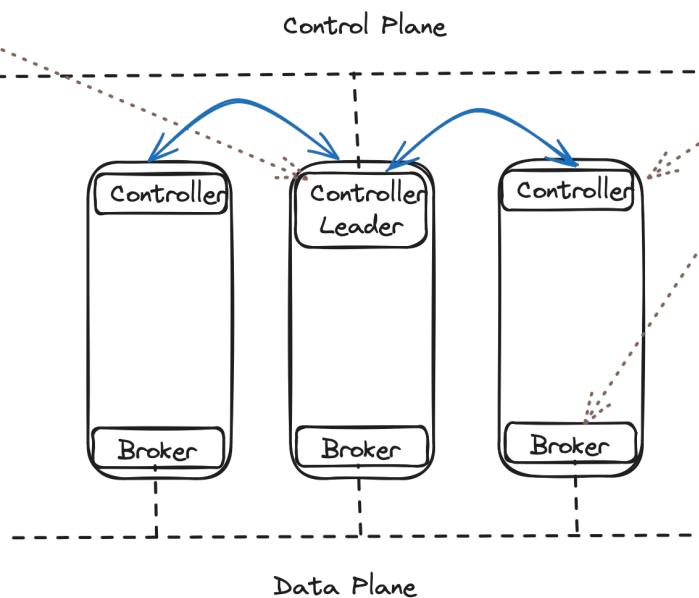


Kafka Cluster

Kafka brokers try to create a / ephemeral znode. Only one broker will succeed, it becomes the controller.

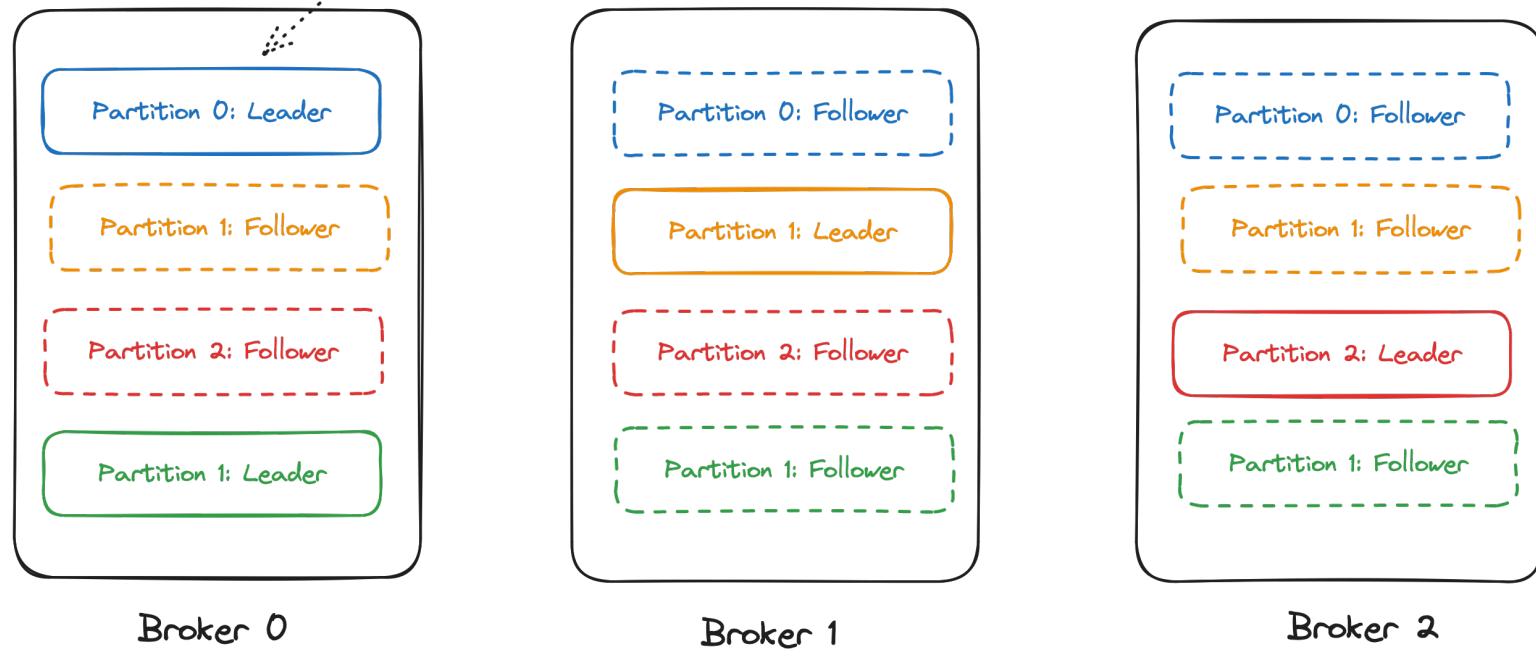
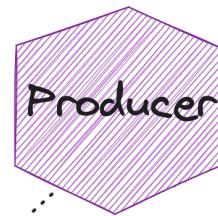
As the znode is ephemeral, if the controller crashes/shuts down it will disappear and another broker will create the /node and become the controller.

With Kafka Raft (KRaft) we no longer need Zookeeper, instead the brokers implement KRaft and are able to elect a leader from amongst themselves.



Not all nodes are controllers; we need enough in our quorum to break ties. We can configure so that nodes can be both controllers or brokers, or dedicated controllers and brokers.

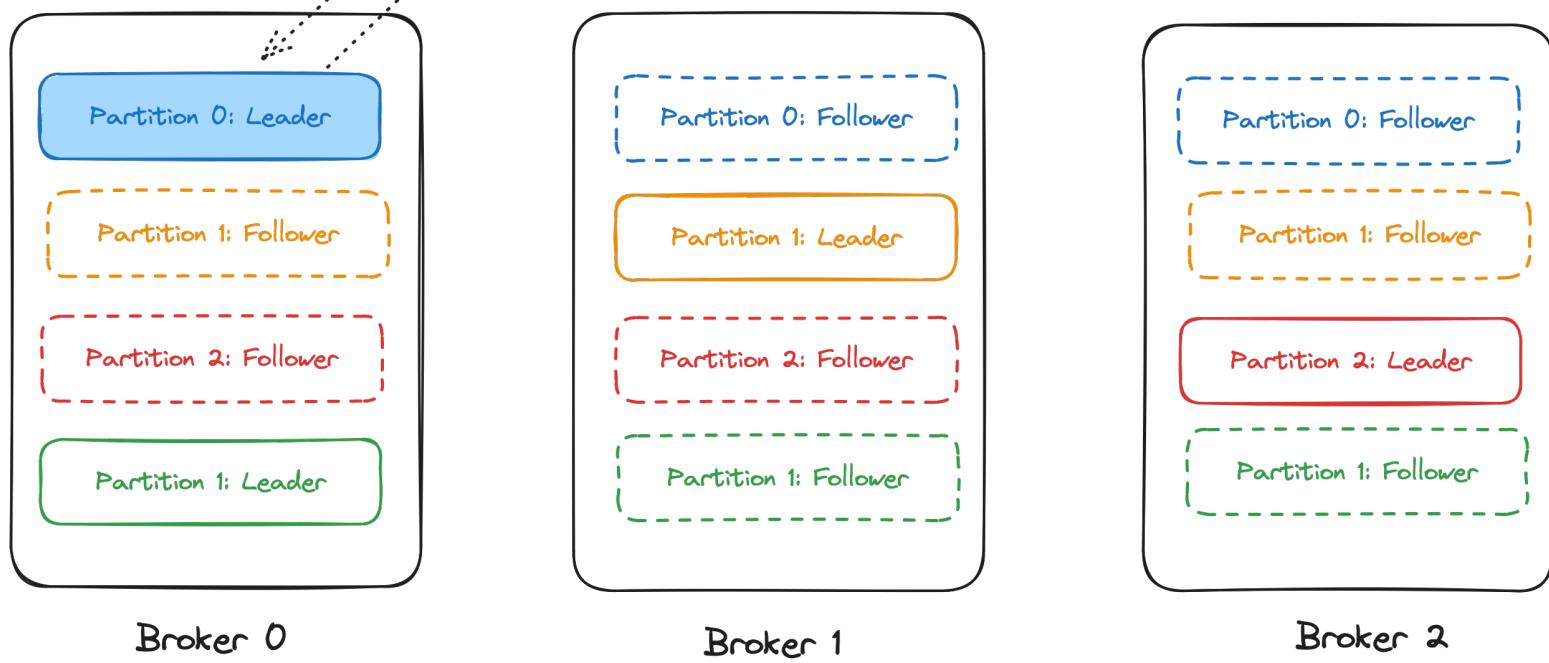
Acks = 0



`Acks = 0` is fire-and-forget. Don't wait for an acknowledgement from the server that the write has happened.

Risk => write does not happen, so producer is out of sync with consumers.

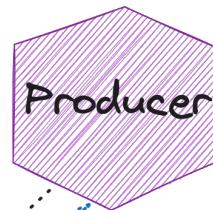
Acks = 1



Acks = 1 is leader only. Wait for an acknowledgement from the server that the write has happened; don't wait for it to be propagated to replicas.

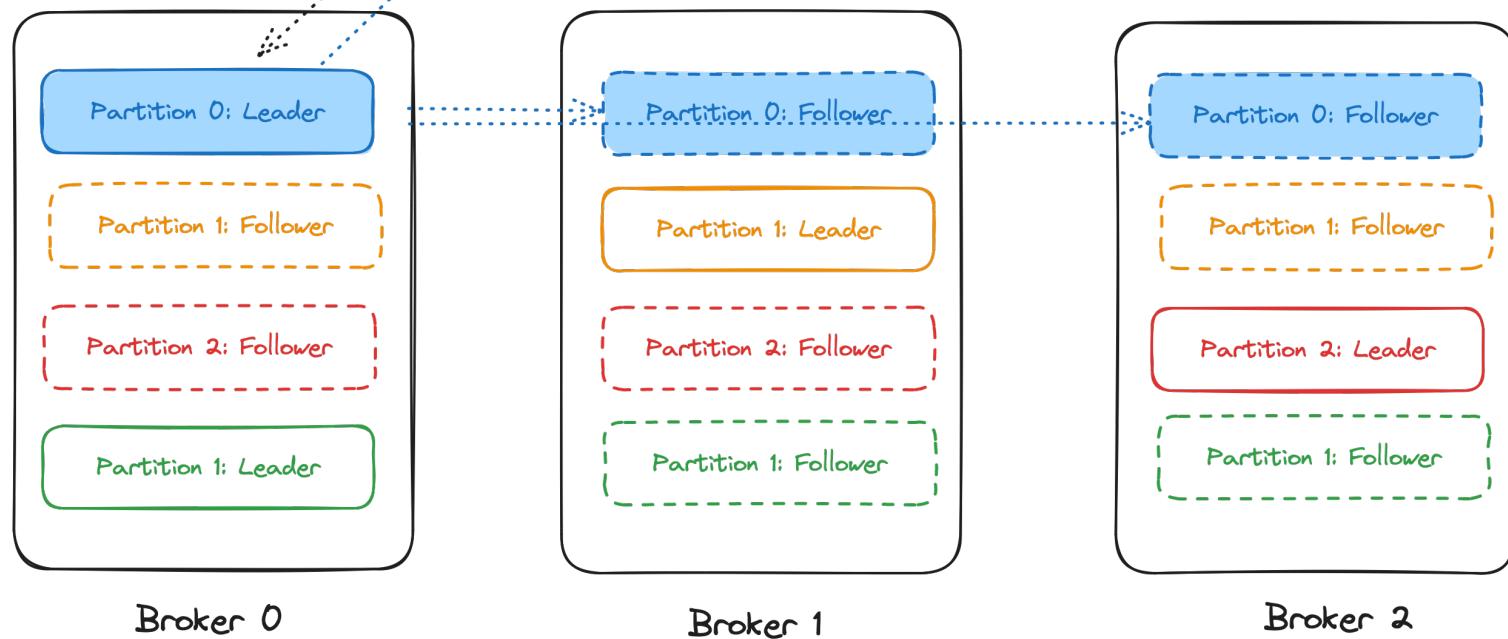
Risk => write to other replicas does not happen; in the event of failure we will lose messages.

Acks = All

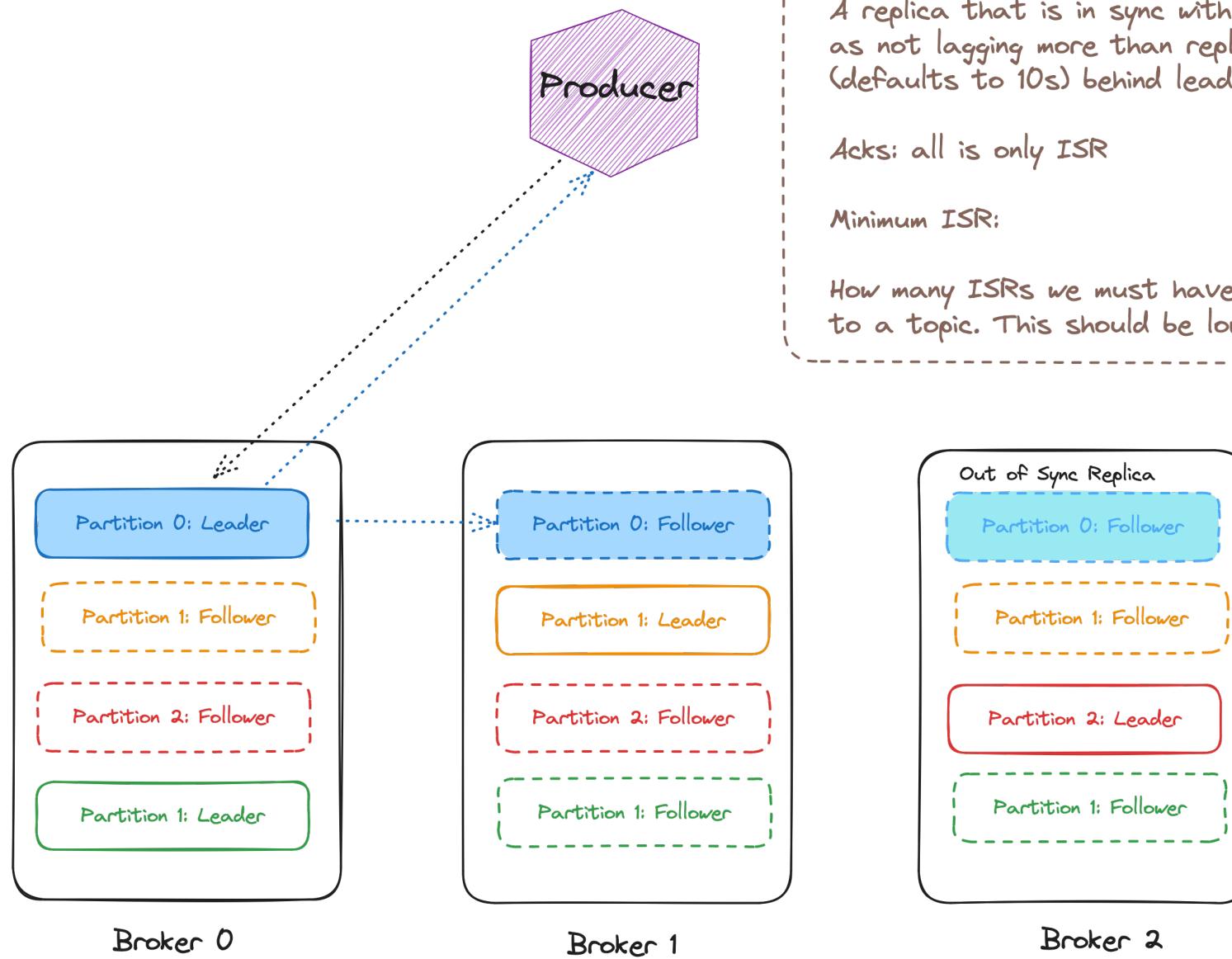


Acks = All is leader + followers. Wait for an acknowledgement from the server that the write has happened and has been propagated to all replicas.

Risk => performance, as it takes time to write to all replicas.



## In-Sync Replicas



### In-Sync Replica (ISR):

A replica that is in sync with the leader defined as not lagging more than `replica.lag.time.max.ms` (defaults to 10s) behind leader.

Acks: all is only ISR

Minimum ISR:

How many ISRs we must have before we can publish to a topic. This should be lower than nodes in cluster

# Reliable Producers

## Reliability

```
acks = all  
retries = 3  
retry.backoff.ms = 200  
max.in.flights.requests.per.connection = 1
```

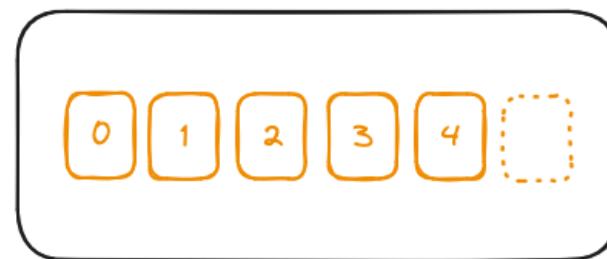
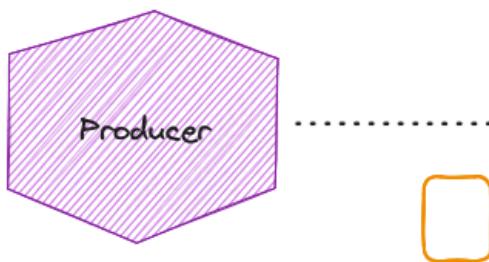
How many brokers need to confirm; alternative is leader or none

How many times to retry if we fail

How many ms between retries  
(default is 100ms)

How many requests can be "in flight"  
Set this to 1 to avoid retries leaving  
you out of order.

TOPIC FOO



# Reliably Producing a Message in C#

Configure how the SDK gives us feedback on producing messages

```
1 using Confluent.Kafka;
2
3 namespace Transmogrification;
4
5 public class Dispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
6 {
7     private readonly IProducer<string, string> _producer = new ProducerBuilder<string, string>(producerConfig)
8         .SetLogHandler((_, message) =>
9             Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
10        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
11        .Build();
12
13     public void Transmogrify(TransmogrificationSettings settings)
14     {
15         ...
16     }
17
18     public void Dispose()
19     {
20         _producer.Dispose();
21     }
22 }
```



```
1 var producerConfig = new ProducerConfig
2 {
3     BootstrapServers = "localhost:9092",
4     EnableDeliveryReports = true,
5     ClientId = Dns.GetHostName(),
6     Acks = Acks.Leader,
7     //EnableIdempotence = true
8     MessageSendMaxRetries = 3,
9     MaxInFlight = 1,
10};
```

# Reliably Producing a Message in C#

```
1  public void Transmogrify(TransmogrificationSettings settings)
2  {
3      try
4      {
5          Action<DeliveryReport<string, string>> handler = report =>
6          {
7              //we use an outbox to be able to retry sending the message if it fails
8              //we don't show a sweeper in this example, but it would be a separate process that would
9              //periodically check the outbox for messages that haven't been persisted yet and retry sending them
10             outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
11         };
12
13         var message = new Message<string, string>
14         {
15             Key = settings.Name,
16             Value = settings.Transformation
17         };
18
19         outbox.Add(topic, message.Key, message);
20
21         //This runs asynchronously - it won't block the thread, instead it will call the handler when the message is sent
22         _producer.Produce(topic, message, handler);
23     }
24     finally
25     {
26         _producer.Flush(TimeSpan.FromSeconds(10));
27     }
28 }
```

Mark the outbox with the status of the produce on callback

Use an “outbox” for “Transactional Messaging” [Simplified]

Don’t block on ack, instead set a callback to assess the delivery report

# Reliably Producing a Message in C#

Configure how the SDK gives us feedback on producing messages

Mark the outbox with the status of the produce on callback

Use an outbox for Transactional Messaging

Don't block on ack, instead set a callback to assess the delivery report

```
1 using Confluent.Kafka;
2
3 namespace Transmogrification;
4
5 public class Dispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
6 {
7     private readonly IProducer<string, string> _producer = new ProducerBuilder<string, string>(producerConfig)
8         .SetLogHandler((_, message) =>
9             Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
10        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
11        .Build();
12
13     public void Transmogrify(TransmogrificationSettings settings)
14     {
15         try
16         {
17             Action<DeliveryReport<string, string>> handler = report =>
18                 outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
19
20         };
21
22         var message = new Message<string, string>
23         {
24             Key = settings.Name,
25             Value = settings.Transformation
26         };
27
28         outbox.Add(topic, message.Key, message);
29
30         _producer.Produce(topic, message, handler);
31     }
32     finally
33     {
34         _producer.Flush(TimeSpan.FromSeconds(10));
35     }
36 }
37
38 public void Dispose()
39 {
40     _producer.Dispose();
41 }
```

```
1 var producerConfig = new ProducerConfig
2 {
3     BootstrapServers = "localhost:9092",
4     EnableDeliveryReports = true,
5     ClientId = Dns.GetHostName(),
6     Acks = Acks.Leader,
7     //EnableIdempotence = true
8     MessageSendMaxRetries = 3,
9     MaxInFlight = 1,
10 };
```

# Async Reliably Producing a Message in C#

Instead of a callback using  
async/await for the delivery  
report – will not return until  
delivery report is produced,  
which depends on acks.

```
1 using Confluent.Kafka;
2
3 namespace Transmogrification;
4
5 public class AsyncDispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
6 {
7     private readonly IProducer<string, string> _producer = new ProducerBuilder<string, string>(producerConfig)
8         .SetLogHandler(_ =>
9             Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
10        .SetErrorHandler(_ => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
11        .Build();
12
13     public async Task Transmogrify(TransmogrificationSettings settings, CancellationToken cancellationToken = default)
14     {
15         try
16         {
17             var message = new Message<string, string>
18             {
19                 Key = settings.Name,
20                 Value = settings.Transformation
21             };
22
23             outbox.AddTopic(message.Key, message);
24
25             var report = await _producer.ProduceAsync(topic, message, cancellationToken);
26
27             outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
28         }
29         finally
30         {
31             _producer.Flush(TimeSpan.FromSeconds(10));
32         }
33     }
34
35     public void Dispose()
36     {
37         _producer.Dispose();
38     }
39 }
```

In principle, at scale, the throughput of  
async/await will be lower than using  
Produce with a callback

# Demo Three: Reliable Producer

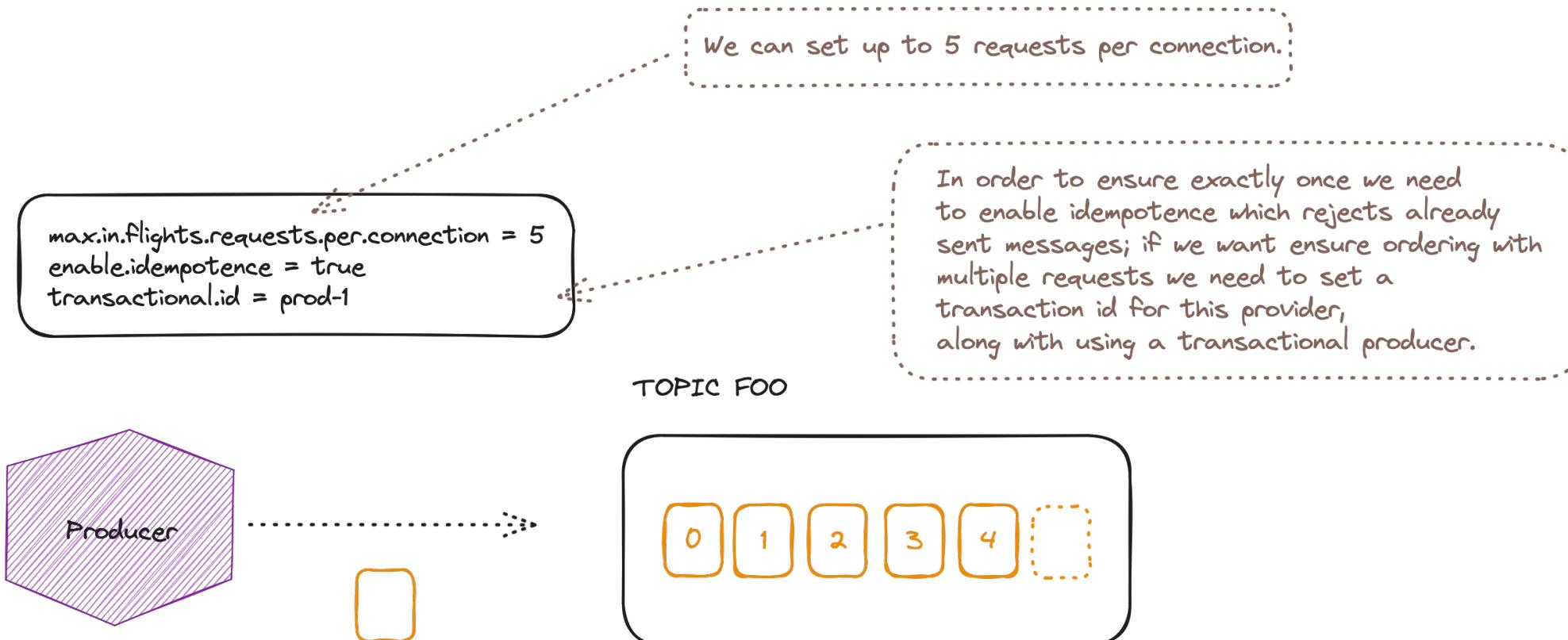
YOU STEP INTO THIS CHAMBER,  
SET THE APPROPRIATE DIALS,  
AND IT TURNS YOU INTO  
WHATEVER YOU'D LIKE TO BE.



Calvin and Hobbes – Bill Watterson

```
1 using Confluent.Kafka;
2
3 namespace Transmogrification;
4
5 public class Dispatcher(string topic, ProducerConfig producerConfig, IOutbox outbox) : IDisposable
6 {
7     private readonly IProducer<string, string> _producer = new ProducerBuilder<string, string>(producerConfig)
8         .SetLogHandler((_, message) =>
9             Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
10        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
11        .Build();
12
13     public void Transmogrify(TransmogrificationSettings settings)
14     {
15         try
16         {
17             Action<DeliveryReport<string, string>> handler = report =>
18             {
19                 outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
20             };
21
22             var message = new Message<string, string>
23             {
24                 Key = settings.Name,
25                 Value = settings.Transformation
26             };
27
28             outbox.Add(topic, message.Key, message);
29
30             _producer.Produce(topic, message, handler);
31         }
32         finally
33         {
34             _producer.Flush(TimeSpan.FromSeconds(10));
35         }
36     }
37
38     public void Dispose()
39     {
40         _producer.Dispose();
41     }
42 }
```

## Transactional Reliability



# Exactly Once Producing a Message in C#

```
1  try
2  {
3      _producer.InitTransactions(TimeSpan.FromSeconds(10));
4      _producer.BeginTransaction();
5
6      Action<DeliveryReport<string, string>> handler = report =>
7      {
8          //we use an outbox to be able to retry sending message if it fails
9          //we don't show a sweeper in this example, but it would be a separate process that would
10         //periodically check the outbox for messages that haven't been persisted yet and retry sending them
11         outbox.MarkStatus(report.Topic, report.Key, report.Partition, report.Status, report.Timestamp);
12     };
13
14     var message = new Message<string, string>
15     {
16         Key = settings.Name,
17         Value = settings.Transformation
18     };
19
20
21     //We could send multiple messages as part of the transaction, but we only send one
22     _producer.Produce(topic, message, handler);
23
24     //If the outbox add fails, then the transaction will be aborted and the message won't be sent
25     outbox.Add(topic, message.Key, message);
26
27     _producer.CommitTransaction(TimeSpan.FromSeconds(10));
28 }
29 catch (ProduceException<string, string> e)
30 {
31     Console.WriteLine($"Delivery failed: {e.Error.Reason}");
32     _producer.AbortTransaction(TimeSpan.FromSeconds(10));
33 }
34 catch (KafkaException e)
35 {
36     Console.WriteLine($"Fatal error: {e.Error.Reason}");
37     _producer.AbortTransaction(TimeSpan.FromSeconds(10));
38 }
```

Mainly intended for a streaming app that does:  
- consume  
- process  
- read

Where it is important that we get one produce per offset committed from the stream we are reading

We can use a transaction with begin, commit/abort to ensure atomic-to-all partitions/only once production of a message to Kafka

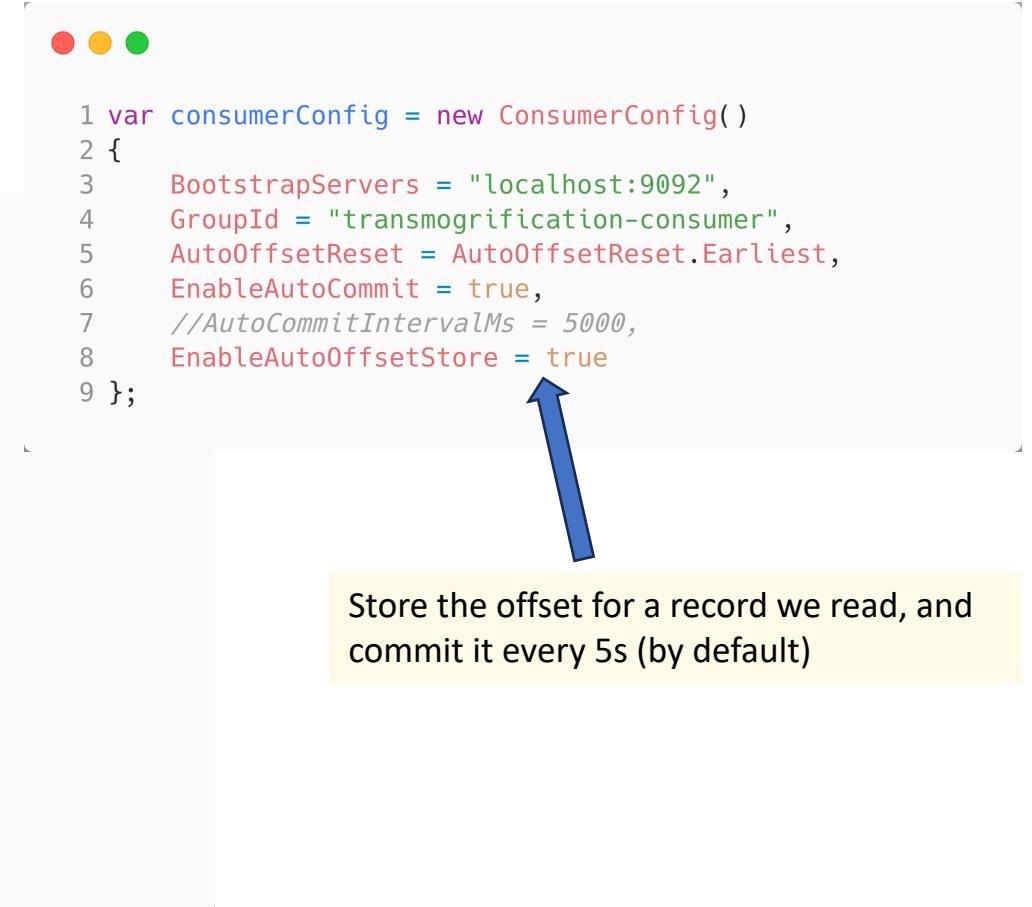
```
1 var producerConfig = new ProducerConfig
2 {
3     BootstrapServers = "localhost:9092",
4     EnableDeliveryReports = true,
5     ClientId = Dns.GetHostName(),
6     Acks = Acks.All,
7     EnableIdempotence = true,
8     TransactionalId = "transmogrification-1",
9     TransactionTimeoutMs = 30000,
10    };
11 }
```

Idempotence requires Acks.All, sets MaxInFlight 5, MAX\_INT retries

# Reliable Consumers

# Auto Store and Commit Offsets in C#

```
1  try
2  {
3      _consumer.Subscribe(topic);
4
5      while (true)
6      {
7          var consumeResult = _consumer.Consume(cancellationToken);
8
9          if (consumeResult.IsPartitionEOF)
10         {
11             await Task.Delay(1000, cancellationToken);
12             continue;
13         }
14
15         var dataType = translator(consumeResult.Message);
16         var result = handler(dataType);
17     }
18 }
```



```
1 var consumerConfig = new ConsumerConfig()
2 {
3     BootstrapServers = "localhost:9092",
4     GroupId = "transmogrification-consumer",
5     AutoOffsetReset = AutoOffsetReset.Earliest,
6     EnableAutoCommit = true,
7     //AutoCommitIntervalMs = 5000,
8     EnableAutoOffsetStore = true
9 };
```

A blue arrow points from the explanatory text at the bottom right to the line of code `EnableAutoOffsetStore = true` in the configuration block.

Store the offset for a record we read, and commit it every 5s (by default)

If we fail to translate, or fail to handle the record, (including a crash) we will have committed the offset and will not re-attempt processing the record

# Auto Commit, Manually Store Offsets in C#

```
1  try
2  {
3      _consumer.Subscribe(topic);
4
5      while (true)
6      {
7          var consumeResult = _consumer.Consume(cancellationToken);
8
9          if (consumeResult.IsPartitionEOF)
10         {
11             await Task.Delay(1000, cancellationToken);
12             continue;
13         }
14
15         var dataType = translator(consumeResult.Message);
16
17         var result = handler(dataType);
18         if (result.Success)
19         {
20             _consumer.StoreOffset(consumeResult);
21         }
22     }
23 }
```

```
1 var consumerConfig = new ConsumerConfig()
2 {
3     BootstrapServers = "localhost:9092",
4     GroupId = "transmogrification-consumer",
5     AutoOffsetReset = AutoOffsetReset.Earliest,
6     EnableAutoCommit = true,
7     EnableAutoOffsetStore = false
8 };
```

EnableAutoOffsetStore = false  
Manually store offsets to the consumer's cache; autocommit will write on a background thread

This means we only store processed records (every 5s)

# Manually Commit, Manually Store Offsets in C#

```
1  try
2  {
3      _consumer.Subscribe(topic);
4
5      while (true)
6      {
7          var consumeResult = _consumer.Consume(cancellationToken);
8
9          if (consumeResult.IsPartitionEOF)
10         {
11             await Task.Delay(1000, cancellationToken);
12             continue;
13         }
14
15         var dataType = translator(consumeResult.Message);
16
17         var result = handler(dataType);
18         if (result.Success)
19         {
20             //We don't want to commit unless we have successfully handled the message
21             //Commit directly. Normally we would want to batch these up, but for the demo we
22             //will
23             //commit after each message
24             _consumer.Commit(consumeResult);
25         }
26     }
}
```

```
1 var consumerConfig = new ConsumerConfig()
2 {
3     BootstrapServers = "localhost:9092",
4     GroupId = "transmogrification-consumer",
5     AutoOffsetReset = AutoOffsetReset.Earliest,
6     EnableAutoCommit = false,
7     EnableAutoOffsetStore = false
8 };
```

EnableAutoOffsetStore = false  
EnableAutoCommit = false  
We choose to manually store the commits

Manually store commits with:  
`_consumer.Commit(consumeResult)`

# Partition Revoke and Offsets in C#



```
1 private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig)
2     .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}"))
3     .SetLogHandler((_, lm) => Console.WriteLine($"Facility: {lm.Facility} Level: {lm.Level} Log: {lm.Message}"))
4     .SetPartitionsRevokedHandler((c, partitions) => partitions.ForEach(p => c.StoreOffset(p)))
5     .Build();
```

Our strategy when a partition is revoked (a new consumer joins the group) must match the strategy we use when we have that partition, and ensure outstanding offsets will be committed.



```
1 private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig)
2     .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}"))
3     .SetLogHandler((_, lm) => Console.WriteLine($"Facility: {lm.Facility} Level: {lm.Level} Log: {lm.Message}"))
4     .SetPartitionsRevokedHandler((c, partitions) => c.Commit(partitions))
5     .Build();
```

# Demo Five: Reliably Consume A Message

YOU STEP INTO THIS CHAMBER,  
SET THE APPROPRIATE DIALS,  
AND IT TURNS YOU INTO  
WHATEVER YOU'D LIKE TO BE.



Calvin and Hobbes – Bill Watterson

```
● ● ●

1 using Confluent.Kafka;
2 using Spectre.Console;
3
4 namespace Transmogrifier;
5
6 public record HandleResult(bool Success);
7
8 public class MessagePump(string topic, ConsumerConfig consumerConfig)
9 {
10    private readonly IConsumer<string, string> _consumer = new ConsumerBuilder<string, string>(consumerConfig)
11        .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}"))
12        .SetLogHandler((_, lm) => Console.WriteLine($"Facility: {lm.Facility} Level: {lm.Level} Log: {lm.Message}"))
13        .SetPartitionsRevokedHandler((c, partitions) => partitions.ForEach(p => c.StoreOffset(p)))
14        .Build();
15
16    public async Task Run<TDataType>(
17        Func<Message<string, string>, TDataType> translator,
18        Func<TDataType, HandleResult> handler,
19        CancellationToken cancellationToken = default)
20    {
21        try
22        {
23            _consumer.Subscribe(topic);
24
25            while (true)
26            {
27                var consumeResult = _consumer.Consume(cancellationToken);
28
29                if (consumeResult.IsPartitionEOF)
30                {
31                    await Task.Delay(1000, cancellationToken);
32                    continue;
33                }
34
35                var dataType = translator(consumeResult.Message);
36
37                var result = handler(dataType);
38                if (result.Success)
39                {
40                    _consumer.StoreOffset(consumeResult);
41                }
42            }
43        }
44        catch(ConsumeException e)
45        {
46            AnsiConsole.WriteLine(e);
47        }
48        catch (OperationCanceledException)
49        {
50            //Pump was cancelled, exit
51        }
52        finally
53        {
54            _consumer.Close();
55        }
56    }
57 }
```

# Offset Reset

# Offset Reset in C#



```
1 var consumerConfig = new ConsumerConfig()
2 {
3     BootstrapServers = "localhost:9092",
4     GroupId = "transmogrification-consumer",
5     AutoOffsetReset = AutoOffsetReset.Earliest,
6     EnableAutoCommit = true,
7     EnableAutoOffsetStore = false
8 };
```

**AutoOffsetReset** indicates what we should do if there is no offset for this GroupId in the \_consumer\_offsets topic.

*Earliest* – read from the earliest offset in the log

*Latest* – read any new messages appended to the log

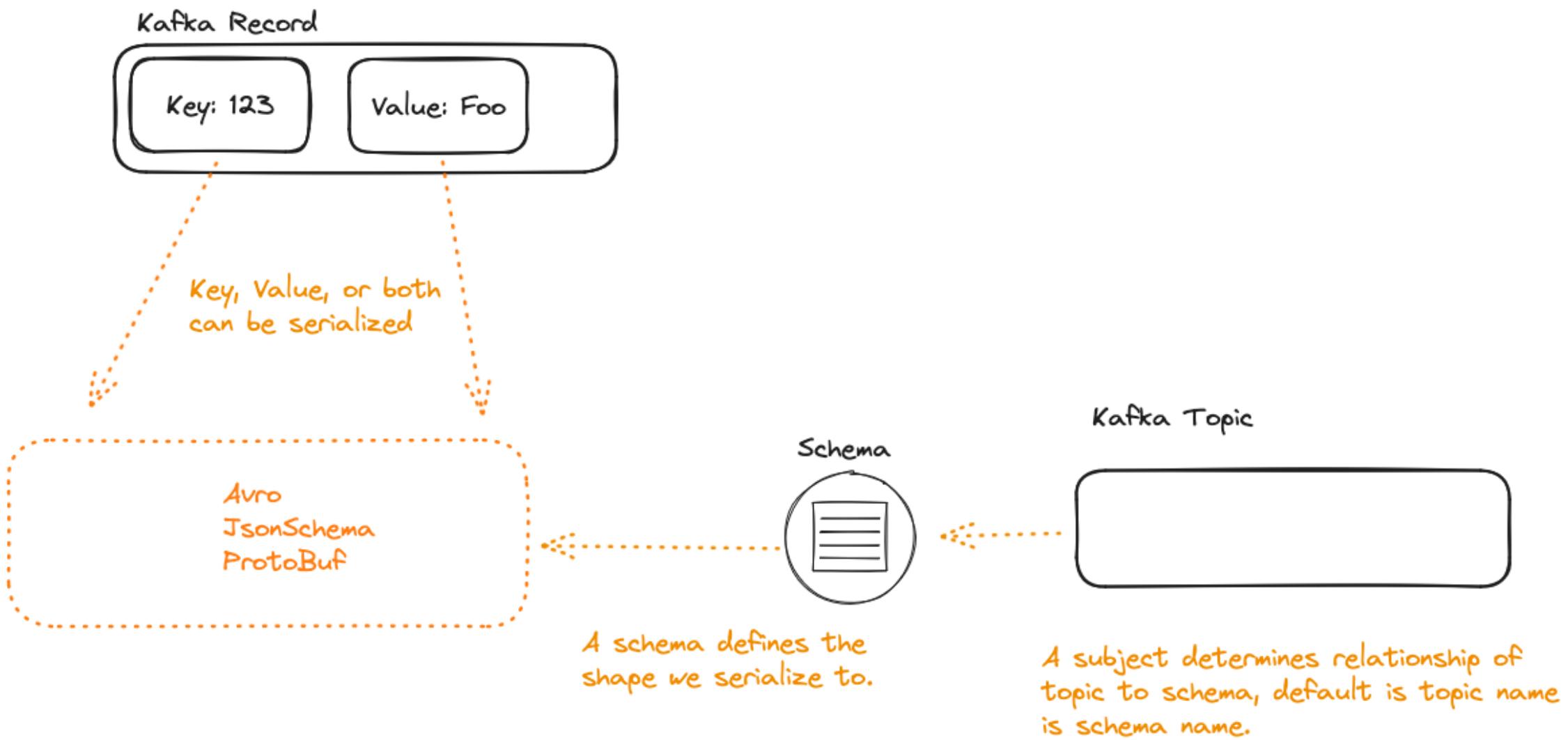


# Group Id and Offset Reset in C#

```
1 var consumerConfig = new ConsumerConfig()
2 {
3     BootstrapServers = "localhost:9092",
4     GroupId = "transmogrification-consumer-2",
5     AutoOffsetReset = AutoOffsetReset.Earliest,
6     EnableAutoCommit = true,
7     EnableAutoOffsetStore = false
8 };
```

If you need to reset the offset to the beginning, so as to read the stream again, the easiest option is use a new **GroupId**.

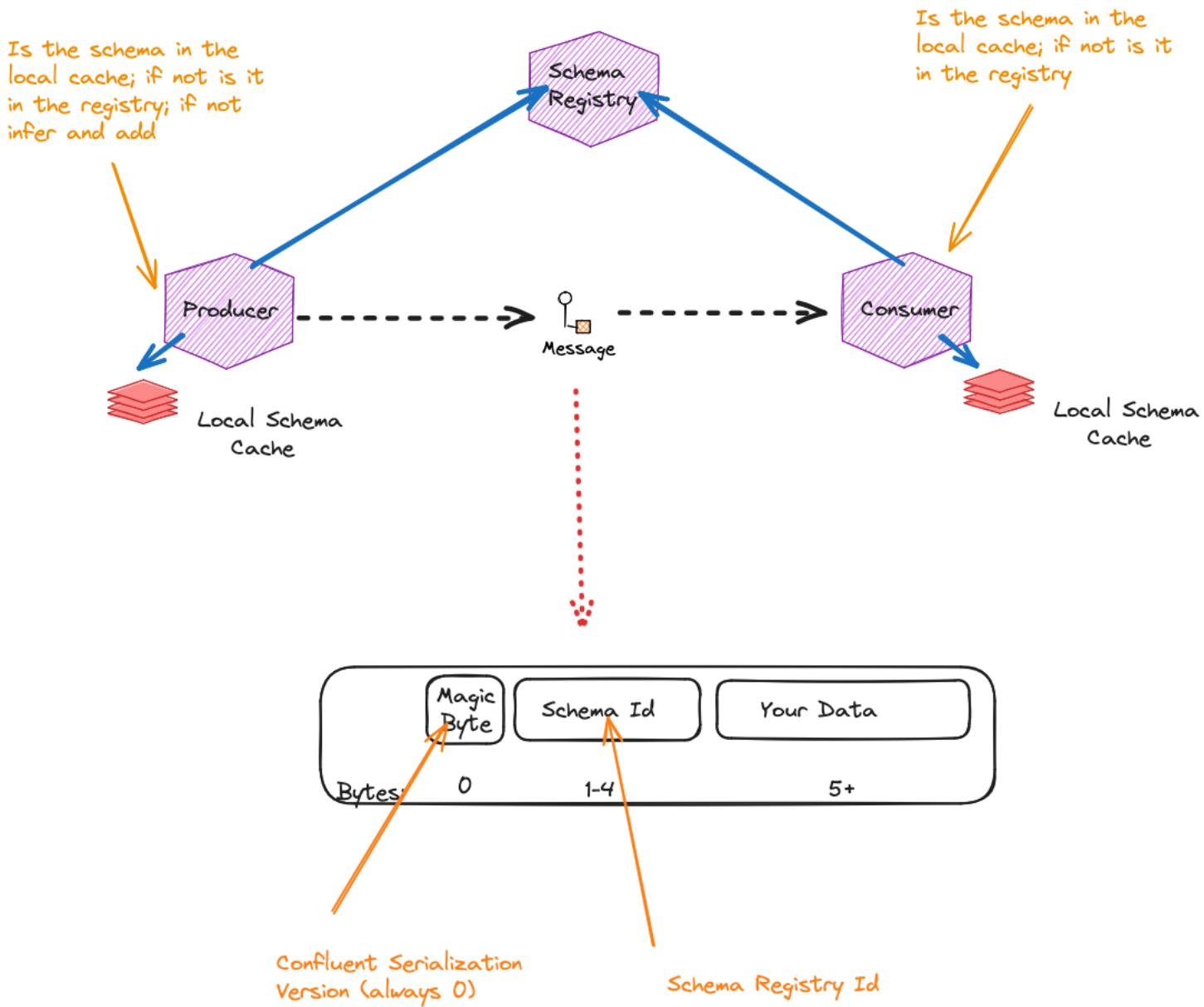
# Serializers, De-serializers & Schema Registry



## Schema Compatibility

Compatibility Type	Changes Allowed	Check Against which Version	Upgrade who first?
Backward Transitive	Delete fields Add optional fields	All previous versions	Consumers
Forward Transitive	Add fields Delete optional fields	All previous versions	Producers
Full	Add optional fields	Last version	Any order
Full Transitive	Add optional fields Delete optional fields	All previous versions	Any order
None	All changes are accepted	None	

## Kafka and the Schema Registry



# Producing with a Schema Registry in C#

We need to pass in a configuration for the Schema registry

```
1 public Dispatcher(string topic, ProducerConfig producerConfig, SchemaRegistryConfig schemaRegistryConfig, IOutbox outbox)
2 {
3     _topic = topic;
4     _outbox = outbox;
5     _schemaRegistryClient = new CachedSchemaRegistryClient(schemaRegistryConfig);
6     var serializerConfig = new JsonSerializerConfig { BufferBytes = 100 };
7
8     _producer = new ProducerBuilder<string, TransmogrificationSettings>(producerConfig)
9         .SetLogHandler((_, message) =>
10             Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}"))
11         .SetErrorHandler((_, e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}"))
12         .SetValueSerializer(new JsonSerializer<TransmogrificationSettings>(_schemaRegistryClient, serializerConfig).AsSyncOverAsync())
13         .Build();
14 }
```

We use the Confluent Serdes Serializer that passes in the schema registry. It will register our schema if none exists.

Note that it uses NJsonSchema for markup and Newtonsoft for JSON serialization, under the hood

# Demo Six: Schema Registry in C#

YOU STEP INTO THIS CHAMBER,  
SET THE APPROPRIATE DIALS,  
AND IT TURNS YOU INTO  
WHATEVER YOU'D LIKE TO BE.



Calvin and Hobbes – Bill Watterson

```
1 public Dispatcher(string topic, ProducerConfig producerConfig, SchemaRegistryConfig schemaRegistryConfig, IOutbox outbox)
2 {
3     _topic = topic;
4     _outbox = outbox;
5     _schemaRegistryClient = new CachedSchemaRegistryClient(schemaRegistryConfig);
6     var serializerConfig = new JsonSerializerConfig { BufferBytes = 100 };
7
8     _producer = new ProducerBuilder<string, TransmogrificationSettings>(producerConfig)
9         .SetLogHandler(_ , message) =>
10            Console.WriteLine($"Facility: {message.Facility}-{message.Level} Message: {message.Message}")
11            .SetErrorHandler(_ , e) => Console.WriteLine($"Error: {e.Reason}. Is Fatal: {e.IsFatal}")
12            .SetValueSerializer(new JsonSerializer<TransmogrificationSettings>(_schemaRegistryClient, serializerConfig).AsSyncOverAsync())
13            .Build();
14 }
```

# Diving Deeper

## Links

The Sample Code: <https://github.com/iancooper/KafkaForDotNetDevs>

Brighter: <https://github.com/BrighterCommand/Brighter>

This Presentation: <https://github.com/iancooper/Presentations>

## Other Topics

Kafka Connect: Turn a data source into a Kafka stream and vice versa

KSQL: Run queries on Kafka Streams

Streams Applications: Run queries on Kafka Streams in Code (C#

Community Support only)

Alternatives to Connect: Flink, Benthos, etc

End