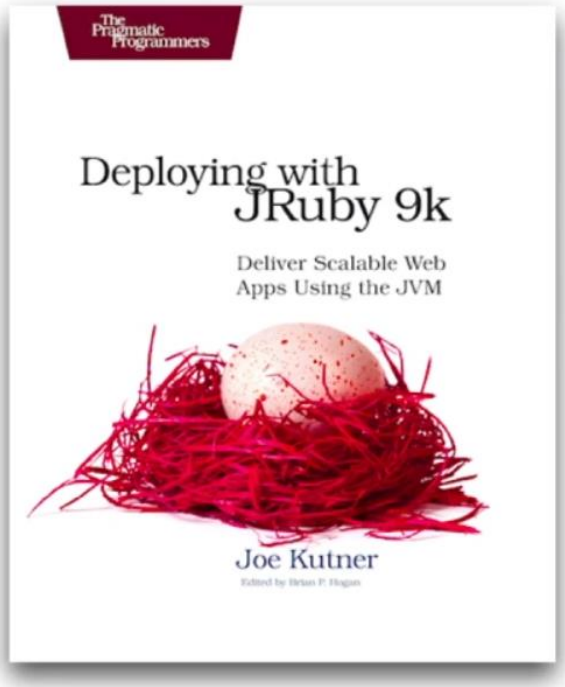
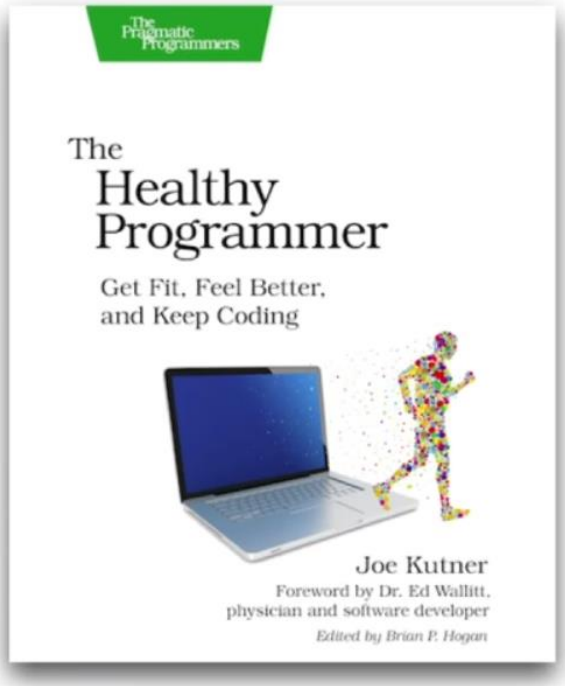
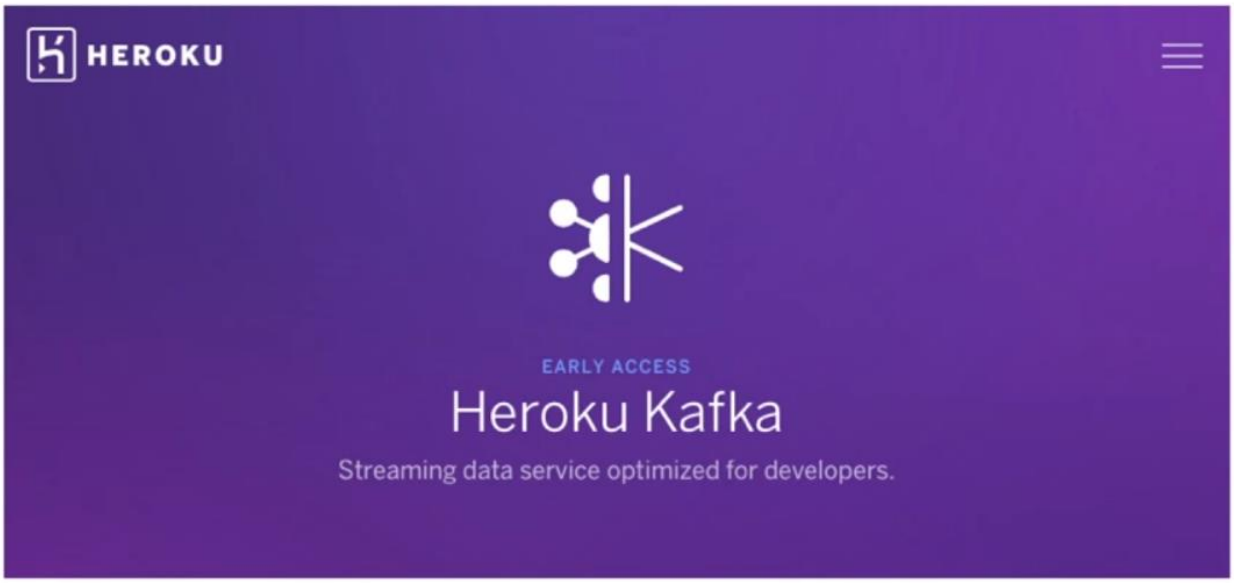
I Can't Believe It's Not a Queue!®

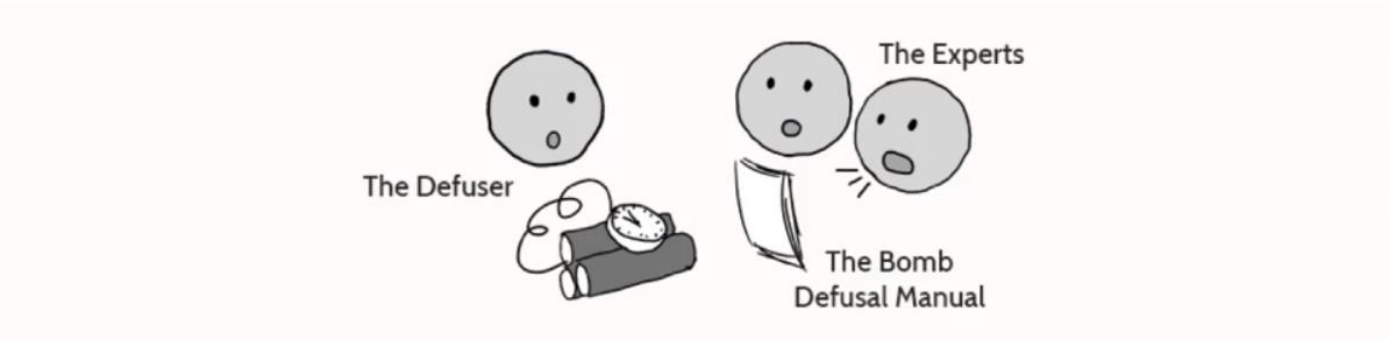
Joe Kutner
@codefinger



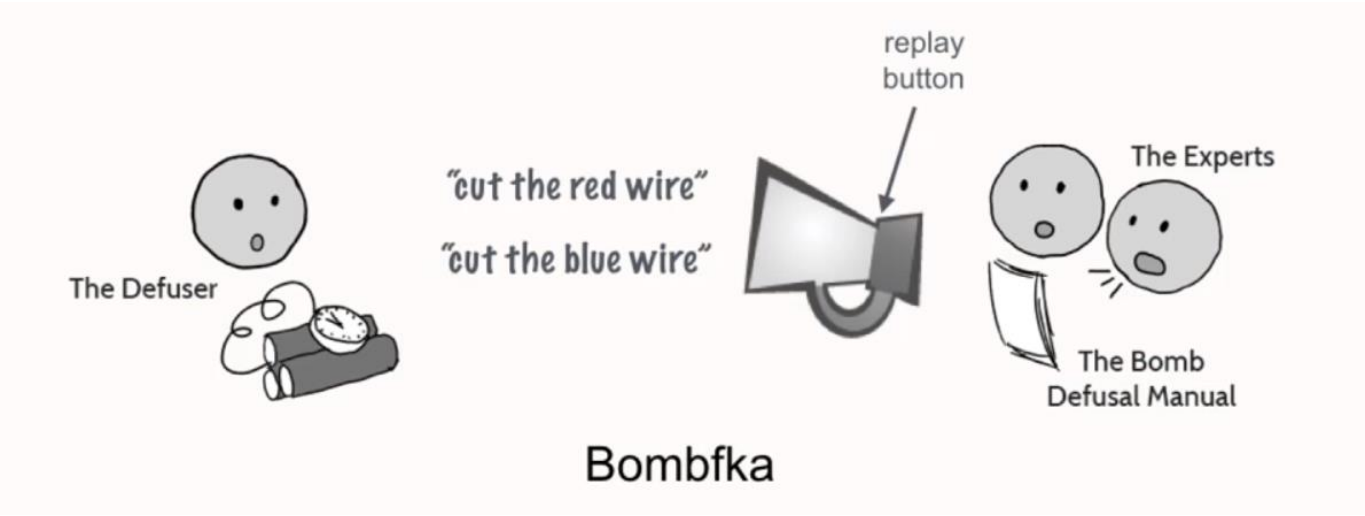




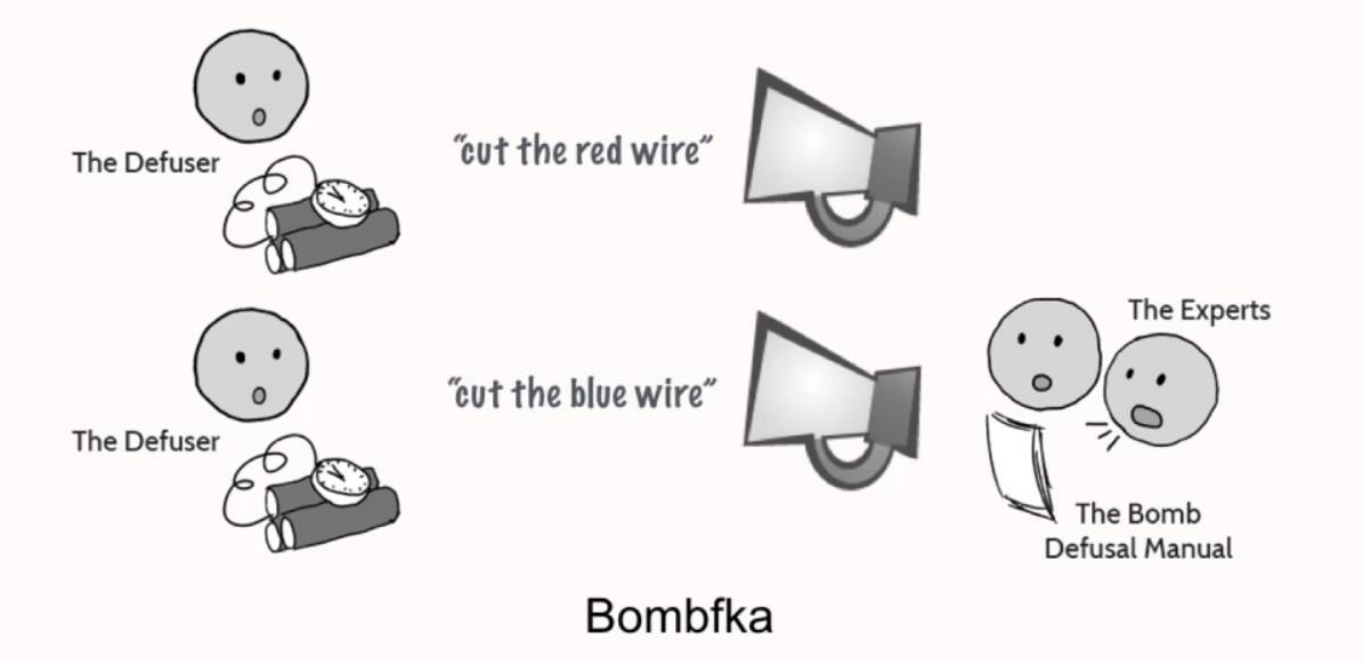
This is Kafka as a service based on Heroku's use of its data and the events generated by all the systems



Things that come out are not in the order you put them in



This megaphone preserves order and the messages are there forever and can be replayed



You can even have multiple megaphones

Agenda

- What is Kafka?
- Kafka + Spring
- Metrics Example
- How Heroku uses Kafka

We will also see an example app that calculates metrics using Kafka

What is Kafka?

Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.

Distributed

Publish

Subscribe

Messaging

Fast

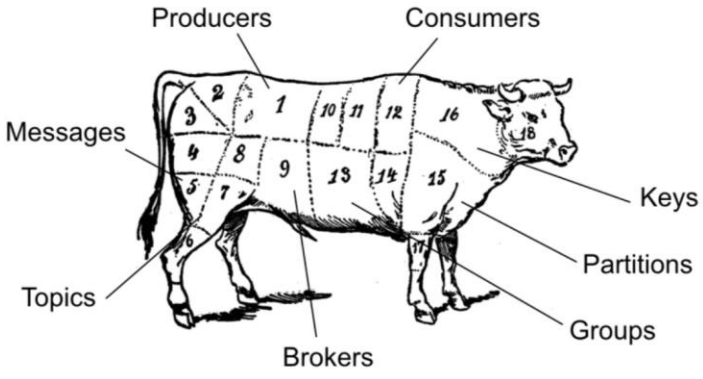
Scalable

Durable

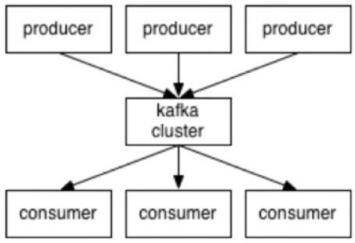
“hundreds of thousands to millions of messages a second on a small cluster”

Tom Crayford
Heroku Kafka

Know Your Cuts of Kafka



Producers & Consumers



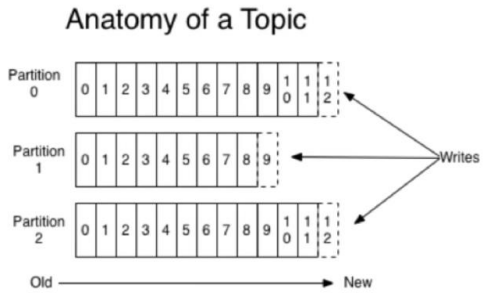
The difference in Kafka is the contracts between the producers and consumers clients and the Kafka cluster. The consumers also have a synchronous connection.

Messages

(Byte Array)

- Header
- Key
- Value

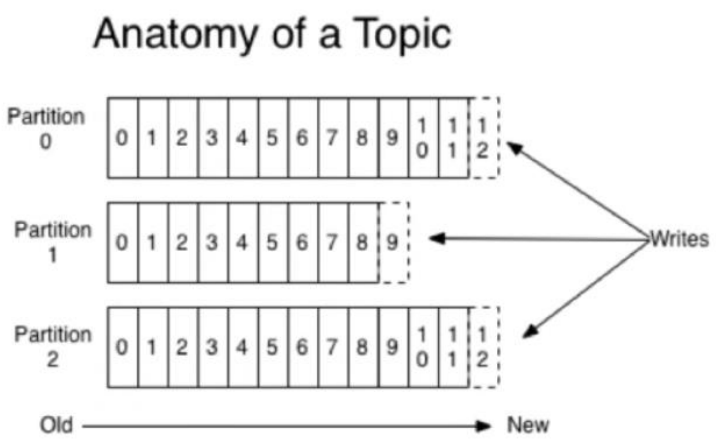
Messages feed into Topics



In Kafka, topics are partitioned and the partitions are spread across brokers in the cluster.

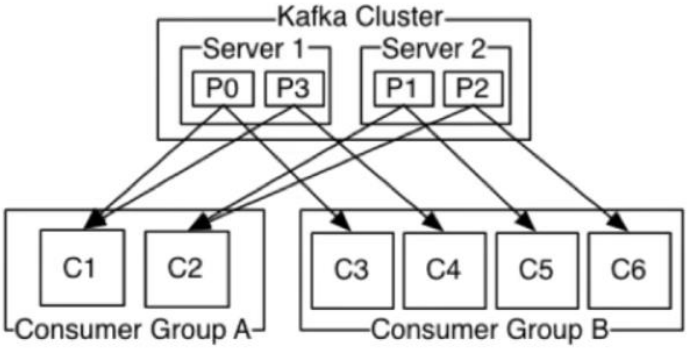
Each Topic Partition is an ordered log of immutable messages, append-only

Offsets



Each consumer has an offset that tells the cluster where that particular consumer is in reading from that partition. The offset is what allows a consumer of Kafka topic to shut down and resume consuming messages from where they left off after restarting.

Consumer Groups



Consumers are arranged into consumer groups that are analogous to consumers in a traditional message queue. We can have multiple consumer groups subscribe to the same topic. Each consumer group is guaranteed to receive all the messages available from a subscribed topic, but the individual consumers in the consumer group do not receive every message from a topic. The individual consumers are assigned a partition of the cluster and they only receive messages for that partition. The partition is what allows the cluster scale and the consumer groups to scale independently.

Kafka Guarantees

- Messages are produced in order
- Messages are consumed in order
- Topics are distributed and replicated

Kafka + Java

Kafka is a JVM technology, so it works using the Java Client library for Kafka. The Java client library for Kafka is low level and requires the use of the Java Concurrency API and not really cloud native. We have frameworks like Spring and Akka that are building higher level frameworks on top of the lower level APIs to make it more comfortable to use Kafka.

Producer API

```
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("key.serializer", StringSerializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());

Producer<String, String> producer = new KafkaProducer<>(props);
```

The Kafka Client Library ***KCL*** for Producers requires that you create a ***Producer*** object that you instantiate with a Java.util properties object to be ***used for configuring the producer*** and must at minimum have 3 properties specified as above. The ***bootstrap.servers*** are the seed brokers that distribute your connection across all the brokers in the cluster, and then a ***key*** and ***value serializers***. Since messages being sent are byte arrays, our clients are going to do the serialization and deserialization, this is why we need to tell the Producer how we want to serialize our messages. If the above case, our messages will be strings and that is why we are using the Kafka provided ***StringSerializer class***.

Producer API

```
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("key.serializer", StringSerializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());

Producer<String, String> producer = new KafkaProducer<>(props);

producer.send(new ProducerRecord<>("my-topic", message1));
```

Once we have the Producer, we create a ProducerRecord for the message that represents both the message we want to send (***message1)*** and the destination topic (***'my-topic'***) as above. We then pass the record to the producer.send() method that will send it to the Kafka cluster

Producer API

```
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("key.serializer", StringSerializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());

Producer<String, String> producer = new KafkaProducer<>(props);

producer.send(new ProducerRecord<>("my-topic", message1));

producer.send(new ProducerRecord<>("my-topic", message2));

producer.send(new ProducerRecord<>("my-topic", message3));

producer.send(...).get();
```

The producer.send() method above is not a synchronous operation, it does not hit the wire once we invoke the send method. The Kafka producer in a background thread will queue up each of the messages being sent and then send them to the broker in batches based on their destination that is determined by what message it is based on type, topic, etc. the ***producer.send(…)*** method returns a Java Future type and we have to use the ***.get()*** method to see what exactly was returned in the Future object using the Concurrency API.

Consumer API

Automatic Offset Committing

```
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

The Consumer API looks similar, you create a consumer and pass it the properties object with the 3 needed values.

Consumer API

Automatic Offset Committing

```
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(singletonList("my-topic"));

while (running.get()) {
    ConsumerRecords<String, String> records = consumer.poll(100);

    for (ConsumerRecord<String, String> record : records) {
        // ...
    }
}
```

Once you get back the ***KafkaConsumer*** instance, you then call the ***consumer.subscribe(…)*** method and give it a topic or a list of topics to listen for. You then invoke the ***consumer.poll(…)*** method (which is a blocking operation) and pass it a timeout as above. This will return back to you a list of ***ConsumerRecords*** that represent the messages and some metadata about the messages like where they came from.

Consumer API

Automatic Offset Committing

```
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(singletonList("my-topic"));

while (running.get()) {
    ConsumerRecords<String, String> records = consumer.poll(100);

    for (ConsumerRecord<String, String> record : records) {
        // ...
    }
}
```

Since poll() is a blocking operation, we typically put it inside some kind of loop with a control variable like ***running*** as an atomic boolean so that we have to control that outside of the thread of execution with the Concurrency API's ***get()*** method. The ***poll()*** method does a lot more than receiving messages, the first time you invoke the ***consumer.poll()*** method, it locates the group coordinator for the cluster and joins this consumer to its ***ConsumerGroup***. The cluster then assigns this consumer a partition or set of partitions and then rebalances all the partition assignments for that ConsumerGroup. It also establishes a heartbeat from the consumer to the cluster so that the cluster knows that that particular consumer is online and working. Finally, after receiving messages, the consumer automatically updates the consumer offset that is stored on the cluster.

Consumer API

Manual Offset Committing

```
props.put("enable.auto.commit", "false");
// ...

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(singletonList("my-topic"));

final int minBatchSize = 200;

while (running.get()) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        buffer.add(record);
    }
    if (buffer.size() >= minBatchSize) {
        insertIntoDb(buffer);
        consumer.commitSync();
        buffer.clear();
    }
}
```

We can also do manual offset committing if we want to do things like guarantee that a message was actually processed before the offset is updated. You simply call the ***consumer.commitSync()*** method after processing the message successfully.

Consumer API

Kafka Consumer is **NOT** threadsafe

By design, you should have one consumer per thread so that you can use the Concurrency API to execute a service to spurn off threads for each of your consumers.

Consumer API

Advanced!

- Per-message offset commit
- Manual Partition Assignment
- Storing Offsets Outside Kafka
- Kafka Streams (since 0.10)

The Consumer API also allows you to do per-message offset committing, manually assign partitions to consumers, you can also store offsets outside of Kafka, and use Kafka Streams from the most recent version of Kafka, this allows you to do single message processing as opposed to the default of processing micro-batches using the poll() method.

Using Kafka with Spring

Producer

```
@SpringBootApplication
@EnableKafka
public class SpringApplicationProducer {
    @Bean
    public KafkaTemplate<Integer, String> kafkaTemplate() {
        return new KafkaTemplate<Integer, String>(producerFactory());
    }

    @Bean
    public ProducerFactory<Integer, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfigs());
    }

    private Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class);
        // ...
        return props;
    }
}
```

Spring has introduced SpringKafka which makes it easier to use the Kafka client. At a high level you will need to use the @EnableKafka annotation in your spring boot app and you then create a KafkaTemplate bean and provide it with a map of properties identical to the Java.util properties we saw earlier.

Using Kafka with Spring

Producer

```
@Autowired
private KafkaTemplate<Integer, String> template;

public void send() throws Exception {
   template.send("my-topic", message);
}
```

For the Producer API, the KafkaTemplate represents your interface into the Kafka cluster, you wire up some other components and then call the template.send() method as above.

Using Kafka with Spring

Consumer

```
@Service
public class MyKafkaListener {

    @KafkaListener(topicPattern = "my-topic")
    public void listen(String message) {
        System.out.println("received: " + message);
    }
}
```

For the Consumer API, the KafkaConsumer introduces the @KafkaListener annotation that you can apply to a method to effectively make it like a callback for the KafkaConsumer. The spring KafkaConsumer is going to handle things like the subscription to messages, the polling for messages, looping around the poll() method, etc. then it will just invoke our callbacks when it receives messages.

Using Kafka with Spring

Consumer

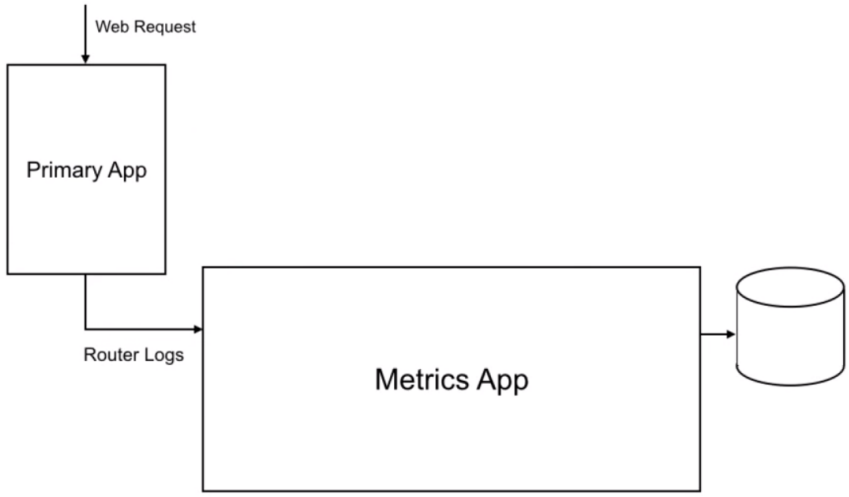
```
@Service
public class MyKafkaListener {

  @KafkaListener(id = “my-listener", topicPartitions = {
        @TopicPartition(topic = "topic1", partitions = { "0", "1" }),
        @TopicPartition(topic = "topic2", partitions = { "0", "1" })
  })
  public void listen(String message) {
    System.out.println("received: " + message);
  }
}
```

The @KafkaListener annotation allows us to set attributes for different types on configurations like assigning specific partitions to this method.

Metrics App Example

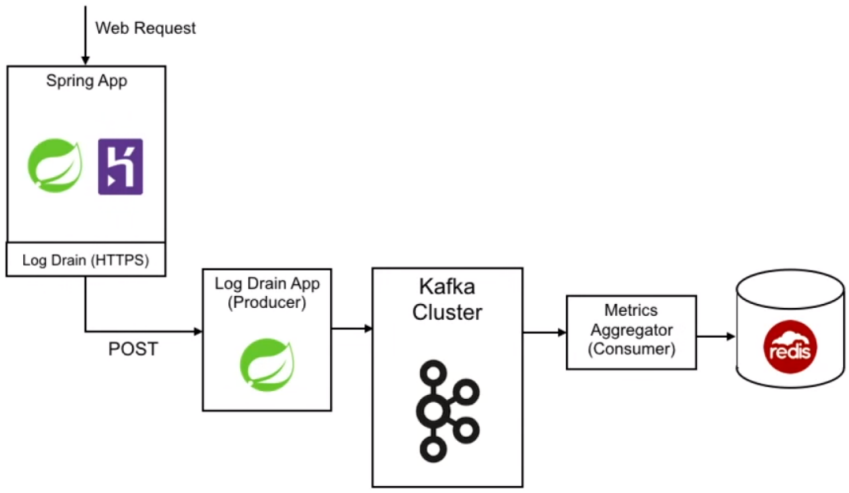
This example application is a Metric calculation app that takes a primary application like your web app that receives web requests, and calculates metrics for that web app for things like the response times based on the requested route, it does this by looking at the applications logs.

Architecture



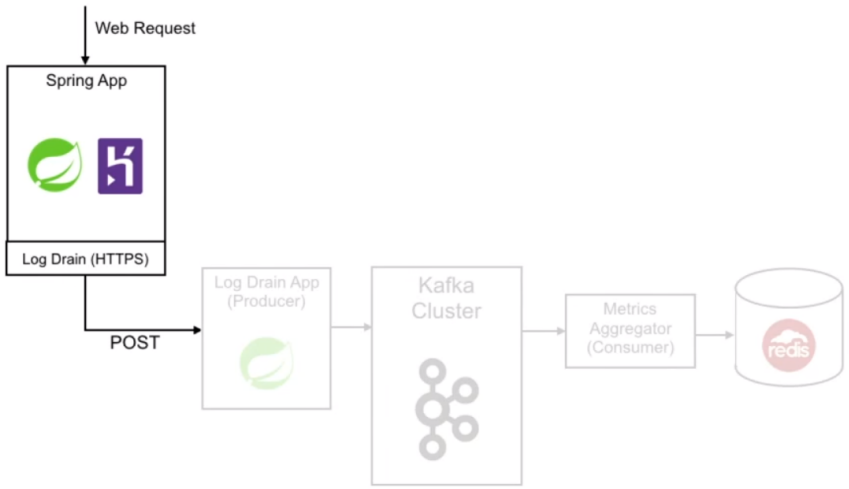
Each web request to the primary app will generate a Router log with the path requested and the request and response times, status codes, etc. We then funnel the logs into our Metrics app which will use Kafka, then roll our metrics calculations up and store them in some other persistence storage.

Architecture



Above is what we get at a lower level, the primary app is a Spring boot app that is receiving web requests.

Architecture



If the Primary app is running on Heroku, the LogDrain is a feature that allows you to consume your application logs over HTTP. You can register a callback URL with the application and then Heroku will invoke that with a POST request for you for each logging statement.

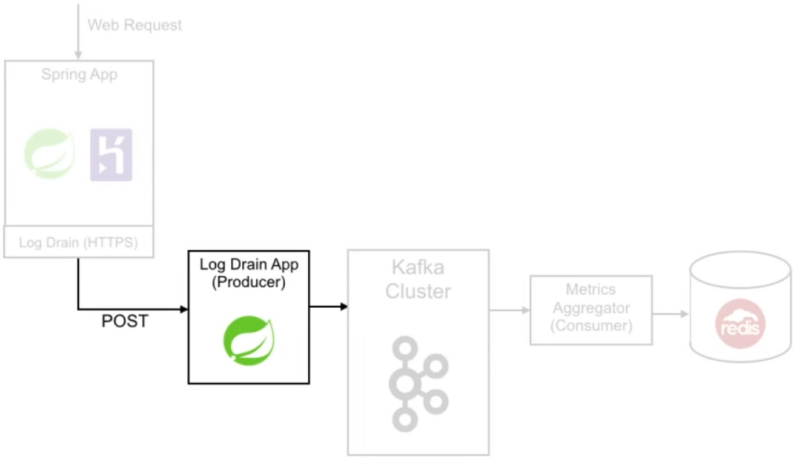
Heroku Log Drains

```
$ heroku drains:add https://<metrics-app>/logs
```

```
242 <158>1 2016-06-20T21:56:57.107495+00:00 host heroku router -
at=info method=GET path="/" host=demodayex.herokuapp.com
request_id=1850b395-c7aa-485c-aa04-7d0894b5f276 fwd="68.32.161.89"
dyno=web.1 connect=0ms service=6ms status=200 bytes=1548
```

You simply provide it the logging endpoint for our metrics app. Each time the primary app receives a web request, it will send a POST request to the registered metrics endpoint a stream that is in syslog formatted text containing data about the web request.

Architecture

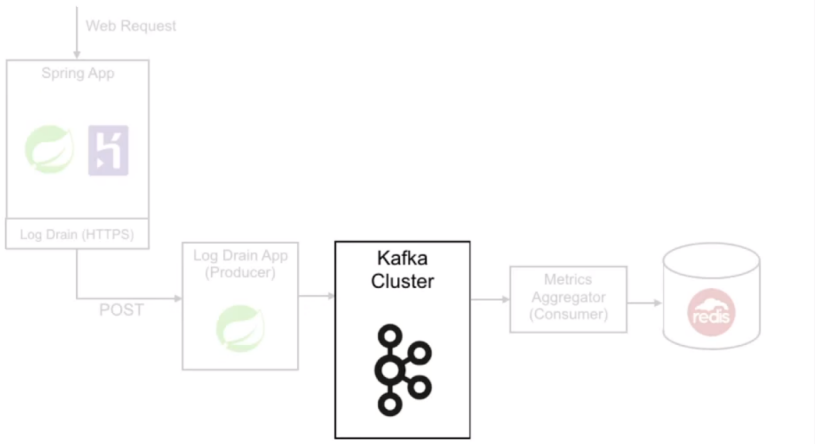


The Metrics app endpoint is a simple Spring boot app that is also a Kafka producer, this app has a single entrypoint called /logs that receives a streaming body as below

```java
@RequestMapping(value = "/logs", method = RequestMethod.POST)
@ResponseBody
public String logs(@RequestBody String body) throws IOException {

    // "application/logplex-1" does not conform to RFC5424.
    // It leaves out STRUCTURED-DATA but does not replace it with
    // a NILVALUE. To workaround this, we inject empty STRUCTURED-DATA.
    String[] parts = body.split("router - ");
    String log = parts[0] + "router - [] " + (parts.length > 1 ? parts[1] : "");

    RFC6587SyslogDeserializer parser = new RFC6587SyslogDeserializer();
    InputStream is = new ByteArrayInputStream(log.getBytes());
    Map<String, ?> messages = parser.deserialize(is);
    ObjectMapper mapper = new ObjectMapper();
    String json = mapper.writeValueAsString(messages);

    template.send("logs", json);

    return "ok";
}
```

The app manages the incoming stream and then use the deserializer to parse the syslog record into a map. We then turn that map into a JSON object and then converts the JSON object into a String. Finally, we send that string to Kafka through the KafkaTemplate using template.send() as above.

Architecture



We put the messages as string into Kafka

Heroku Kafka

```
$ heroku addons:create heroku-kafka
```

← Create the Cluster

This ***$ Heroku addons:create heroku-kafka*** command will provision a Kafka cluster for you and also provision a Zookeeper cluster to use also. It then attaches this Kafka cluster to your application and you will be provided with the seed broker URLs.

Heroku Kafka

```
$ heroku addons:create heroku-kafka
$ heroku plugins:install heroku-kafka
$ heroku kafka:create logs
```

Create the Cluster

Create the Topic

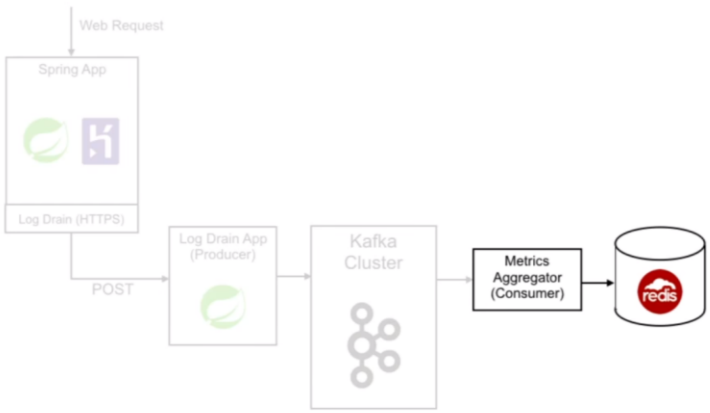
We create a logs topic that we will be sending our logs messages to from the producers.

Heroku Kafka

```
$ heroku kafka:info
=== KAFKA_URL

Name:        kafka-asymmetrical-77749
Created:     2016-06-20 18:21 UTC
Plan:        Beta Dev
Status:      available
Version:     0.9.0.1
Topics:      2 topics (see heroku kafka:list)
Connections: 0 consumers (0 applications)
Messages:    32.0 messages/s
Traffic:     2.25 KB/s in /  2.25 KB/s out
```

We can also inspect the Kafka cluster and get information on a topic as above

Architecture



We then have a Consumer that is a simple metrics aggregator running a s a simple Java process.

Main Consumer Class

```
public class Metrics {
  // ...

  public static void main(String[] args) { /* … */ }

  public Metrics() throws Exception {
    // ...

    URI redisUri = new URI(System.getenv("REDIS_URL"));
    pool = new JedisPool(redisUri);
  }

  private void start() {
    // ...
    running.set(true);
    executor = Executors.newSingleThreadExecutor();
    executor.submit(this::loop);
    stopLatch = new CountDownLatch(1);
  }
}
```

When instantiated, this process will create a pool of connections to ***Redis*** since this is where we are going to store our metrics results. Then the ***start()*** method fires off a new thread for our consumers for thread safety.

Main Consumer Method

```
private void loop() {
  // ...

  consumer = new KafkaConsumer<>(properties);
  consumer.subscribe(singletonList(KafkaConfig.getTopic()));

  while (running.get()) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
      try {
        Map<String,String> recordMap =
            mapper.readValue(record.value(), typeRef);
        Route route = new Route(recordMap);
        receive(route);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

  consumer.close();
  stopLatch.countDown();
}
```

Each of the threads is going to launch with the ***loop()*** method, we then subscribe it to our topic, then use the ***poll()*** method inside the loop with the control variable ***running*** as above.

Main Consumer Method

```
private void loop() {
  // ...

  consumer = new KafkaConsumer<>(properties);
  consumer.subscribe(singletonList(KafkaConfig.getTopic()));

  while (running.get()) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
      try {
        Map<String,String> recordMap =
            mapper.readValue(record.value(), typeRef);
        Route route = new Route(recordMap);
        receive(route);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

  consumer.close();
  stopLatch.countDown();
}
```

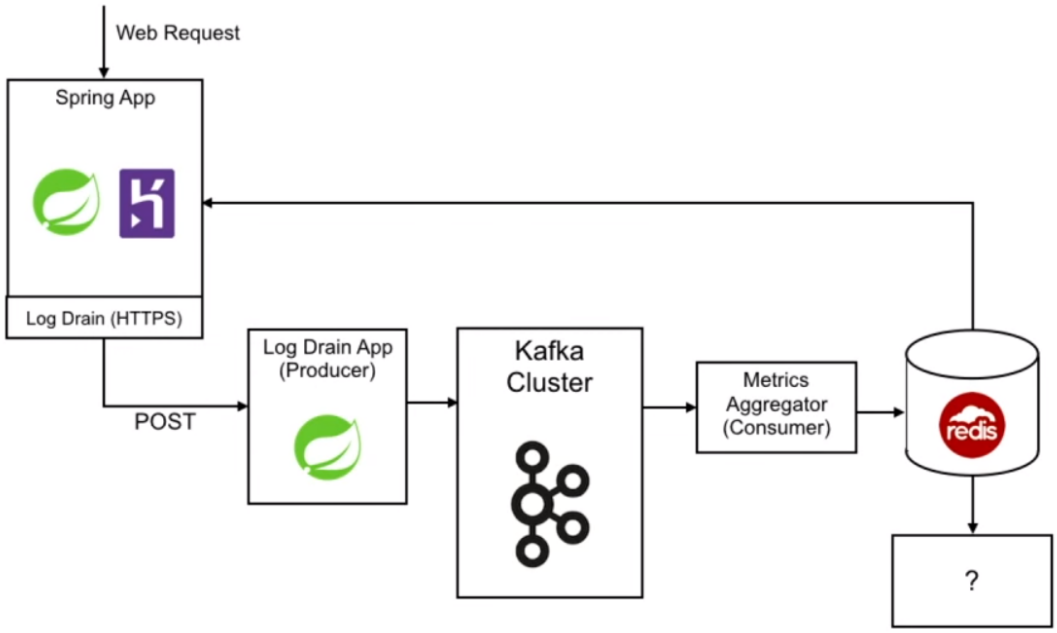
Each time we receive some messages, we will convert those messages into Route objects. A Route object is a simple POJO that we use to represent each of the syslog records. We then pass the route to the receive() method

Update Redis

```
private void receive(Route route) {
    // …

    jedis.hincrBy(key, "sum", value);
    jedis.hincrBy(key, "count", 1);

    Integer sum = Integer.valueOf(jedis.hget(key, "sum"));
    Float count = Float.valueOf(jedis.hget(key, "count"));
    Float avg = sum / count;

    jedis.hset(key, "average", String.valueOf(avg));
}
```

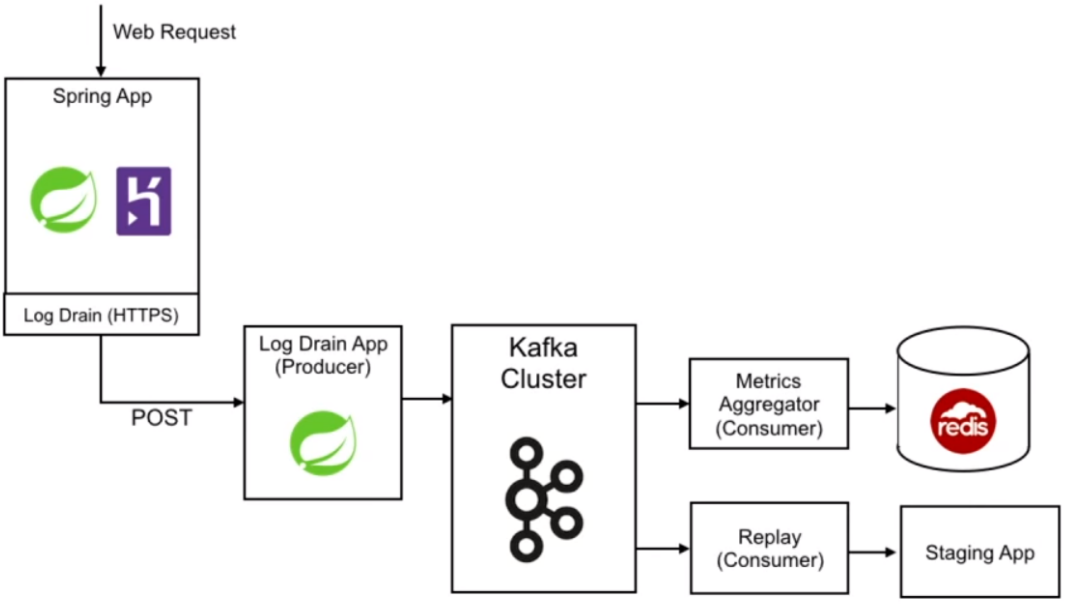
The ***receive()*** method largely is just interfacing with ***Redis*** to recalculate an average response time for that particular request each time we receive a new request on our web app as seen above.

Architecture



Once all our metrics data is in Redis we can then consume it from anywhere, we can have a view in the primary app or some admin app that displays the current metrics.

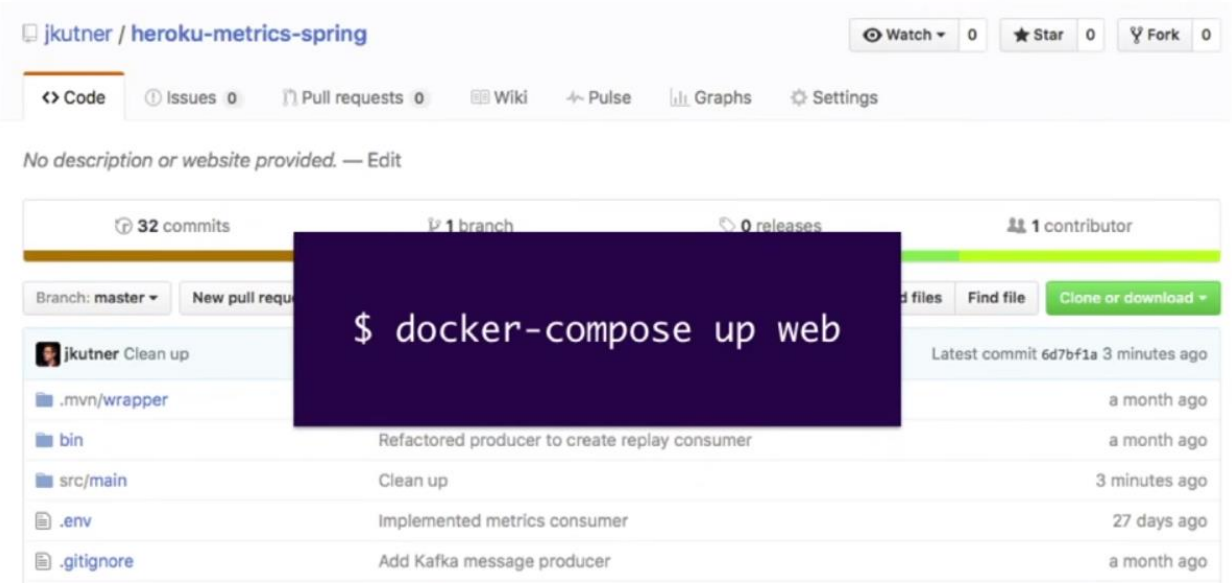
Architecture



We can also reuse the Kafka feed of syslog data, we can have a Replay consumer that can take the log statements and replay the requests against a staging application. This allows us to test our staging application with real production requests without affecting the production application.

Demo App

https://github.com/jkutner/heroku-metrics-spring

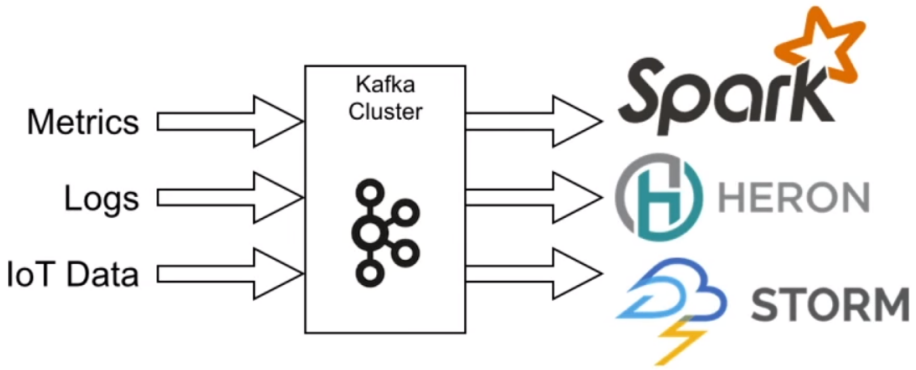


This ***$ docker compose up*** command will standup Kafka, Zookeeper, Redis, and the metrics app

Other Use Cases

- User Activity
- Stream Processing

Stream Processing



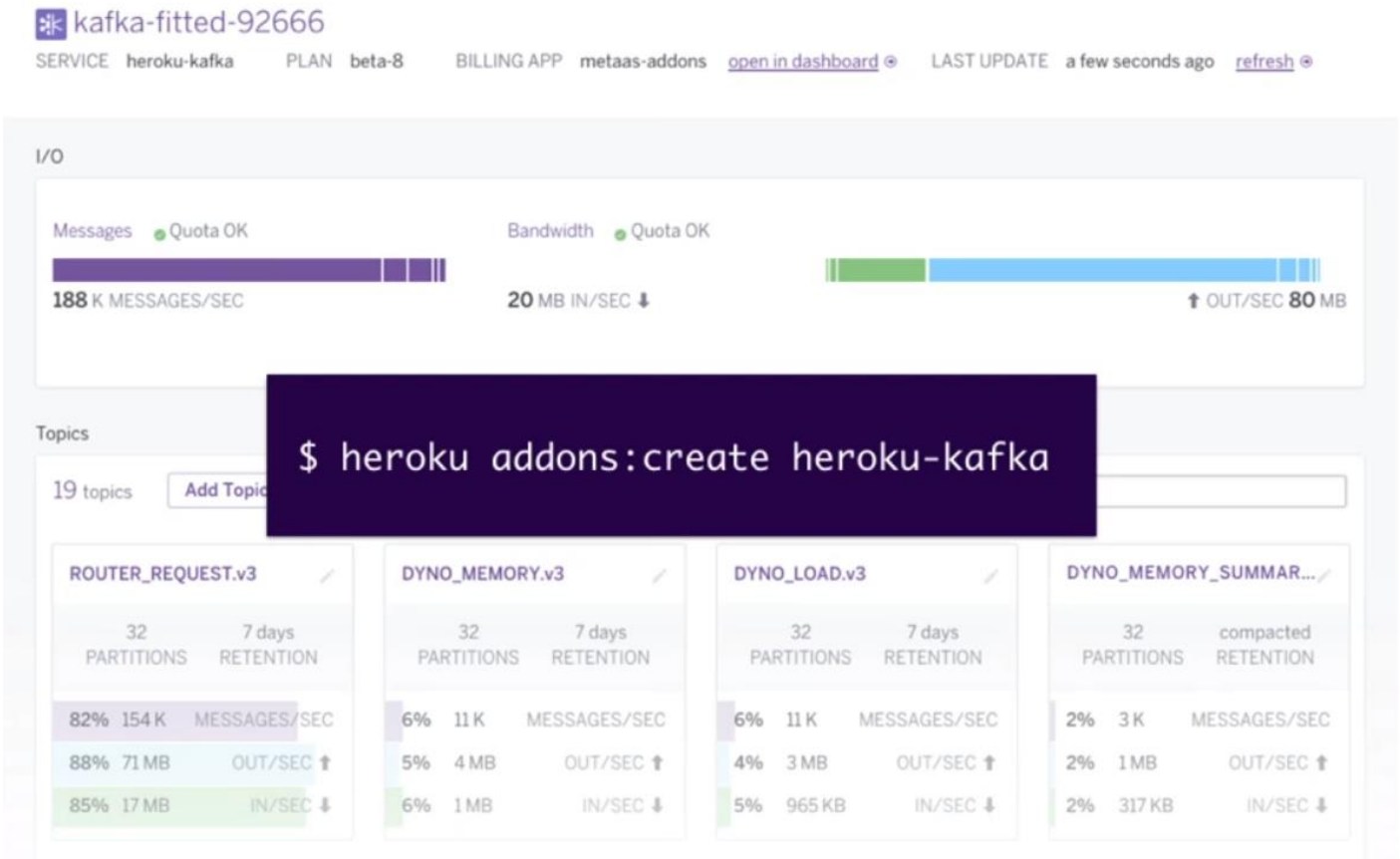
Kafka @ Heroku

- Metrics
- API Event Bus

Heroku Metrics Dashboard

Heroku API Event Bus

Heroku Kafka http://heroku.com/kafka



HEROKU

Joe Kutner
@codefinger