Chapter 7: Serialization

The examples we have come across so far demonstrated fundamental Kafka producer and consumer behaviour by serializing basic types, such as strings. While this may be sufficient to garner an introductory level of awareness, it is of limited use in practice, as real-life applications rarely publish or consume unstructured strings.

Distributed applications communicating in either a message-passing style or as part of an event-driven architecture will utilise a broad catalogue of structured datatypes and corresponding schema contracts. The business logic embedded in producer and consumer applications will typically deal with native domain models, requiring a bridging mechanism to marshal these models to Kafka topics when producing records, and perform the opposite when consuming from a topic.

This chapter covers the broad topic of record serialization. In the course of the discussion, we shall also explore complementary design patterns that streamline the interfacing of an application's business logic with the underlying event stream.

Key and value serializer

Prior examples have revealed that the Kafka `Producer` and `Consumer` API, as well as the `ProducerRecord` and `ConsumerRecord` classes, are generically typed. The `Producer` interface is parametrised with a key and a value type, denoted `K` and `V` in the type parameter list:

```
/**
 * The interface for the {@link KafkaProducer}
 * @see KafkaProducer
 * @see MockProducer
 */
public interface Producer<K, V> extends Closeable {
  /** some methods omitted for brevity */

  /**
   * See {@link KafkaProducer#send(ProducerRecord)}
   */
  Future<RecordMetadata> send(ProducerRecord<K, V> record);

  /**
   * See {@link KafkaProducer#send(ProducerRecord, Callback)}
   */
```

```
  Future<RecordMetadata> send(ProducerRecord<K, V> record,
                              Callback callback);
}
```

The `send()` methods reference the type parameters, requiring the supplied `ProducerRecord` to be of a matching generic type.

Kafka's ingrained type-safety mechanism assumes a pair of compatible serializers for supported key and value types. A custom serializer must conform to the `org.apache.kafka.common.serialization.Serializer` interface, listed below.

```
public interface Serializer<T> extends Closeable {
  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  byte[] serialize(String topic, T data);

  default byte[] serialize(String topic, Headers headers, T data) {
    return serialize(topic, data);
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

Serializers are configured in one of two ways:

1. Passing the fully-qualified class name of a `Serializer` implementation to the producer via the `key.serializer` and the `value.serializer` properties. Note, there are no default values assigned to these properties.
2. Directly instantiating the serializer and passing it as a reference to an overloaded `KafkaProducer` constructor.

The property-based mechanism has the advantage of simplicity, in that it lives alongside the rest of the producer configuration. One can look at the configuration properties and instantly determine that the producer is configured with a specific key and value serializer. The drawback of this configuration style is that it requires the `Serializer` implementation to include a public, no-argument constructor. It also makes it difficult to configure. Because the serializer is instantiated reflectively by the producer client, the application code is unable to inject its own set of arguments at the point of initialisation. The only way to configure a reflectively-instantiated serializer is to supply a set of custom properties to the producer, then retrieve the values of these properties in the `Serializer` implementation, using the optional `configure()` callback:

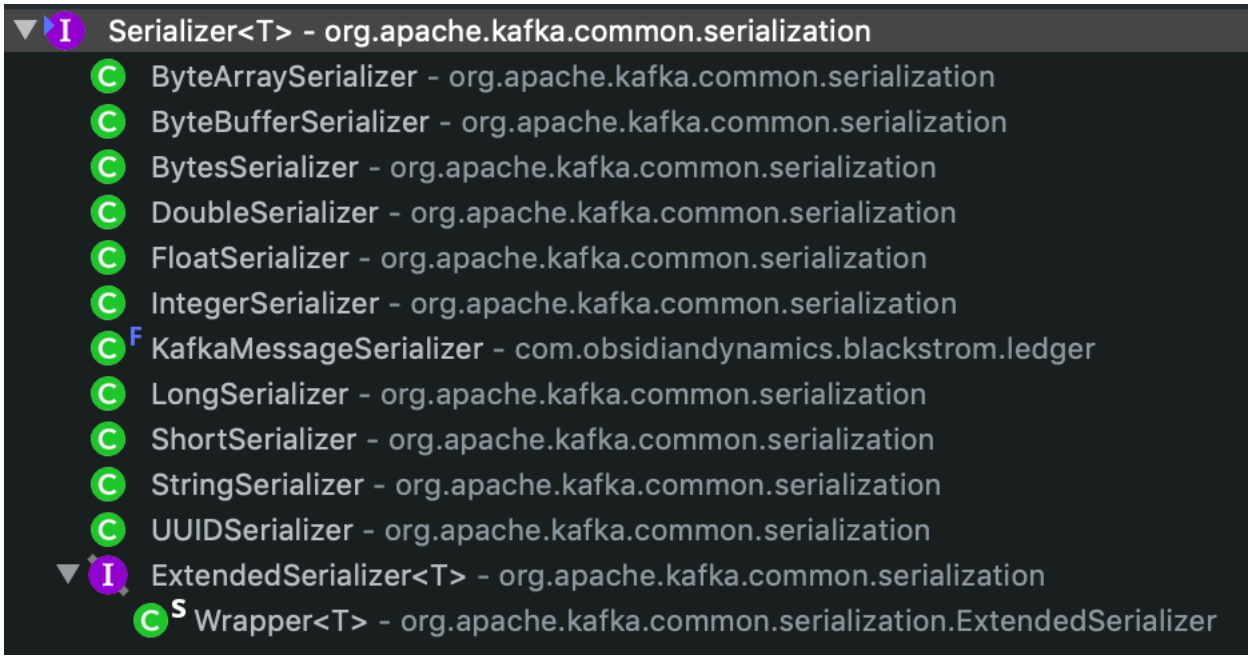
```
/**
 * Configure this class.
 * @param configs configs in key/value pairs
 * @param isKey whether is for key or value
 */
default void configure(Map<String, ?> configs, boolean isKey) {
  // intentionally left blank
}
```

The `configure()` method also helps the instance determine whether it is a key or a value serializer.

Another drawback of the property-based approach is that it ignores the generic type constraints imposed by the `Producer` interface. For example, it is possible to instantiate a `KafkaProducer<Integer, String>` using a `FloatSerializer` for the key and a `ByteArraySerializer` for the value. The problem will remain unnoticed until you try to publish the first record, which will summarily fail with a `ClassCastException`.

Compared to the property-based approach, the passing of a pre-instantiated serializer to a `KafkaProducer` simultaneously solves the problems of maintaining generic type-safety and the configuration of the `Serializer` instance. The application code will instantiate a `Serializer` and configure it appropriately before invoking the `KafkaProducer` constructor. In turn, the constructor's signature will ensure that the given pair of `Serializer` instances conform to the `K` and `V` generic type parameters.

The Kafka client library comes with several pre-canned serializers for common data types, listed below.



Supported serializers

In most applications, record *keys* are simple unstructured values such as integers, strings, or UUIDs, and a built-in serializer will suffice. Record *values* tend to be structured payloads conforming to some pre-agreed schema, represented using a text or binary encoding. Typical examples include JSON, XML, Avro, Thrift, and Protocol Buffers. When serializing a custom payload to a Kafka record, there are generally two approaches one may pursue. These are:

1. Implement a custom serializer to directly handle the payload.
2. Serialize the payload at the application level.

The first approach is idiomatic; unquestionably, it is more fitting to the design of the Kafka API. We would subclass `Serializer`, implementing its `serialize()` method:

```
/**
 * Convert {@code data} into a byte array.
 *
 * @param topic topic associated with data
 * @param data typed data
 * @return serialized bytes
 */
byte[] serialize(String topic, T data);
```

The `serialize()` method accepts the `data` argument that is typed in accordance with the generic type constraint of the `Serializer` interface. From there it is just a matter of marshalling the payload to a byte array.

The alternate method involves piggybacking on an existing serializer that matches the underlying encoding. When dealing with text-based formats, such as JSON or XML, one would use the `StringSerializer`. Conversely, when dealing with binary data, the `ByteArraySerializer` would be selected. This leaves Kafka's `ProducerRecord` and `Producer` instance unaware of the application-level datatype, relying on the application code to pre-serialize the value before constructing a `ProducerRecord`.

A potential advantage of a custom Kafka serializer over application-level serialization is the additional type-safety that the former offers. Looking at it from a different lens, the need for type-safety at the level of a Kafka producer is questionable, as it would likely be encapsulated within a dedicated messaging layer.

This is a good segue into layering. A well-thought-out application will clearly separate business logic from the persistence and messaging concerns. For example, you don't expect to find JDBC `Connection` instances scattered unceremoniously among the business logic classes of a well-designed application. (Nor are JDBC classes type-safe for that matter.) By the same token, it makes sense for the `Producer` class to also be encapsulated in its own layer, ideally using an interface that allows the messaging code to be mocked out independently as part of unit testing. Throughout the chapter,

we will list arguments in favour of encapsulating common messaging concerns within a dedicated layer.

Returning to the question of a custom serializer versus a piggybacked approach, the former is the idiomatic approach, and for this reason we will stick with custom (de)serializers throughout the chapter.

Sending events

For the forthcoming discussion, consider a contrived event streaming scenario involving a basic application for managing customer records.

The complete source code for the upcoming examples is available at github.com/ekoutanov/effectivekafka[8] in the `src/main/java/effectivekafka/customerevents` directory. Most of the relevant code listings are also included here for the reader's convenience; some of the more esoteric items may have been omitted, but they should nonetheless be present in the `effectivekafka` repository on GitHub.

Every change to the customer entity results in the publishing of a corresponding event to a single Kafka topic, keyed by the customer ID. Each event is strongly typed, but there are several event classes and each is bound to a dedicated schema. The POJO representation of these events might be `CreateCustomer`, `UpdateCustomer`, `SuspendCustomer`, and `ReinstateCustomer`. The abstract base class for all customer-related events will be `CustomerPayload`. The base class also houses the common fields, which for the sake of simplicity have been reduced to a single UUID-based unique identifier. This is the ID of the notional customer entity to which the event refers. (For simplicity, the examples will not contain the persistent entities — just the event notifications.)

We are going to assume that records should be serialized using JSON. Along with Avro, JSON is one of the most popular formats for streaming event data over Kafka. The examples in this book use the *FasterXML Jackson* library for working with JSON, which is the *de facto* JSON parser within the Java ecosystem. Subclasses of `CustomerPayload` are specified using a `@JsonSubTypes` annotation, which allows us to use Jackson's built-in support for polymorphic types. Every serialized `CustomerPayload` instance will contain a `type` property, specifying an aliased name of the concrete type, for example, `CREATE_CUSTOMER` for the `CreateCustomer` class. Jackson uses this property as a hint during deserialization, picking the correct subclass of `CustomerPayload` to map the JSON document to.

[8] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/customerevents

```
import java.util.*;

import com.fasterxml.jackson.annotation.*;

@JsonTypeInfo(use=JsonTypeInfo.Id.NAME,
              include=JsonTypeInfo.As.EXISTING_PROPERTY,
              property="type")
@JsonSubTypes({
  @JsonSubTypes.Type(value=CreateCustomer.class,
                     name=CreateCustomer.TYPE),
  @JsonSubTypes.Type(value=UpdateCustomer.class,
                     name=UpdateCustomer.TYPE),
  @JsonSubTypes.Type(value=SuspendCustomer.class,
                     name=SuspendCustomer.TYPE),
  @JsonSubTypes.Type(value=ReinstateCustomer.class,
                     name=ReinstateCustomer.TYPE)
})
public abstract class CustomerPayload {
  @JsonProperty
  private final UUID id;

  CustomerPayload(UUID id) {
    this.id = id;
  }

  public abstract String getType();

  public final UUID getId() {
    return id;
  }

  protected final String baseToString() {
    return "id=" + id;
  }
}
```

Exposing a stable alias rather than the fully-qualified Java class name makes our message schema portable, enabling us to interoperate with non-Java clients. This also aligns with the cornerstone principle of event-driven architecture — the producer has minimal awareness of the downstream consumers, and makes no assumption as to their role and cause, nor their implementation.

For a sampling of a typical concrete event, we have the `CreateCustomer` class. There are a few others,

but they are conceptually similar.

```
public final class CreateCustomer extends CustomerPayload {
  static final String TYPE = "CREATE_CUSTOMER";

  @JsonProperty
  private final String firstName;

  @JsonProperty
  private final String lastName;

  public CreateCustomer(@JsonProperty("id") UUID id,
                        @JsonProperty("firstName") String firstName,
                        @JsonProperty("lastName") String lastName) {
    super(id);
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @Override
  public String getType() {
    return TYPE;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  @Override
  public String toString() {
    return CreateCustomer.class.getSimpleName() + " ["
        + baseToString() + ", firstName=" + firstName
        + ", lastName=" + lastName + "]";
  }
}
```

Ideally, we would like to inject a high-level event sender into the business logic, then have our business logic invoke the sender whenever it needs to produce an event, without concerning itself with how the event is serialized or published to Kafka. This is the perfect case for an interface:

```
import java.io.*;
import java.util.concurrent.*;

import org.apache.kafka.clients.producer.*;

public interface EventSender extends Closeable {
  Future<RecordMetadata> send(CustomerPayload payload);

  final class SendException extends Exception {
    private static final long serialVersionUID = 1L;

    SendException(Throwable cause) { super(cause); }
  }

  default RecordMetadata blockingSend(CustomerPayload payload)
      throws SendException, InterruptedException {
    try {
      return send(payload).get();
    } catch (ExecutionException e) {
      throw new SendException(e.getCause());
    }
  }

  @Override
  public void close();
}
```

The application might want to send records asynchronously — continuing without waiting for an outcome, or synchronously — blocking until the record has been published. We have specified a `Future<RecordMetadata> send(CustomerPayload payload)` method signature for the asynchronous operation — to be implemented by the concrete `EventSender` subclass. The synchronous case is taken care of by the `blockingSend()` method, which simply delegates to `send()`, blocking on the result of the returned `Future`. The sender implementation may choose to override this method with a more suitable one if need be. (Hopefully, the default implementation is good enough.)

Next, we are going to look at a sample user of `EventSender` — the `ProducerBusinessLogic` class.

```
public final class ProducerBusinessLogic {
  private final EventSender sender;

  public ProducerBusinessLogic(EventSender sender) {
    this.sender = sender;
  }

  public void generateRandomEvents()
      throws SendException, InterruptedException {
    final var create =
        new CreateCustomer(UUID.randomUUID(), "Bob", "Brown");
    blockingSend(create);

    if (Math.random() > 0.5) {
      final var update =
          new UpdateCustomer(create.getId(), "Charlie", "Brown");
      blockingSend(update);
    }

    if (Math.random() > 0.5) {
      final var suspend = new SuspendCustomer(create.getId());
      blockingSend(suspend);

      if (Math.random() > 0.5) {
        final var reinstate = new ReinstateCustomer(create.getId());
        blockingSend(reinstate);
      }
    }
  }

  private void blockingSend(CustomerPayload payload)
      throws SendException, InterruptedException {
    System.out.format("Publishing %s%n", payload);
    sender.blockingSend(payload);
  }
}
```

We are not actually going to implement any life-like business logic for this example; the intention is merely to simulate some activity and exercise our future `EventSender` implementation.

The complete sender

Using interfaces is only going to get us so far; we need a concrete implementation of a `EventSender` to make the example work. Here is the simplest sender implementation that will get the job done:

```
import java.util.*;
import java.util.concurrent.*;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

public final class DirectSender implements EventSender {
  private final Producer<String, CustomerPayload> producer;

  private final String topic;

  public DirectSender(Map<String, Object> producerConfig,
                      String topic) {
    this.topic = topic;

    final var mergedConfig = new HashMap<String, Object>();
    mergedConfig.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                     StringSerializer.class.getName());
    mergedConfig.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                     CustomerPayloadSerializer.class.getName());
    mergedConfig.putAll(producerConfig);
    producer = new KafkaProducer<>(mergedConfig);
  }

  @Override
  public Future<RecordMetadata> send(CustomerPayload payload) {
    final var record =
        new ProducerRecord<>(topic,
                             payload.getId().toString(),
                             payload);
    return producer.send(record);
  }

  @Override
  public void close() {
    producer.close();
  }
}
```

There really isn't much to it. The `DirectSender` encapsulates a `KafkaProducer`, configured using a supplied map of properties. The constructor will overwrite certain key properties in the user-specified configuration map — properties that are required for the correct operation of the sender and should not be interfered with by external code. The `DirectSender` also requires the name of the topic.

The `send()` method simply creates a new `ProducerRecord` and enqueues it for sending, delegating to the underlying `Producer` instance. Before this, the `send()` method will also assign the newly created record's key, which, as previously agreed, is expected of the producer application. By setting the key to the customer ID, we ensure that records are strictly ordered by customer.

The benefits of layering become immediately apparent. Without an `EventSender` implementation to guide the construction and sending of records, the responsibility of enforcing invariants would have rested with the business logic layer. This prevents us from enforcing simple invariants that operate at record scope, such as "the key of a record must equal to the ID of the encompassed customer event". Relying on the business logic to set the key correctly is error-prone, especially when you consider that there will be several places where this would be done. By layering our producer application, we can enforce this behaviour deeper in the stack, thereby minimising code duplication and avoiding a whole class of potential bugs.

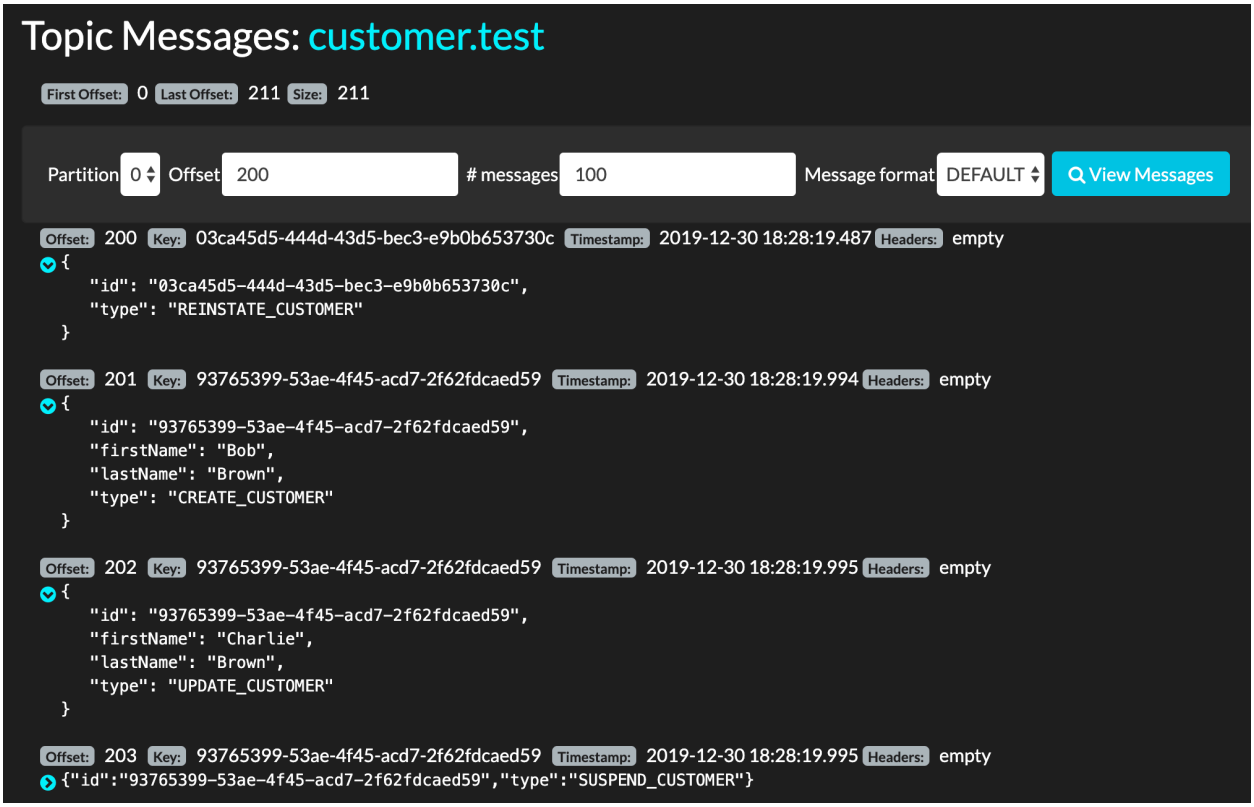
To try out the example, first launch the `RunRandomEventProducer` class. As the name suggests, it will publish a sequence of random customer events using the `ProducerBusinessLogic` defined earlier.

```
public final class RunRandomEventProducer {
  public static void main(String[] args)
      throws InterruptedException, SendException {
    final Map<String, Object> config =
        Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ProducerConfig.CLIENT_ID_CONFIG,
               "customer-producer-sample");

    final var topic = "customer.test";

    try (var sender = new DirectSender(config, topic)) {
      final var businessLogic = new ProducerBusinessLogic(sender);
      while (true) {
        businessLogic.generateRandomEvents();
        Thread.sleep(500);
      }
    }
  }
}
```

Assuming everything goes well, you should see some logs printed to standard out:

```
...
omitted for brevity
...
Publishing {"id":"28361a15-7cef-47f3-9819-f8a629491c5a", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"28361a15-7cef-47f3-9819-f8a629491c5a", □
    "type":"SUSPEND_CUSTOMER"}
Publishing {"id":"28361a15-7cef-47f3-9819-f8a629491c5a", □
    "type":"REINSTATE_CUSTOMER"}
Publishing {"id":"b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8", □
    "firstName":"Charlie","lastName":"Brown","type":"UPDATE_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "firstName":"Charlie","lastName":"Brown","type":"UPDATE_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "type":"SUSPEND_CUSTOMER"}
Publishing {"id":"83bb14e7-9139-4699-a843-8b3a90ae26e2", □
    "type":"REINSTATE_CUSTOMER"}
Publishing {"id":"fe02fe96-a410-44a5-b636-dced53cf4590", □
    "firstName":"Bob","lastName":"Brown","type":"CREATE_CUSTOMER"}
Publishing {"id":"fe02fe96-a410-44a5-b636-dced53cf4590", □
    "firstName":"Charlie","lastName":"Brown","type":"UPDATE_CUSTOMER"}
```

When enough events have been emitted, stop the producer. Switch over to Kafdrop; you should see a series of JSON records appear in the `customer.test` topic:



Kafdrop — random customer events

Key and value deserializer

Analogously to the generic type constraints prevalent in the producer API, the `Consumer` interface enforces an equivalent constraint *vis-à-vis* the `ConsumerRecords` class returned by the `poll()` method, which carries a collection of individual `ConsumerRecord` objects:

```
/**
 * @see KafkaConsumer
 * @see MockConsumer
 */
public interface Consumer<K, V> extends Closeable {
  /** some methods omitted for brevity */

  /**
   * @see KafkaConsumer#poll(Duration)
   */
  ConsumerRecords<K, V> poll(Duration timeout);
}
```

```
/**
 * A container that holds the list {@link ConsumerRecord} per
 * partition for a particular topic. There is one
 * {@link ConsumerRecord} list for every topic partition returned
 * by a {@link Consumer#poll(java.time.Duration)} operation.
 */
public class ConsumerRecords<K, V>
    implements Iterable<ConsumerRecord<K, V>> {
  /** fields and methods omitted for brevity */
}
```

Similarly to the producer scenario, a consumer must be configured with the appropriate key and value deserializers. A deserializer must conform to the `org.apache.kafka.common.serialization.Deserializer` interface, listed below.

```
public interface Deserializer<T> extends Closeable {
  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  T deserialize(String topic, byte[] data);

  default T deserialize(String topic, Headers headers, byte[] data) {
    return deserialize(topic, data);
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

The consumer client allows the user to specify the key and value deserializers in one of two ways:

1. Passing the fully-qualified class name of a `Deserializer` implementation to the consumer via the `key.deserializer` or the `value.deserializer` property.
2. Instantiating the deserializer and passing its reference to an overloaded `KafkaConsumer` constructor.

In virtually every way, the configuration of deserializers on the consumer is consistent with its producer counterpart. Behaviourally, deserializers are the logical reciprocal of serializers.

Akin to the serialization scenario, the user can select one of two strategies for unmarshalling data:

1. Implement a custom deserializer to directly handle the encoded form, such that the application code deals exclusively with typed payloads.
2. Piggyback on an existing deserializer, such as a `StringDeserializer` (for text encodings) or a `ByteArrayDeserializer` (for binary encodings), deferring the final unmarshalling of the encoded payload to the application.

There are no strong merits of one approach over the other that are worthy of a debate. Like in the producer scenario, we will use a custom deserializer to implement the forthcoming examples, being the idiomatic approach.

The section on serializers questioned the merits of type safety at the level of the producer, instead advocating for a façade over the top of the Kafka client code to insulate the business logic from the intricacies of Kafka connectivity, and to easily mock the latter in unit tests. As we will shortly discover, the case for a dedicated insulation layer is further bolstered when dealing with the consumer scenario.

Receiving events

Continuing from the producer example, let's examine the routine concerns of a typical business logic layer that might reside in a consumer application. How would it react to events received from Kafka? And more importantly, how would it even receive these events?

The standard mechanism for interacting with a Kafka consumer is to block on `Consumer.poll()`, then iterate over the returned records — invoking an application-level handler for each record. Kafka's defaults around automatic offset committing have also been designed specifically around this pattern — the *poll-process loop*.

The *poll-process loop* is a coined term, in lieu of an official name put forth by the Kafka documentation or a common name adopted by the user community.

A poll-process loop requires a thread on the consumer, as well as all the life-cycle management code that goes with it — when to start the thread, how to stop it, and so on.

Ideally, we would simply inject a high-level event receiver into the business logic, then register a listener callback with the receiver to be invoked every time the receiver pulls a record from the topic. Perhaps something along these lines:

```
public final class ConsumerBusinessLogic {
  private final EventReceiver receiver;

  public ConsumerBusinessLogic(EventReceiver receiver) {
    this.receiver = receiver;
    receiver.addListener(this::onEvent);
  }

  private void onEvent(CustomerPayload payload) {
    System.out.format("Received %s%n", payload);
  }
}
```

Again, what we do inside the business logic is of little consequence. The purpose of these examples is to illustrate how the business logic layer interacts with Kafka.

The `EventReceiver` and `EventListener` code listings, respectively:

```
public interface EventReceiver extends Closeable {
  void addListener(EventListener listener);

  void start();

  @Override
  void close();
}


@FunctionalInterface
public interface EventListener {
  void onEvent(CustomerPayload payload);
}
```

This approach completely decouples the `ConsumerBusinessLogic` class from the consumer code, being aware only of `EventReceiver`, which in itself is merely an interface. All communications with Kafka will be proxied via a suitable `EventReceiver` implementation.

Corrupt records

A producer has the benefit of knowing that the records given to it by the application code are valid, at least as far as the application is concerned. A consumer reading from an event stream does not have this luxury. A rogue or defective producer may have published garbage onto the topic, which would be summarily fed to all downstream consumers.

Ideally, we should handle any potential deserialization issues gracefully. As deserialization is within our control, we have several choices around the error-handling behaviour:

- Just log the error and discard the record;
- Propagate the error to the application via the (modified) callback, along with the malformed record; or
- Publish the malformed record to a dedicated 'dead letter' topic for subsequent inspection.

Assuming the decision is to pass the error to the application, the modified code might resemble the following:

```
public final class ConsumerBusinessLogic {
  public ConsumerBusinessLogic(EventReceiver receiver) {
    receiver.addListener(this::onEvent);
  }

  private void onEvent(ReceiveEvent event) {
    if (! event.isError()) {
      System.out.format("Received %s%n", event.getPayload());
    } else {
      System.err.format("Error in record %s: %s%n",
                        event.getRecord(), event.getError());
    }
  }
}


@FunctionalInterface
public interface EventListener {
  void onEvent(ReceiveEvent event);
}
```

The new `ReceiveEvent` class encapsulates both the `CustomerPayload` object — if one was unmarshalled successfully, or a `Throwable` error — if an exception occurred during unmarshalling. In both cases, the original `ConsumerRecord` is also included for reference, as well as the original encoded value. The source listing of `ReceiveEvent` follows.

```
public final class ReceiveEvent {
  private final CustomerPayload payload;

  private final Throwable error;

  private final ConsumerRecord<String, ?> record;

  private final String encodedValue;

  public ReceiveEvent(CustomerPayload payload,
                      Throwable error,
                      ConsumerRecord<String, ?> record,
                      String encodedValue) {
    this.record = record;
    this.payload = payload;
    this.error = error;
    this.encodedValue = encodedValue;
  }

  public CustomerPayload getPayload() {
    return payload;
  }

  public boolean isError() {
    return error != null;
  }

  public Throwable getError() {
    return error;
  }

  public ConsumerRecord<String, ?> getRecord() {
    return record;
  }

  public String getEncodedValue() {
    return encodedValue;
  }

  @Override
  public String toString() {
    return ReceiveEvent.class.getSimpleName() + " [payload="
        + payload + ", error=" + error + ", record=" + record
```

```
        + ", encodedValue=" + encodedValue + "]";
  }
}
```

The complete receiver

Now, to complete the implementation, we require a functioning `EventReceiver`. The listing below is that of the `DirectReceiver`, which is an implementation of the poll-process loop.

The choice of the term 'direct' is for consistency with the producer example. In both cases, the implementations directly employ the underlying Kafka API, without deviating from the standard behaviour or acquiring any additional characteristics — hence the name.

```
import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.errors.*;
import org.apache.kafka.common.serialization.*;

import com.obsidiandynamics.worker.*;

public final class DirectReceiver extends AbstractReceiver {
  private final WorkerThread pollingThread;

  private final Consumer<String, CustomerPayloadOrError> consumer;

  private final Duration pollTimeout;

  public DirectReceiver(Map<String, Object> consumerConfig,
                        String topic,
                        Duration pollTimeout) {
    this.pollTimeout = pollTimeout;

    final var mergedConfig = new HashMap<String, Object>();
    mergedConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                     StringDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                     CustomerPayloadDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
                     false);
```

```
    mergedConfig.putAll(consumerConfig);
    consumer = new KafkaConsumer<>(mergedConfig);
    consumer.subscribe(Set.of(topic));

    pollingThread = WorkerThread.builder()
        .withOptions(new WorkerOptions()
                     .daemon()
                     .withName(DirectReceiver.class, "poller"))
        .onCycle(this::onPollCycle)
        .build();
  }

  @Override
  public void start() {
    pollingThread.start();
  }

  private void onPollCycle(WorkerThread t)
      throws InterruptedException {
    final ConsumerRecords<String, CustomerPayloadOrError> records;

    try {
      records = consumer.poll(pollTimeout);
    } catch (InterruptException e) {
      throw new InterruptedException("Interrupted during poll");
    }

    if (! records.isEmpty()) {
      for (var record : records) {
        final var payloadOrError = record.value();
        final var event =
            new ReceiveEvent(payloadOrError.getPayload(),
                             payloadOrError.getError(),
                             record,
                             payloadOrError.getEncodedValue());
        fire(event);
      }
      consumer.commitAsync();
    }
  }

  @Override
  public void close() {
```

```
    pollingThread.terminate().joinSilently();
    consumer.close();
  }
}
```

The `DirectReceiver` maintains a single polling thread. Rather than incorporating threading from first principles, the examples in this book use the *Fulcrum* micro-library, available at github.com/obsidiandynamics/f Specifically, the examples import the `fulcrum-worker` module, which provides complete life-cycle management on top of a conventional `java.lang.Thread`. A Fulcrum `WorkerThread` class provides an opinionated set of controls and templates that standardise all key aspects of a thread's behaviour — the initial startup, steady-state operation, interrupt-triggered termination, and exception handling. These are typical concerns in multi-threaded applications, which are difficult to get right and require a copious amount of non-trivial boilerplate code to adequately cover all the edge cases.

Our receiver takes three parameters — a map of configuration properties for the Kafka consumer, the name of the topic to subscribe to, and a timeout value to use in `Consumer.poll()`. Like its `DirectSender` counterpart, the `DirectReceiver` will overwrite certain key properties in the user-specified configuration map — settings that are required for the correct operation of the receiver and should not be interfered with by external code.

The `onPollCycle()` method represents a single iteration of the poll-process loop. Its role is straightforward — fetch records from Kafka, construct a corresponding `ReceiveEvent`, and dispatch the event to all registered listeners. Once all records in the batch have been dispatched, the `commitAsync()` method of the consumer is invoked, which will have the effect of asynchronously committing the offsets for all records fetched in the last call to `poll()`. Being asynchronous, the client will dispatch the request in a background thread, not waiting for the commit response from the brokers; the responses will arrive at an indeterminate time in the future, after `commitAsync()` returns.

The use of `commitAsync()` makes it possible to process multiple batches before the effects of committing the first are reflected on the brokers, increasing the window of uncommitted records. And while this will lead to a greater number of replayed records following partition reassignment, this behaviour is still consistent with the concept of *at-least-once* delivery. Using the blocking `commitSync()` variant reduces the number of uncommitted records to the in-flight batch at the expense of throughput. Unless the cost of processing a record is very high, the asynchronous commit model is generally preferred.

Finally, the `close()` method disposes of the receiver by terminating the polling thread, awaiting its termination, then closing the Kafka consumer.

Notice how we have caught an odd-looking `org.apache.kafka.common.errors.InterruptException` in the body of the `onPollCycle()` method, re-throwing a `java.lang.InterruptedException` in its place. This is one of the idiosyncrasies of the Kafka API — its origins are traceable to Scala,

[9] https://github.com/obsidiandynamics/fulcrum

which does not support checked exceptions, arguing vigorously against their use. As a result, the unchecked-exceptions-only philosophy has been carried over to the Java port, going against the grain of idiomatic Java.

```
try {
  records = consumer.poll(pollTimeout);
} catch (InterruptException e) {
  throw new InterruptedException("Interrupted during poll");
}
```

The standard Java thread interrupt signalling has been unceremoniously discarded in the bowels of the `KafkaConsumer` client and replaced with a bespoke runtime exception type. The code above corrects for this, trapping the bespoke exception and re-throwing a standard one. When this occurs, the Fulcrum `WorkerThread` will detect the interrupt and gracefully shut down the underlying primordial thread.

To run the example, launch the `RunDirectConsumer` class:

```
public final class RunDirectConsumer {
  public static void main(String[] args)
      throws InterruptedException {
    final Map<String, Object> consumerConfig =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.GROUP_ID_CONFIG,
               "customer-direct-consumer",
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest");

    try (var receiver = new DirectReceiver(consumerConfig,
                                           "customer.test",
                                           Duration.ofMillis(100))) {
      new ConsumerBusinessLogic(receiver);
      receiver.start();
      Thread.sleep(10_000);
    }
  }
}
```

This will run for ten seconds, outputting the records that have been published since the consumer was last run. Since we are running it for the first time, expect to see all records in the `customer.test` topic:

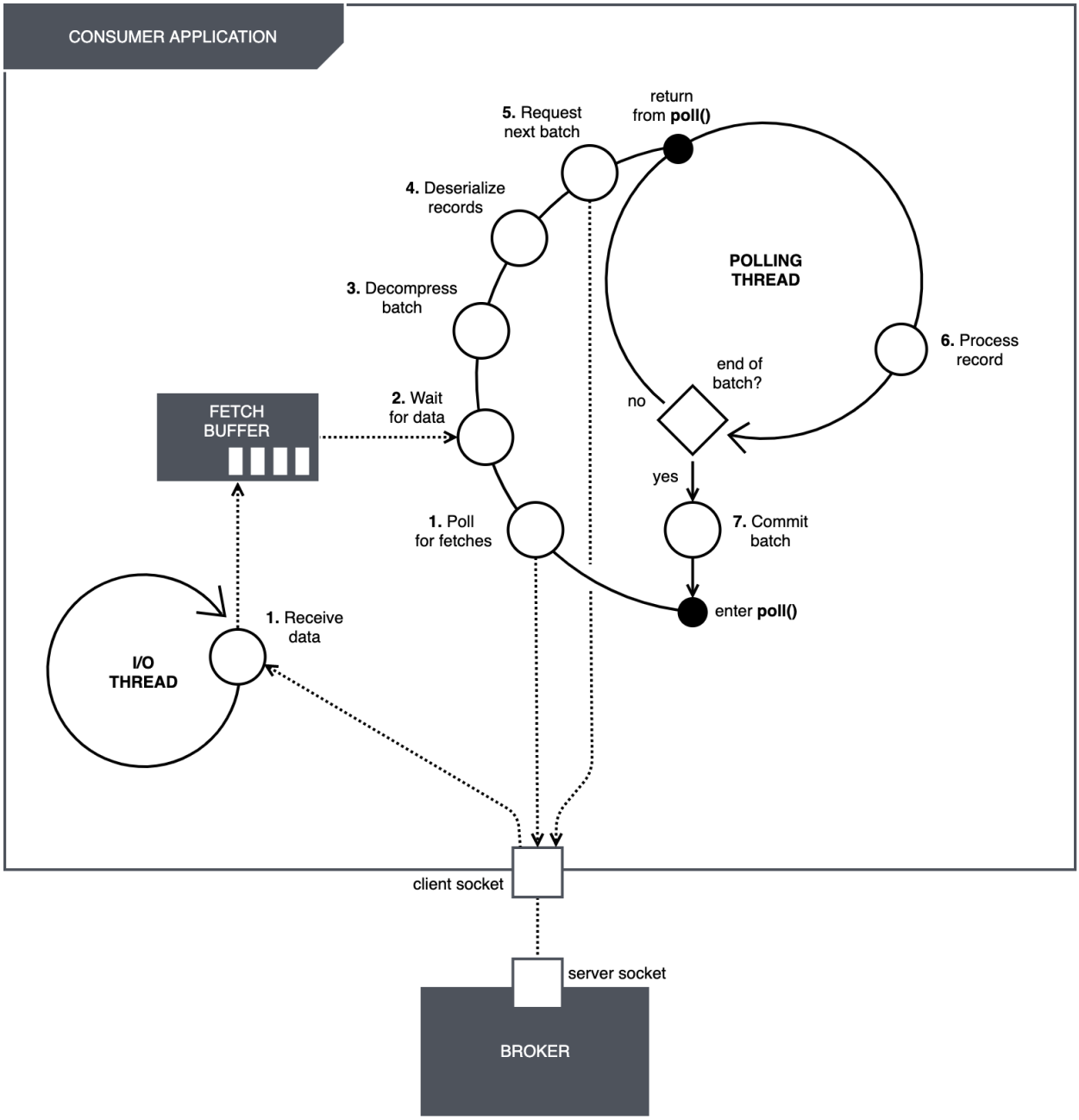
```
...
omitted for brevity
...
Received CreateCustomer [id=28361a15-7cef-47f3-9819-f8a629491c5a, ▯
    firstName=Bob, lastName=Brown]
Received SuspendCustomer [id=28361a15-7cef-47f3-9819-f8a629491c5a]
Received ReinstateCustomer [id=28361a15-7cef-47f3-9819-f8a629491c5a]
Received CreateCustomer [id=b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8, ▯
    firstName=Bob, lastName=Brown]
Received UpdateCustomer [id=b3cd538c-90cc-4f5f-a5dc-b1c469fc0bf8, ▯
    firstName=Charlie, lastName=Brown]
Received CreateCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2, ▯
    firstName=Bob, lastName=Brown]
Received UpdateCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2, ▯
    firstName=Charlie, lastName=Brown]
Received SuspendCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2]
Received ReinstateCustomer [id=83bb14e7-9139-4699-a843-8b3a90ae26e2]
Received CreateCustomer [id=fe02fe96-a410-44a5-b636-dced53cf4590, ▯
    firstName=Bob, lastName=Brown]
Received UpdateCustomer [id=fe02fe96-a410-44a5-b636-dced53cf4590, ▯
    firstName=Charlie, lastName=Brown]
```

Pipelining

One material argument for layering a consumer application stems from the realm of performance optimisation, namely a technique called ‘pipelining’.

A *pipeline* is a decomposition of a sequential process into a set of chained stages, where the output of one stage is fed as the input to the next via an intermediate bounded buffer. Each stage functions semi-independently of its neighbours; it can operate for as long as at least one element is available in its input buffer and will also halt for as long as the output buffer is full.

Pipelining allows the application to recruit additional threads — increasing the throughput at the expense of processor utilisation. To appreciate where the performance gains may be obtained, consider the routine operation of a regular consumer application — identical to the one we just implemented — illustrated below.



Consumer without additional pipelining

The `KafkaConsumer` implementation utilises a rudimentary form of pipelining under the hood, prefetching and buffering records to accelerate content delivery. In other words, our application is already multi-threaded with us hardly realising this — separating the record retrieval and processing operations into distinct execution contexts. The main *poller* thread will —

1. Invoke `KafkaConsumer.poll()`, potentially sending fetch queries to the cluster. If there are pending queries for which responses have not yet been received, no further queries are issued.

2. Wait for the outcome of a pending fetch, checking the status of the fetch buffer. The accumulation of the fetch results will be performed by a background I/O thread. This operation will block until the data becomes available or the poll timeout expires.
3. Decompress the batch if compression was set on the producer.
4. Deserialize each record in the batch.
5. Prior to returning from `poll()`, initiate a prefetch. This action is non-blocking; it fires off a set of queries to the brokers and returns immediately, without waiting for responses. The control is transferred back to the application code. When the prefetch responses eventually arrive, these will be decompressed and deserialized, with the resulting records placed into a fetch buffer.
6. The application then applies the requisite business logic to each record by invoking the registered `EventListener` callbacks. In most applications, this would involve updating a database and possibly other I/O. More often than not, the cost of processing a record is significantly greater than the cost of reading it off a Kafka topic.
7. After the application has completed processing the batch, it can safely commit the consumer's offsets by invoking `Consumer.commitAsync()`. This will have the effect of committing the offsets for all records returned during the last `poll()`. Being an asynchronous operation, the committing of offsets will occur in a background I/O thread.

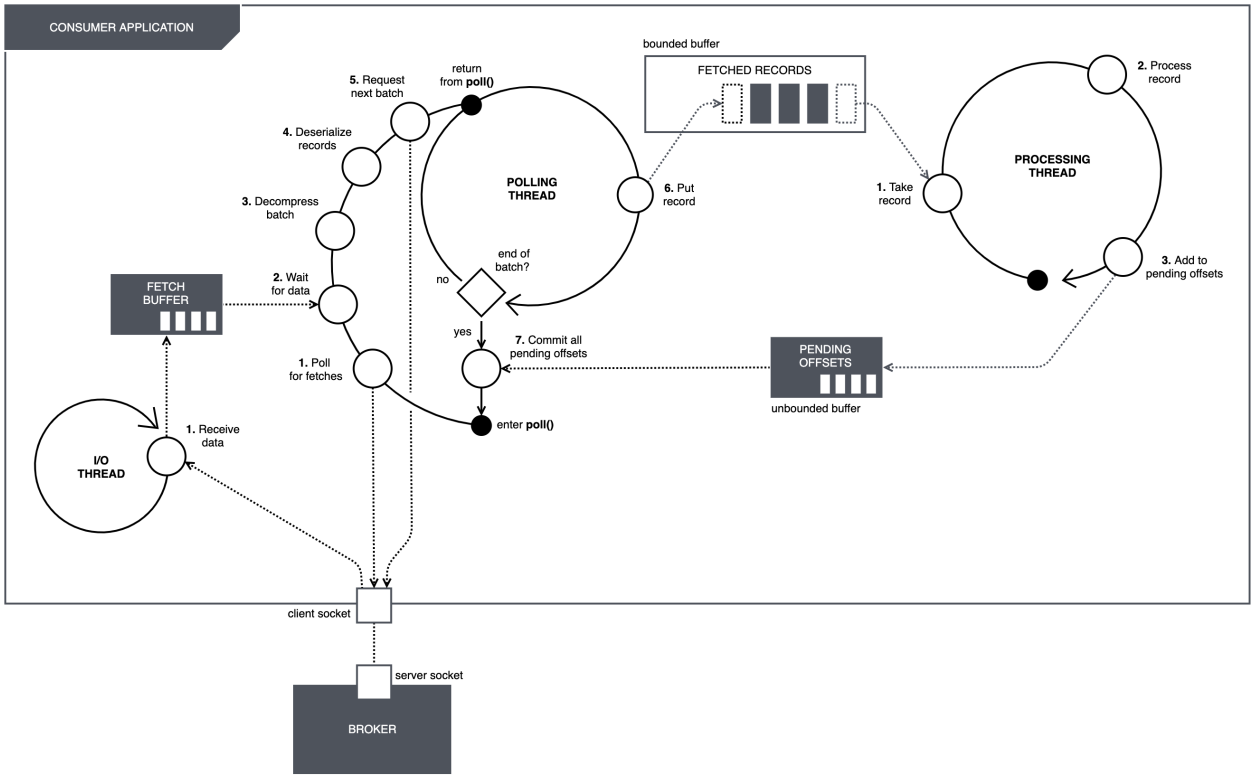
The last step in the poll-process loop is optional, in that we could have just deferred to the consumer's built-in 'automatic offset committing' feature by leaving `enable.auto.commit` at its default value of `true`. In the `DirectReceiver` example, we have chosen to disable offset auto-commit, going with the manual option instead. The principal advantage of committing offsets manually is that it results in a narrower window of uncommitted offsets compared to the automatic option, even if done asynchronously, as the auto-commit is bounded by a timer and lazily initiated. Conversely, the benefit of the offset auto-commit feature is the reduction in the number of commit requests sent to the brokers, which has a small positive effect on throughput.

The I/O thread works in the background. Among its chief responsibilities is the handling of responses from earlier fetch requests, accumulating the received batches in a fetch buffer. With the assistance of the background prefetch mechanism, and assuming a steady flow of records through the pipeline, most calls to `poll()` should not block.

While the `KafkaConsumer` allows for pipelining via its prefetch mechanism, the implementation stacks the deserialization of records and their subsequent handling onto a single thread of execution. The thread that is responsible for deserializing the records is also used to drive business logic. Both operations are potentially time-consuming; when a record is being deserialized, the polling thread is unable to execute the `EventListener` callbacks, and vice versa.

While we cannot control this element of the client's standard behaviour, we can make greater use of the pipeline pattern, harnessing additional performance gains by separating record deserialization from payload handling.

The diagram below illustrates this.



Consumer with pipelining

The fetching, deserialization, and processing of records has now been separated into three stages, each powered by a dedicated thread. For simplicity, we are going to refer to these as the *I/O thread*, the *polling thread*, and the *processing thread*. The I/O thread is native to the `KafkaConsumer` and its behaviour is unchanged from the previous example.

The polling thread is altered in two crucial ways:

1. Rather than invoking the `EventListener` in step 6, the thread will append the received record onto a bounded buffer. In a Java application, we can use an `ArrayBlockingQueue` or a `LinkedBlockingQueue` to implement this buffer. This operation will block if the queue is at its maximum capacity.
2. Instead of committing the offsets of the recent batch, the polling thread will commit just those offsets that have been appended to the 'pending offsets queue' by the processing thread.

On the processing thread, we have the following steps:

1. Remove the queued record from the bounded buffer. This operation will block if the queue is empty.
2. Invoke the registered `EventListener` callbacks to process the record.
3. Having processed the record, append a corresponding entry to the 'pending offsets queue'. This entry specifies the topic-partition pair for the record, as well as its offset *plus one*.

The last steps in each of the two threads may appear confusing at first. Couldn't we just commit the offsets after enqueuing the batch? What is the purpose of shuttling the offsets back from the processing thread to the polling thread? And why would we add one to the offset of a processed record?

When pipelining records, one needs to take particular care when committing the records' offsets, as the records might not be processed for some time after being queued. Depending on the capacity of the bounded buffer, the polling loop may complete several cycles before the processing thread gets an opportunity to attend to the first queued record. The failure of the consumer application would lead to missed records; the newly assigned consumer will have naturally assumed that the committed records were processed. To achieve at-least-once delivery semantics, the offsets of a record must be committed at some point after the record is processed.

While disabling `enable.auto.commit` is optional in the direct consumer scenario, it must categorically be disabled in the pipeline scenario. The effect of leaving offset auto-commit on is the logical equivalent of committing records after queuing them, with no regard as to whether they were processed by the downstream stage.

The need to shuttle the offsets back to the I/O thread addresses an inherent limitation of the `KafkaConsumer` implementation. Namely, *the consumer is not thread-safe*. Attempting to invoke `commitAsync()` from a thread that is different to the one that invoked `poll()` will result in a `java.util.ConcurrentModificationException` exception. As such, we have no choice but to repatriate the offsets to the polling thread.

The final point — the addition of one to a record's offset — accounts for the fact that a Kafka consumer will start processing records from the exact offset persisted against its encompassing consumer group. Naively committing the record's offset 'as is' will result in the replaying of the last committed record following a topic rebalancing event, where partitions may be reassigned among the consumer population. By adding one to the offset, we are ensuring that the new assignee will skip over the last processed record — seamlessly taking over from where the last consumer left off.

Invoking the no-argument `commitAsync()` method, as in the `DirectReceiver` scenario, will automatically add one to the offsets of the last records processed for each partition. When specifying offsets explicitly, the offset arithmetic becomes the responsibility of the application.

For our next trick, we shall conjure up an alternate receiver implementation, this time exploiting the pipeline pattern outside of the consumer:

```
import java.time.*;
import java.util.*;
import java.util.concurrent.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.*;
import org.apache.kafka.common.errors.*;
import org.apache.kafka.common.serialization.*;

import com.obsidiandynamics.worker.*;
import com.obsidiandynamics.worker.Terminator;

public final class PipelinedReceiver extends AbstractReceiver {
  private final WorkerThread pollingThread;

  private final WorkerThread processingThread;

  private final Consumer<String, CustomerPayloadOrError> consumer;

  private final Duration pollTimeout;

  private final BlockingQueue<ReceiveEvent> receivedEvents;

  private final Queue<Map<TopicPartition, OffsetAndMetadata>>
      pendingOffsets = new LinkedBlockingQueue<>();

  public PipelinedReceiver(Map<String, Object> consumerConfig,
                           String topic,
                           Duration pollTimeout,
                           int queueCapacity) {
    this.pollTimeout = pollTimeout;
    receivedEvents = new LinkedBlockingQueue<>(queueCapacity);

    final var mergedConfig = new HashMap<String, Object>();
    mergedConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                     StringDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                     CustomerPayloadDeserializer.class.getName());
    mergedConfig.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
                     false);
    mergedConfig.putAll(consumerConfig);
    consumer = new KafkaConsumer<>(mergedConfig);
    consumer.subscribe(Set.of(topic));
```

```
    pollingThread = WorkerThread.builder()
        .withOptions(new WorkerOptions()
                     .daemon()
                     .withName(PipelinedReceiver.class, "poller"))
        .onCycle(this::onPollCycle)
        .build();

    processingThread = WorkerThread.builder()
        .withOptions(new WorkerOptions()
                     .daemon()
                     .withName(PipelinedReceiver.class, "processor"))
        .onCycle(this::onProcessCycle)
        .build();
  }

  @Override
  public void start() {
    pollingThread.start();
    processingThread.start();
  }

  private void onPollCycle(WorkerThread t)
      throws InterruptedException {
    final ConsumerRecords<String, CustomerPayloadOrError> records;

    try {
      records = consumer.poll(pollTimeout);
    } catch (InterruptException e) {
      throw new InterruptedException("Interrupted during poll");
    }

    if (! records.isEmpty()) {
      for (var record : records) {
        final var value = record.value();
        final var event = new ReceiveEvent(value.getPayload(),
                                           value.getError(),
                                           record,
                                           value.getEncodedValue());
        receivedEvents.put(event);
      }
    }
```

```
    for (Map<TopicPartition, OffsetAndMetadata> pendingOffset;
        (pendingOffset = pendingOffsets.poll()) != null;) {
      consumer.commitAsync(pendingOffset, null);
    }
  }

  private void onProcessCycle(WorkerThread t)
      throws InterruptedException {
    final var event = receivedEvents.take();
    fire(event);
    final var record = event.getRecord();
    pendingOffsets
    .add(Map.of(new TopicPartition(record.topic(),
                                   record.partition()),
                new OffsetAndMetadata(record.offset() + 1)));
  }

  @Override
  public void close() {
    Terminator.of(pollingThread, processingThread)
    .terminate()
    .joinSilently();
    consumer.close();
  }
}
```

There are a few notable differences between a `PipelinedReceiver` and its `DirectReceiver` counterpart:

1. The addition of a second thread — we now have a distinct `processingThread` and a `pollingThread`, whereas the original implementation made do with a single `pollingThread`. Correspondingly, the pipelined implementation has two `onCycle` handlers.
2. The addition of a `LinkedBlockingQueue`, acting as a bounded buffer between the two worker threads.
3. An additional parameter to the constructor, specifying the capacity of the blocking queue.

The `onPollCycle()` method fetches records, but does not dispatch the event. Instead, it puts the event onto the `receivedEvents` queue, blocking if necessary until space becomes available. Having dealt with the batch, it will gather any pending offsets that require committing, taking care not to block while consuming from the `pendingOffsets` queue.

Using the non-blocking `Queue.poll()` method prevents a deadlock condition, where the polling thread is blocked on the processor thread to submit additional offsets, while the processor thread is hopelessly waiting on the polling thread to convey records through the pipeline.

The `onProcessCycle()` method takes records from the head of the `receivedEvents` queue, waiting if necessary for an event to become available. The event is then dispatched to all registered listeners. Finally, the offsets of the underlying record are incremented and submitted to the `pendingOffsets` queue for subsequent committing by the polling thread.

To run the pipelined example, launch the `RunPipelinedConsumer` class:

```
public final class RunPipelinedConsumer {
  public static void main(String[] args)
      throws InterruptedException {
    final Map<String, Object> consumerConfig =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.GROUP_ID_CONFIG,
               "customer-pipelined-consumer",
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest");

    try (var receiver =
             new PipelinedReceiver(consumerConfig,
                                   "customer.test",
                                   Duration.ofMillis(100), 10)) {
      new ConsumerBusinessLogic(receiver);
      receiver.start();
      Thread.sleep(10_000);
    }
  }
}
```

By applying the pipeline pattern, we have decoupled two potentially slow operations, allowing them to operate independently of one another. The performance gains are not exclusive to multi-core or multi-processor architectures; even single-core, pseudo-concurrent systems will benefit from pipelining by maximising the amount of useful work a processor can do.

One of the perceived drawbacks of pipelining is that it adds latency to the process; the contention over a shared buffer and the overhead of thread scheduling and cache coherence will add to the end-to-end propagation delay of a record flowing through the pipeline. And while the added latency is typically more than made up for in throughput gains, it is really up to the application designer to

make the final call on the optimisation strategy. An argument could be made that since *Kafka-based applications tend to be throughput-oriented*, optimisations of this nature are timely and appropriate.

Kafka's performance doctrine has traditionally been to add more consumers, more partitions, and more brokers. Often, little regard is given to the cost-effectiveness of this model. It is not uncommon to observe a high number of mostly-idling consumers spawned in an autoscaling group to get through a moderately loaded topic. One cannot help but question the actual number of consumer instances required, had each instance utilised its available resources to their provisioned capacity.

And now for the best part. Previous examples have leveraged the `EventReceiver` interface to decouple the business logic from the low-level Kafka `Consumer` handling. As we have just demonstrated, pipelining can fit entirely into the `EventReceiver` implementation, with *no impact to the business logic layer*. The complexities of multi-threaded code, queuing of records, and the shuttling of offsets are entirely concealed behind the `EventReceiver` interface. In fact, we can stock multiple `EventReceiver` implementations — a more conventional `DirectReceiver` and a multi-threaded `PipelinedReceiver`. These are functionally equivalent, but exhibit different performance and resource utilisation characteristics. The `PipelinedReceiver` adds 30 or so lines on top of the `DirectReceiver` implementation. Comparatively speaking, this may seem like a lot, given that the extra code is around 50% of the original implementation. But in the larger scheme of things, 30 lines are small potatoes and the added complexity is incurred *once* — the improved resource utilisation profile and the resulting performance gains more than make up for the additional effort.

In case the reader is wondering, the pipelining optimisation is less effective in producer applications, as serialization is typically much less processor and memory-intensive compared to deserialization. Also, the `KafkaProducer` is pipelined internally, separating serialization from network I/O. Little in the way of performance gains would be accomplished by moving record serialization to a dedicated thread.

Record filtering

In rounding off this chapter, we shall highlight another compelling reason for an abstraction layer: the filtering of records. Filtering fulfills a set of use cases where either a deserializer, or an application-level unmarshaller might conditionally present a record to the rest of the application. This is not a native capability of a Kafka consumer, requiring a bespoke implementation.

The natural question one might ask is: Why not filter at the business logic layer with some `if` statements?

There are two challenges with this approach, which also become more difficult to solve as one moves up the application stack. Firstly, it assumes that the client has the requisite domain objects that can be mapped from a record's serialized form. Secondly, it incurs the performance overhead of unconditionally unmarshalling all records, only to discard some records shortly thereafter.

The first problem — missing domain objects or the lack of knowledge of certain record schemas (the two are logically equivalent) — can be attributed to several causes:

- The source Kafka topic is broadly-versed, containing more types of records that the consumer legitimately requires for routine operation.
- The record types might be known to the consumer, but it may have no interest in processing them. For example, a topic might represent changes to a global customer database, whereas a consumer deployed in a single region might only care about the subset of customers in its locality.
- The record structure has evolved over time, such that the topic may contain records that comply to varying schema versions. To accommodate the gradual transition of consumers to a newer schema, producers will typically publish the same record in multiple versions.

At any rate, the consumer will have to selectively parse the record's value on the basis of some explicit indicator. This indicator may be a well-known header — for example, `recordType: CREATE_CUSTOMER`, `region: Asia-Pacific`, or `version: 2`. Alternatively, the indicator may be inferred from the record's value without having to parse the entire payload, let alone mapping the payload to a POJO. For example, the Jackson library allows you to create a custom deserializer that can inspect the document object model before deciding to map it to an existing Java class.

In some cases, filtering records from within the business logic is a valid approach, particularly where the filtering predicates require deep inspection of the record's payload or are related to the current application state.

The final point relates to performance. While premature optimisation should be avoided, there may be legitimate cases where the sheer amount of data on a topic places a strain on the consumer ecosystem, particularly when the topic is coarse-grained. If the options for increasing the granularity of topics and the scaling of consumers have been exhausted, the sole remaining option might be to pre-filter records in an attempt to extract the last ounce of performance. Sounds terrible? Agreed! The recommendation is to avoid complexity on the basis of performance alone. Instead, the application should be architected from the outset to cope with the expected load.

Kafka's idiomatic approach for dealing with varying record representations is through custom (de)serializers. Once implemented and configured, (de)serializers work behind the scenes, accepting and delivering application-native record keys and values via a generically typed API. The producer application is expected to address the `Producer` implementation directly, while on the consumer-end, this approach is often paired with a simple poll-process loop.

This chapter has explored some of the typical concerns of producer and consumer applications, arguing for the use of an abstraction layer to separate Kafka-specific messaging code from the

business logic. This makes it easier to encapsulate common behaviour and invariants on one hand, and on the other, simplifies key aspects of the application, making it easier to mock and test in isolation.

We have also come to understand the inefficiency inherent in the poll-process loop, namely the stacking of record deserialization and processing onto a single execution thread. The internal pipelining model of a `KafkaConsumer` was explained, and we explored how the concept of pipelining can be exploited to further decouple the deserialization of records from their subsequent processing.

Finally, we have looked at record versioning and filtering as prime use cases for concealing non-trivial behaviour behind an abstraction layer, reducing the amount of work that needs to happen at the processing layer and its resulting complexity.