Chapter 11: Robust Configuration

Chapter 10: Client Configuration covered all aspects of client configuration in detail. Among other points, it was mentioned that when a client is instantiated, it verifies that the supplied configuration is valid. In other words, it checks that the given keys correspond to valid configuration property names that are supported in the context of that client type. Failing to meet this requirement will result in a warning message being emitted via the configured logger, but the client will still launch.

Kafka's cavalier approach to configuration raises the following questions: How does one make sure that the supplied configuration is valid *before* launching the client application? Or must we wait for the application to be deployed before being told that we mucked something up?

Using constants

The most common source of misconfiguration is a simple typo. Depending on the nature of the misspelled configuration entry, there may be a substantial price to pay for getting it wrong. The example presented in Chapter 10: Client Configuration, involving `max.in.flight.requests.per.connection`, suggests that a mistake may incur the loss of record order under certain circumstances. Relying solely on inspecting log files does not inspire a great deal of confidence — the stakes are too high. There must be a way of nailing the property names; knowing up-front what you give the client will actually be used; and knowing early, before any harm is done.

Kafka does not yet have a satisfactory answer to this, nor is there a KIP in the pipeline that aims to solve this. As a concession, Kafka can meet you halfway with constants:

```
Map<String, Object> config =
    Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
           "localhost:9092",
           ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName(),
           ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName(),
           ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
           1);
```

In the listing above, property names were replaced with static constants. These constants are stable and will not be changed between release versions. So you can be certain that property names have not been misspelled *if they were embedded in code*. The last point is crucial, as properties may be loaded from an external source, as it is often the case. Constants will not guard against this.

Constants will also fail to protect the user in those scenarios where one might accidentally add a constant from `ConsumerConfig` into a producer configuration, or *vice versa*. Admittedly, this is less likely than misspelling a key, but it is still something to be wary of.

Type-safe configuration

When bootstrapping Kafka client configuration from code or loading configuration artifacts from an external source, the recommended approach is to craft dedicated Java classes with strongly typed values, representing the complete set of configuration items that one can reasonably expect to be supplied when their application is run. The external configuration documents can then be mapped to an object form using a JSON or YAML parser. It is also possible to use a plain `.properties` file if the configuration structure is flat. Alternatively, if using an application framework such as Spring Boot or Micronaut, use the built-in configuration mechanism which supports all three formats.

If it is necessary to support free-form configuration *in addition* to some number of expected items, add a `Map<String, Object>` attribute to the configuration class. When staging the Kafka configuration, apply the free-form properties first, then apply the expected ones. When applying the expected properties, check if the staging configuration already has a value set — if it does, throw a runtime exception. While this doesn't protect against misspelt property names in the free-form section, it will at least ensure that the expected configuration takes precedence.

For those cases when there is an absolute requirement for the property names to be correct before initialisation, one can take their validation a step further: First, the `java.lang.reflect` package can be used to scan the static constants in `CommonClientConfigs`, `ProducerConfig` or `ConsumerConfig`, then have those constants cross-checked against the user-supplied property names. The validation method will bail with a runtime exception if a given property name could not be resolved among the scanned constants.

The rest of this section will focus on the uncompromising, fully type-safe case. Expect a bit of coding. All sample code is provided at github.com/ekoutanov/effectivekafka[25], in the `src/main/java/effectivekafka/types` directory.

There are two classes in this example. The first defines a self-validating structure for containing producer client configuration. It has room for both expected properties and a free-form set of custom properties. The listing for `TypesafeProducerConfig` follows.

[25]https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/typesafeproducer

```
import static java.util.function.Predicate.*;

import java.lang.reflect.*;
import java.util.*;
import java.util.stream.*;

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

public final class TypesafeProducerConfig {
  public static final class UnsupportedPropertyException
      extends RuntimeException {
    private static final long serialVersionUID = 1L;

    private UnsupportedPropertyException(String s) { super(s); }
  }

  public static final class ConflictingPropertyException
      extends RuntimeException {
    private static final long serialVersionUID = 1L;

    private ConflictingPropertyException(String s) { super(s); }
  }

  private String bootstrapServers;

  private Class<? extends Serializer<?>> keySerializerClass;

  private Class<? extends Serializer<?>> valueSerializerClass;

  private final Map<String, Object> customEntries = new HashMap<>();

  public TypesafeProducerConfig
      withBootstrapServers(String bootstrapServers) {
    this.bootstrapServers = bootstrapServers;
    return this;
  }

  public TypesafeProducerConfig withKeySerializerClass(
        Class<? extends Serializer<?>> keySerializerClass) {
    this.keySerializerClass = keySerializerClass;
    return this;
  }

  public TypesafeProducerConfig withValueSerializerClass(
        Class<? extends Serializer<?>> valueSerializerClass) {
    this.valueSerializerClass = valueSerializerClass;
    return this;
  }

  public TypesafeProducerConfig withCustomEntry(String propertyName,
                                                Object value) {
    Objects.requireNonNull(propertyName,
                           "Property name cannot be null");
    customEntries.put(propertyName, value);
    return this;
  }

  public Map<String, Object> mapify() {
    final var stagingConfig = new HashMap<String, Object>();
    if (! customEntries.isEmpty()) {
      final var supportedKeys =
          scanClassesForPropertyNames(SecurityConfig.class,
                                      SaslConfigs.class,
                                      ProducerConfig.class);
      final var unsupportedKey = customEntries.keySet()
          .stream()
          .filter(not(supportedKeys::contains))
          .findAny();

      if (unsupportedKey.isPresent()) {
        throw new UnsupportedPropertyException(
            "Unsupported property " + unsupportedKey.get());
      }

      stagingConfig.putAll(customEntries);
    }

    Objects.requireNonNull(bootstrapServers,
                           "Bootstrap servers not set");
    tryInsertEntry(stagingConfig,
                   ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                   bootstrapServers);
    Objects.requireNonNull(keySerializerClass,
                       "Key serializer not set");
    tryInsertEntry(stagingConfig,
                   ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                   keySerializerClass.getName());
    Objects.requireNonNull(valueSerializerClass,
                           "Value serializer not set");
    tryInsertEntry(stagingConfig,
                   ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                   valueSerializerClass.getName());

    return stagingConfig;
  }

  private static void tryInsertEntry(Map<String, Object> staging,
                                     String key,
                                     Object value) {
    staging.compute(key, (__key, existingValue) -> {
      if (existingValue == null) {
        return value;
      } else {
        throw new ConflictingPropertyException("Property "
            + key + " conflicts with an expected property");
      }
    });
  }

  private static Set<String>
      scanClassesForPropertyNames(Class<?>... classes) {
    return Arrays.stream(classes)
        .map(Class::getFields)
        .flatMap(Arrays::stream)
        .filter(TypesafeProducerConfig::isFieldConstant)
        .filter(TypesafeProducerConfig::isFieldStringType)
        .filter(not(TypesafeProducerConfig::isFieldDoc))
        .map(TypesafeProducerConfig::retrieveField)
        .collect(Collectors.toSet());
  }

  private static boolean isFieldConstant(Field field) {
    return Modifier.isFinal(field.getModifiers())
        && Modifier.isStatic(field.getModifiers());
  }
  private static boolean isFieldStringType(Field field) {
    return field.getType().equals(String.class);
  }

  private static boolean isFieldDoc(Field field) {
    return field.getName().endsWith("_DOC");
  }

  private static String retrieveField(Field field) {
    try {
      return (String) field.get(null);
    } catch (IllegalArgumentException | IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
}
```

The `TypesafeProducerConfig` class defines a pair of public nested runtime exceptions — `UnsupportedPropertyExcept` and `ConflictingPropertyException`. The former will be thrown if the user provides a custom property with an unsupported name, trapping those pesky typos. The latter occurs when the custom property conflicts with an expected property. Both are conditions that we ideally would prefer to avoid before initialising the producer client.

Next, we declare our private attributes. This example expects three properties — `bootstrapServers`, `keySerializerClass`, and `valueSerializerClass`. For the serializers, we have taken the extra step of restricting their type to `java.lang.Class<? extends Serializer<?>>`, ensuring that only valid serializer implementations may be assigned. The `customEntries` attribute is responsible for accumulating any free-form entries, supplied in addition to the expected properties.

The `mapify()` method is responsible for converting the stored values into a form suitable for passing to a `KafkaProducer` constructor. It starts by checking if `customEntries` has at least one entry in it. If so, it invokes the `scanClassesForPropertyNames()` method, passing it the class definitions of `ProducerConfig` as well as some common security-related configuration classes. The result will be a set of strings harvested from those classes using reflection. The `ProducerConfig` class already imports the necessary constants from `CommonClientConfigs`, relieving us from having to scan `CommonClientConfigs` explicitly.

The actual implementation of `scanClassesForPropertyNames()` should hopefully be straightforward, requiring basic knowledge of Java 8 and the `java.util.stream` API. Essentially, it enumerates over the `public` fields, filtering those that happen to be constants (having `final` and `static` modifiers), are of a `java.lang.String` type, and are not suffixed with the string `_DOC`. This set of filters should round up all supported property names. The filtered fields are retrieved and packed into a `java.util.Set`. For convenience, the code listing of `scanClassesForPropertyNames()` is repeated below.

```
private static Set<String>
    scanClassesForPropertyNames(Class<?>... classes) {
  return Arrays.stream(classes)
      .map(Class::getFields)
      .flatMap(Arrays::stream)
      .filter(TypesafeProducerConfig::isFieldConstant)
      .filter(TypesafeProducerConfig::isFieldStringType)
      .filter(not(TypesafeProducerConfig::isFieldDoc))
      .map(TypesafeProducerConfig::retrieveField)
      .collect(Collectors.toSet());
}
```

The filter in `scanClassesForPropertyNames()` may also inadvertently include other string constants in the given `classes` array that happen to match its predicates. Kafka's maintainers haven't consistently differentiated between supported property names and other constants. The majority use the `_CONFIG` suffix to indicate a supported name; however, `ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION` has strayed from this convention. Short of specifying an exclusion filter that blacklists known stray constants, there is little we can do about this. A blacklist would create a long-term maintenance headache and is hardly worth the effort. The likelihood of a misspelt user-supplied property name colliding with a stray constant is negligible.

Let's now use our `TypesafeProducerConfig` to configure an actual client:

```
final var config = new TypesafeProducerConfig()
    .withBootstrapServers("localhost:9092")
    .withKeySerializerClass(StringSerializer.class)
    .withValueSerializerClass(StringSerializer.class)
    .withCustomEntry(
        ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);

try (var producer = new KafkaProducer<>(config.mapify())) {
  // do something with producer
}
```

The use of `TypesafeProducerConfig` follows a fluent style of method chaining; the code is compact but readable. But crucially, it is now bullet-proof. You can see it for yourself: feed an invalid property name into `withCustomEntry()` and watch `config.mapify()` bail with a runtime exception, avoiding the initialisation of `KafkaProducer`.

We can also populate a `TypesafeProducerConfig` object from a JSON or YAML configuration file using a parser such as Jackson, available at github.com/FasterXML/jackson[26]. Jackson supports a

[26]https://github.com/FasterXML/jackson

wide range of formats, including JSON, YAML, TOML, and even plain `.properties` files. To map the parsed configuration document to a `TypesafeProducerConfig` instance, Jackson requires either the addition of Jackson-specific annotations to our class or defining a custom deserializer.

Alternatively, if using an application framework such as Spring Boot or Micronaut, we can wire a `TypesafeProducerConfig` object into the framework's native configuration mechanism. This typically requires the addition of setter methods to make the encapsulated attributes writable by the framework. The design of `TypesafeProducerConfig` defers all validation until the `mapify()` method is invoked, thereby remaining agnostic of how its attributes are populated.

The examples in this book are intentionally decoupled from application frameworks. The intention is to demonstrate the correct and practical uses of Kafka, using the simplest and most succinct examples. These examples are intended to be run as standalone applications using any IDE.

You might have noticed that the example still used a constant to specify the custom property name, in spite of having a reliable mechanism for detecting misspelt names early, before client initialisation. Using constants traps errors at compile-time, which is the holy grail of building robust software.

While the `TypesafeProducerConfig` example solves the validation problem in its current guise, the presented solution is not very reusable. It would take a fair amount of copying and pasting of code to apply this pattern to different configuration classes, even if the variations are minor. With a modicum of refactoring, we can turn the underlying ideas into a generalised model for configuration management.

If you are interested, take a look at github.com/ekoutanov/effectivekafka[a]. The `src/main/java/effectivekafka/config` directory contains an example of how this can be achieved. The `AbstractClientConfig` class serves as an abstract base class for a user-defined configuration class. The base class contains the `customEntries` attribute, as well as the validation logic for ensuring the correctness of property names. The deriving class is responsible for providing the expected attributes and the related validation logic. The `mapify()` method lives in the base class, and will invoke the subclass to harvest its expected configuration properties.

[a]https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/config

---

This chapter identified the challenges with configuring Kafka clients — namely, Kafka's permissive stance on validating the user-supplied configuration and treating configuration entries as a typeless map of string-based keys to arbitrary values.

The problem of validating configuration for the general case can be solved by creating an intermediate configuration class. This class houses the expected configuration items as type-safe attributes, as well as free-form configuration, which is validated using reflection. A configuration class acts as an intermediate placeholder for configuration entries, enabling the application to proactively validate the client configuration before instantiating the client, in some cases using nothing more than compile-time type safety.