3 Schema registry

This chapter covers

- Using bytes means serialization rules
- What is a schema and why you need to use one
- What is Schema Registry?
- Ensuring compatibility with changes - schema evolution
- Understanding subject names
- Reusing schemas with references

In chapter 2, you learned about the heart of the Kafka streaming platform, the Kafka broker. In particular, you learned how the broker is the storage layer appending incoming messages to a topic, serving as an immutable, distributed log of events. A topic represents the directory containing the log file(s).

Since the producers send messages over the network, they need to be serialized first into binary format, in other words an array of bytes. The Kafka broker does not change the messages in any way, it stores them in the same format. It's the same when the broker responds to fetch requests from consumers, it retrieves the already serialized messages and sends them over the network.

By only working with messages as arrays of bytes, the broker is completely agnostic to the data type the messages represent and completely independent of the applications that are producing and consuming the messages and the programming languages those applications use. By decoupling the broker from the data format, any client using the Kafka protocol can produce or consume messages.

While bytes are great for storage and transport over the network, developers are far more efficient working at a higher level of abstraction; the object. So where does this transformation

from object to bytes and bytes to object occur then? At the client level in the producers and consumers of messages.

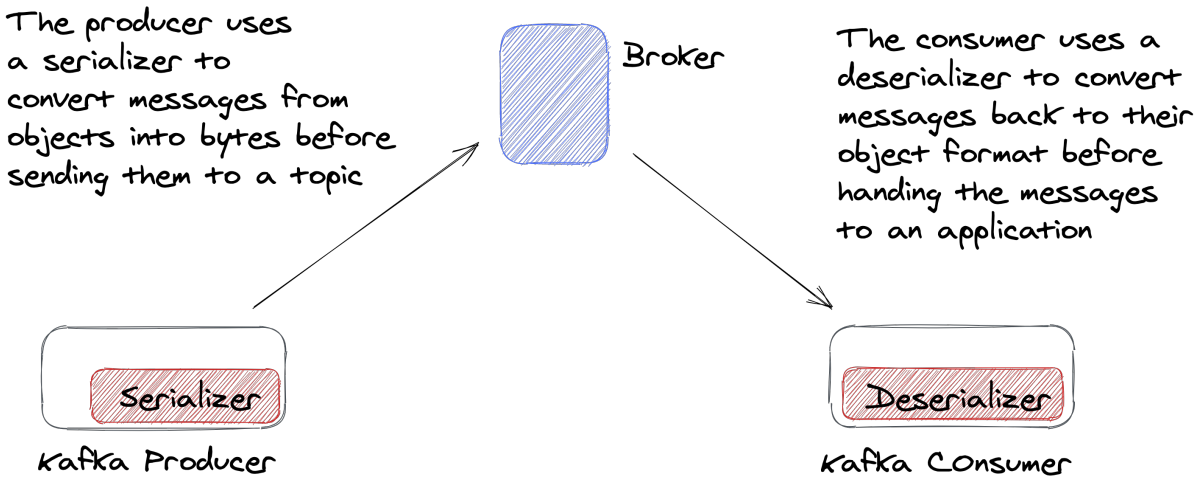

Figure 3.1 The conversion of objects to bytes and bytes to objects happens at the client level

Looking at this illustration, the message producer uses an instance of a `Serializer` to convert the message object into bytes before sending it to the topic on the broker. The message consumer does the opposite process, it receives bytes from the topic, and uses an instance of a `Deserializer` to convert the bytes back into the same object format.

The producer and consumer are decoupled from the (de)serializers; they simply call either the `serialize` or `deserialize` methods.

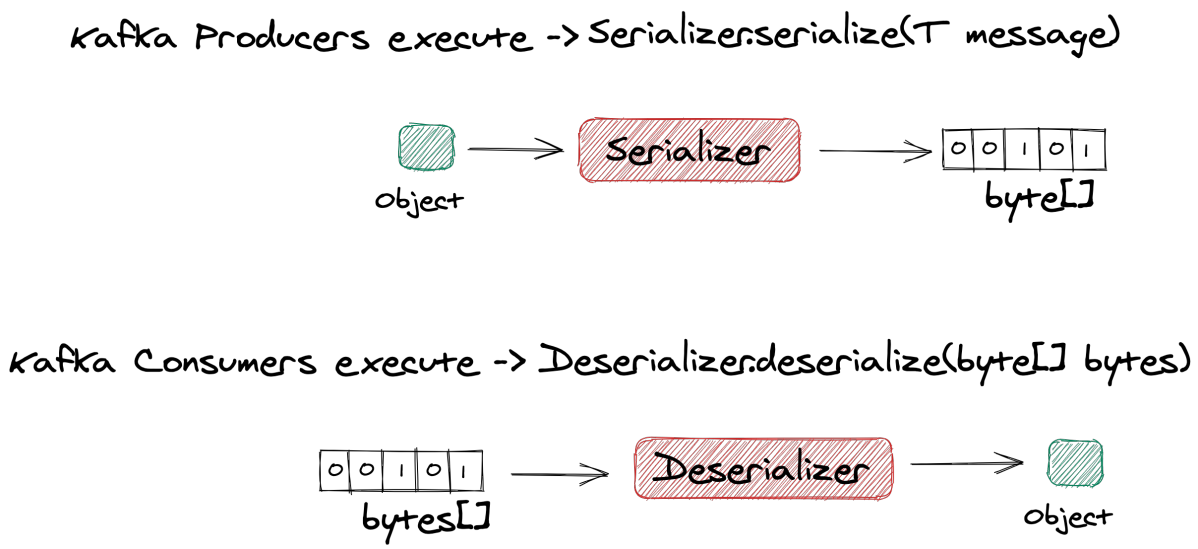

Figure 3.2 The serializer and deserializer are agnostic of the producer and consumer and perform the expected action when the serialize and deserialize methods are called

As depicted in this illustration, the producer expects to use an instance of the `Serializer` interface and just calls the `Serializer.serialize` method passing in an object of a given type

and getting back bytes. The consumer works with the `Deserializer` interface. The consumer provides an array of bytes to the `Deserializer.deserialize` method and receives an object of a given type in return.

The producer and consumer get the (de)serializers via configuration parameters and we'll see examples of this later in the chapter.

> **NOTE** I'm mentioning producers and consumers here and throughout the chapter, but we'll only go into enough detail to understand the context required for this chapter. We'll cover producer and consumer client details in the next chapter.

The point I'm trying to emphasize here is that for a given topic the object type the producer serializes is expected to be the exact same object type that a consumer deserializes. Since producers and consumers are completely agnostic of each other ***these messages or event domain objects represent an implicit contract between the producers and consumers***.

So now the question is does something exist that developers of producers and consumers can use that informs them of the proper structure of messages? The answer to that question is yes, the schema.

3.1 What is a schema and why you need to use one

When you mention the word schema to developers, there's a good chance their first thought is of database schemas. A database schema describes the structure of the database, including the names and startups of the columns in database tables and the relationship between tables. But the schema I'm referring to here, while similar in purpose, is not quite the same thing.

For our purposes what I'm referring to is a ***language agnostic description of an object, including the name, the fields on the object and the type of each field***. Here's an example of a potential schema in json format

Listing 3.1 Basic example of a schema in json format

```
{
"name":"Person",     ❶
  "fields": [        ❷
    {"name": "name", "type":"string"},   ❸
    {"name": "age", "type": "int"},
    {"name": "email", "type":"string"}
  ]
}
```

❶ The name of the object

❷ Defining the fields on the object

❸ The names of the fields and their types

Here our fictional schema describes an object named `Person` with fields we'd expect to find on such an object. Now we have a structured description of an object that producers and consumers can use as an agreement or contract on what the object should look like before and after serialization. I'll cover details on how you use schemas in message construction and (de)serialization in an upcoming section.

But for now I'd like review some key points we've established so far:

- The Kafka broker only works with messages in binary format (byte arrays)
- Kafka producers and consumers are responsible for the (de)serialization of messages. Additionally, since these two are unaware of each other, the records form a contract between them.

And we also learned that we can make the contract between producers and consumers explicit by using a schema. So we have our *why* for using a schema, but what we've defined so far is a bit abstract and we need to answer these questions for the *how* :

- How do you put schemas to use in your application development lifecyle?
- Given that serialization and deserialization is decoupled from the Kafka producers and consumers how can they use serialization that ensures messages are in the correct format?
- How do you enforce the correct version of a schema to use? After all changes are inevitable

The answer to these *how* questions is Schema Registry.

3.1.1 What is Schema Registry?

Schema Registry provides a centralized application for storing schemas, schema validation and sane schema evolution (message structure changes) procedures. Perhaps more importantly, it serves as the source of truth of schemas that producer and consumer clients can easily discover. Schema Registry provides serializers and deserializers that you can configure Kafka Producers and Kafka Consumers easing the development for applications working with Kafka.

The Schema Registry serializing code supports schemas from the serialization frameworks Avro (avro.apache.org/docs/current/) and Protocol Buffers (developers.google.com/protocol-buffers). Note that I'll refer to Protocol Buffers as "Protobuf" going forward. Additionally Schema Registry supports schemas written using the JSON Schema (json-schema.org/), but this is more of a specification vs a framework. I'll get into working with Avro, Protobuf JSON Schema as we progress through the chapter, but for now let's take a high-level view of how Schema Registry works:

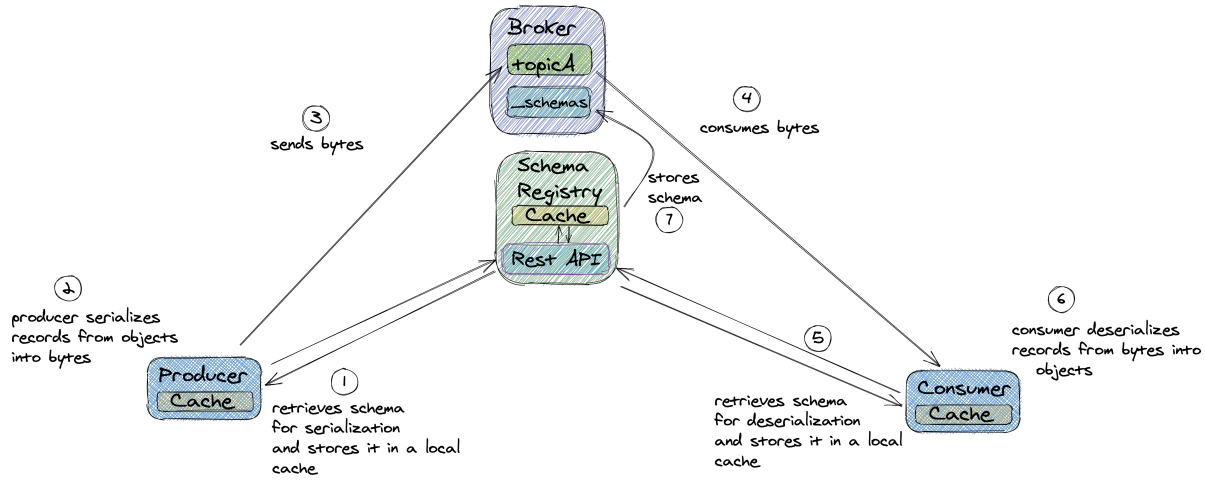

Figure 3.3 Schema registry ensures consistent data format between producers and consumers

Let's quickly walk through how Schema Registry works based on this illustration

1. As a produce calls the `serialize` method, a Schema Registry aware serializer retrieves the schema (via HTTP) and stores it in its local cache
2. The serializer embedded in the producer serializes the record
3. The producer sends the serialized message (bytes) to Kafka
4. A consumer reads in the bytes
5. The Schema Registry aware deserializer in the consumer retrieves the schema and stores it in its local cache
6. The consumer deserializes the the bytes based on the schema
7. The Schema Registry servers produces a message with the schema so that it's stored in the `__schemas` topic

TIP **While I'm presenting Schema Registry as an important part of the Kafka event streaming platform, it's not required. Remember Kafka producers and consumers are decoupled from the serializers and deserializers they use. As long as you provide a class that implements the appropriate interface, they'll work fine with the producer or consumer. But you will lose the validation checks that come from using Schema Registry. I'll cover serializing without Schema Registry at the end of this chapter.**

While the previous illustration gave you a good idea of how schema registry works, there's an important detail I'd like to point out here. While it's true that the serializer or deserializer will reach out to Schema Registry to retrieve a schema for a given record type, it only does so ***once***, the first time it encounters a record type it doesn't have the schema for. After that, the schema needed for (de)serialization operations is retrieved from local cache.

3.1.2 Getting Schema Registry

Our first step is to get Schema Registry up and running. Again you'll use docker-compose to speed up your learning and development process. We'll cover installing Schema Registry from a binary download and other options in an appendix. But for now just grab the `docker-compose.yml` file from the chapter_3 directory in the source code for the book.

This file is very similar to the `docker-compose.yml` file you used in chapter two. But in addition to the Zookeeper and Kafka images, there is an entry for a Schema Registry image as well. Go ahead and run `docker-compose up -d`. To refresh your memory about the docker commands the `-d` is for "detached" mode meaning the docker containers run in the background freeing up the terminal window you've executed the command in.

3.1.3 Architecture

Before we go into the details of how you work with Schema Registry, it would be good to get high level view of how it's designed. Schema Registry is a distributed application that lives outside the Kafka brokers. Clients communicate with Schema Registry via a REST API. A client could be a serializer (producer), deserializer (consumer), a build tool plugin, or a command line request using curl. I'll cover using build tool plugins, gradle in this case, in an upcoming section soon.

Schema Registry uses Kafka as storage (write-ahead-log) of all its schemas in `__schemas` which is a single partitioned, compacted topic. It has a primary architecture meaning there is one leader node in the deployment and the other nodes are secondary.

> **NOTE** **The double underscore characters are a Kafka topic naming convention denoting internal topics not meant for public consumption. From this point forward we'll refer to this topic simply as `schemas`.**

What this means is that only the primary node in the deployment writes to the schemas topic. Any node in the deployment will accept a request to store or update a schema, but secondary nodes forward the request to the primary node. Let's look at an illustration to demonstrate:

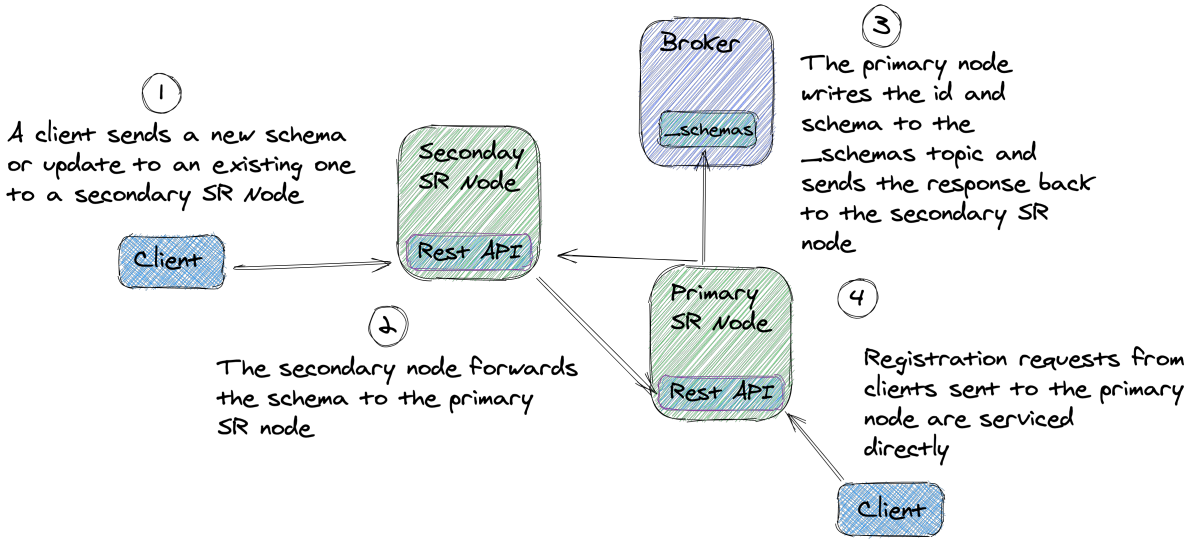

Figure 3.4 Schema Registry is a distributed application where only the primary node communicates with Kafka

Anytime a client registers or updates a schema, the primary node produces a record to the {underscore}schemas topic. Schema Registry uses a Kafka producer for writing and all the nodes use a consumer for reading updates. So you can see that Schema Registry's local state is backed up in a Kafka topic making schemas very durable.

NOTE	**When working with Schema Registry throughout all the examples in the book you'll only use a single node deployment suitable for local development.**

But all Schema Registry nodes serve read requests from clients. If any secondary nodes receive a registration or update request, it is forwarded to the primary node. Then the secondary node returns the response from the primary node. Let's take a look at an illustration of this architecture to solidify your mental model of how this works:

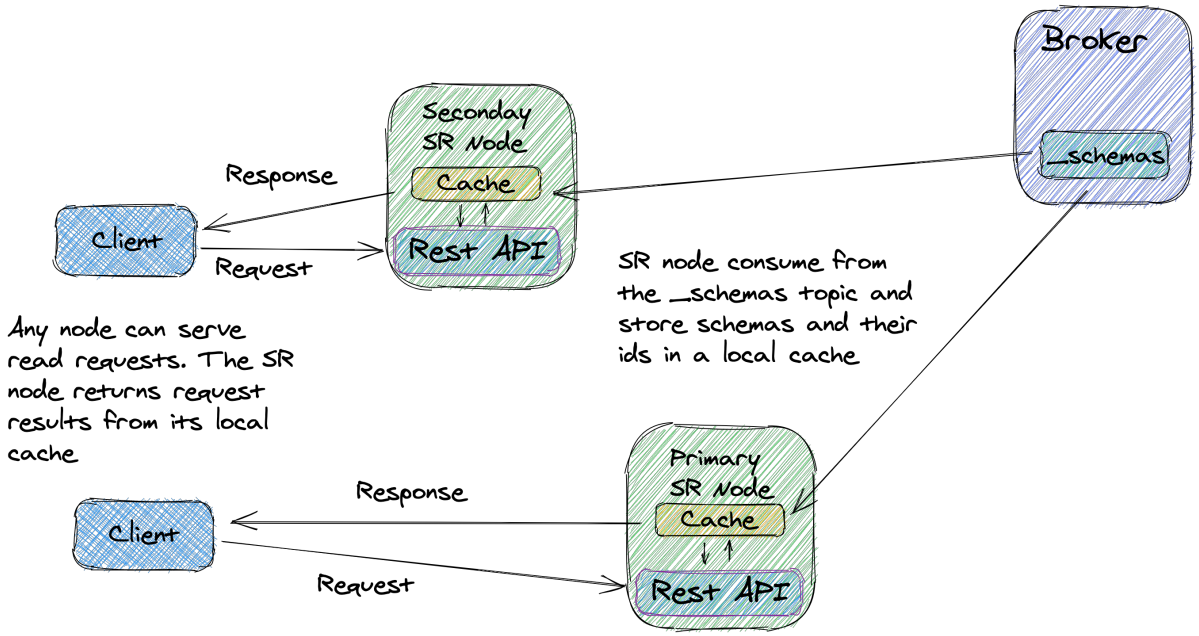

Figure 3.5 All Schema Registry nodes can serve read requests

Now that we've given an overview of the architecture, let's get to work by issuing a few basic commands using Schema Registry REST API.

3.1.4 Communication - Using Schema Registry's REST API

So far we've covered how Schema Registry works, but now it's time to see it in action by uploading a schema then running some additional commands available to get more information about your uploaded schema. For the initial commands you'll use `curl` and `jq` in a terminal window.

NOTE `curl` (curl.se/) is a command line utility for working with data via a URLs. `jq` (stedolan.github.io/jq/) is a command-line json processor. For installing jq for your platform you can visit the jq download site stedolan.github.io/jq/download/. For curl it should come installed on Windows 10+ and Mac Os. On Linux you can install via a package manager. If you are using Mac OS you can install both using homebrew - `brew.sh/`.

In later sections you'll use a `gradle` plugin for your interactions with Schema Registry. After you get an idea of how the different REST API calls work, you'll move on to using the gradle plugins and using some basic producer and consumer examples to see the serialization in action.

Typically you'll use the build tool plugins for performing Schema Registry actions. First they make he development process much faster rather than having run the API calls from the command line, and secondly they will automatically generate source code from schemas. We'll cover using build tool plugins in an upcoming section.

> **NOTE** There are Maven and Gradle plugins for working with Schema Registry, but the source code project for the book uses Gradle, so that's the plugin you'll use.

REGISTER A SCHEMA

Before we get started make sure you've run `docker-compose up -d` so that we'll have a Schema Registry instance running. But there's going to be nothing registered so your first step is to register a schema. Let's have a little fun and create a schema for Marvel Comic super heroes, the Avengers. You'll use Avro for your first schema and let's take a second now to discuss the format:

Listing 3.2 Avro schema for Avengers

```
{"namespace": "bbejeck.chapter_3",   ❶
 "type": "record",                   ❷
 "name": "Avenger",                  ❸
 "fields": [                         ❹
     {"name": "name", "type": "string"},
     {"name": "real_name", "type": "string"},          ❺
     {"name": "movies", "type":
                      {"type": "array", "items": "string"},
      "default": []    ❻
    }
  ]
}
```

❶ The namespace uniquely identifies the schema. For generated Java code the namespace is the package name.

❷ The type is `record` which is a complex type. Other complex types are `enums`, `arrays`, `maps`, `unions` and `fixed`. We'll go into more detail about Avro types later in this chapter.

❸ The name of the record

❹ Declaring the fields of the record

❺ Describing the individual fields. Fields in Avro are either simple or complex.

❻ Providing a default value. If the serialized bytes don't contain this field, Avro uses the default value when deserializing.

You define Avro schemas in JSON format. You'll use this same schema file in a upcoming section when we discuss the gradle plugin for code generation and interactions with Schema Registry. Since Schema Registry supports Protobuf and JSON Schema formats as well let's take a look at the same type in those schema formats here as well:

Listing 3.3 Protobuf schema for Avengers

```
syntax = "proto3";    ❶

package bbejeck.chapter_3.proto;    ❷

option java_outer_classname = "AvengerProto";    ❸

message Avenger {    ❹
    string name = 1;    ❺
    string real_name = 2;
    repeated string movies = 3;    ❻

}
```

❶ Defining the version of Protobuf, we're using version three in this book

❷ Declaring the package name

❸ Specifying the name of the outer class, otherwise the name of the proto file is used

❹ Defining the message

❺ Unique field number

❻ A repeated field; corresponds to a list

The Protobuf schema looks closer to regular code as the format is not JSON. Protobuf uses the numbers you see assigned to the fields to identify those fields in the message binary format. While Avro specification allows for setting default values, in Protobuf (version 3), every field is considered optional, but you don't provide a default value. Instead, Protobuf uses the type of the field to determine the default. For example the default for a numerical field is 0, for strings it's an empty string and repeated fields are an empty list.

NOTE **Protobuf is a deep subject and since this book is about the Kafka event streaming pattern, I'll only cover enough of the Protobuf specification for you to get started and feel comfortable using it. For full details you can read the language guide found here developers.google.com/protocol-buffers/docs/proto3.**

Now let's take a look at the JSON Schema version:

Listing 3.4 JSON Schema schema for Avengers

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",     ❶
  "title": "Avenger",
  "description": "A JSON schema of Avenger object",
  "type": "object",                ❷
  "javaType": "bbejeck.chapter_3.json.SimpleAvengerJson",    ❸
  "properties": {       ❹
    "name": {
      "type": "string"
    },
    "realName": {
      "type": "string"
    },
    "movies": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "default": []    ❺
    }
  },
  "required": [
    "name",
    "realName"
  ]
}
```

❶ Referencing the specific schema spec

❷ Specifying the type is an object

❸ The javaType used when deserializing

❹ Listing the fields of the object

❺ Specifying a default value

The JSON Schema schema resembles the Avro version as both use JSON for the schema file. The biggest difference between the two is that in the JSON Schema you list the object fields under a `properties` element vs. a `fields` array and in the fields themselves you simply declare the name vs. having a `name` element.

NOTE **Please note there is a difference between a schema written in JSON format and one that follows the JSON Schema format. JSON Schema is "a vocabulary that allows you to annotate and validate JSON documents.". As with Avro and Protobuf, I'm going to focus on enough for you to get going using it in your projects, but for in-depth coverage you should visit json-schema.org/ for more information.**

I've shown the different schema formats here for comparison. But in the rest of the chapter, I'll usually only show one version of a schema in an example to save space. But the source code will contain examples for all three supported types.

Now that we've reviewed the schemas, let's go ahead and register one. The command to register a schema with REST API on the command-line looks like this

Listing 3.5 Register a schema on the command line

```
jq '. | {schema: tojson}' src/main/avro/avenger.avsc | \     ❶
curl -s -X POST http://localhost:8081/subjects/avro-avengers-value/versions\    ❷
         -H "Content-Type: application/vnd.schemaregistry.v1+json" \    ❸
         -d @-  \   ❹
         | jq   ❺
```

❶ Using the the jq `tojson` function to format the avenger.avsc file (new lines aren't valid json) for uploading, then pipe the result to the curl command

❷ The POST URL for adding the schema, the `-s` flag suppresses the progress info output from `curl`

❸ The content header

❹ The `-d` flag specifies the data and `@-` means read from STDIN i.e. the data provided by the jq command preceding the curl command

❺ Piping the json response through `jq` to get a nicely formatted response

The result you see from running this command should look like this:

Listing 3.6 Expected response from uploading a schema

```
{
  "id": 1
}
```

The response from the `POST` request is the id that Schema Registry assigned to the new schema. Schema Registry assigns a unique id (a monotonically increasing number) to each newly added schema. Clients use this id for storing schemas in their local cache.

Before we move on to another command I want to call your attention to annotation 2, specifically this part - `subjects/avro-avengers-value/`, it specifies the subject name for the schema. Schema Registry uses the subject name to manage the scope of any changes made to a schema. In this case it's confined to `avro-avengers-value` which means that values (in the key-value pairs) going into the `avro-avengers` topic need to be in the format of the registered schema. We'll cover subject names and the role they have in making changes in an upcoming section.

Next, let's take a look at some of the available commands you can use to retrieve information from Schema Registry.

Imagine you are working on building a new application to work with Kafka. You've heard about Schema Registry and you'd like to take a look at particular schema one of your co-workers

developed, but you can't remember the name and it's the weekend and you don't want to bother anyone. What you can do is list all the subjects of registered schemas with the following command:

Listing 3.7 Listing the subjects of registered schemas

```
curl -s "http://localhost:8081/subjects" | jq
```

The response from this command is a json array of all the subjects. Since we've only registered once schema so far the results should look like this

```
[
  "avro-avengers-value"
]
```

Great, you find here what you are looking for, the schema registered for the `avro-avengers` topic.

Now let's consider there's been some changes to the latest schema and you'd like to see what the previous version was. The problem is you don't know the version history. The next command shows you all of versions for a given schema

Listing 3.8 Getting all versions for a given schema

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/versions" | jq
```

This command returns a json array of the versions of the given schema. In our case here the results should look like this:

[1]

Now that you have the version number you need, now you can run another command to retrieve the schema at a specific version:

Listing 3.9 Retrieving a specific version of a schema

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/versions/1"\
 | jq '.'
```

After running this command you should see something resembling this:

```
{
  "subject": "avro-avengers-value",
  "version": 1,
  "id": 1,
  "schema": "{\"type\":\"record\",\"name\":\"AvengerAvro\",
      \"namespace\":\"bbejeck.chapter_3.avro\",\"fields\"
      :[{\"name\":\"name\",\"type\":\"string\"},{\"name\"
        :\"real_name\",\"type\":\"string\"},{\"name\"
          :\"movies\",\"type\":{\"type\":\"array\"
            ,\"items\":\"string\"},\"default\":[]}]}"
}
```

The value for the `schema` field is formatted as a string, so the quotes are escaped and all new-line characters are removed.

With a couple of quick commands from a console window, you've been able to find a schema, determine the version history and view the schema of a particular version.

As a side note, if you don't care about previous versions of a schema and you only want the latest one, you don't need to know the actual latest version number. You can use the following REST API call to retrieve the latest schema:

Listing 3.10 Getting the latest version of a schema

```
curl -s "http://localhost:8081/subjects/avro-avengers-value/
  versions/latest" | jq '.'
```

I won't show the results of this command here, as it is identical to the previous command.

That has been a quick tour of some of the commands available in the REST API for Schema Registry. This just a small subset of the available commands. For a full reference go to docs.confluent.io/platform/current/schema-registry/develop/api.html#sr-api-reference.

Next we'll move on to using gradle plugins for working with Schema Registry and Avro, Protobuf and JSON Schema schemas.

3.1.5 Plugins and serialization platform tools

So far you've learned that the event objects written by producers and read by consumers represent the contract between the producer and consumer clients. You've also learned that this "implicit" contract can be a concrete one in the form of a schema. Additionally you've seen how you can use Schema Registry to store the schemas and make them available to the producer and consumer clients when the need to serialize and deserialize records.

In the upcoming sections you'll see even more functionality with Schema Registry. I'm referring to testing schemas for compatibility, different compatibility modes and how it can make changing or evolving a schema a relatively painless process for the involved producer and consumer clients.

But so far, you've only worked with a schema file and that's still a bit abstract. As I said earlier in the chapter, developers work with objects when building applications. So our next step is to see how we can convert these schema files into concrete objects you can use in an application.

Schema Registry supports schemas in Avro, Protobuf and JSON Schema format. Avro and Protobuf are serialization platforms that provide tooling for working with schemas in their respective formats. One of the most important tools is the ability to generate objects from the schemas.

Since JSON Schema is a standard and not a library or platform you'll need to use an open source tool for code generation. For this book we're using the github.com/eirnym/js2p-gradle project. For (de)serialization without Schema Registry I would recommend using `ObjectMapper` from the github.com/FasterXML/jackson-databind project.

Generating code from the schema makes your life as developer easier, as it automates the repetitive, boilerplate process of creating domain objects. Additionally since you maintain the schemas in source control (git in our case), the chance for error, such as making a field string type when it should be a long, when creating the domain objects is all but eliminated.

Also when making a change to a schema, you just commit the change and other developers pull the update and re-generate the code and everyone is unsung fairly quickly.

In this book we'll use the gradle build tool (gradle.org/) to manage the book's source code. Fortunately there are gradle plugins we can use for working with Schema Registry, Avro, Protobuf, and JSON Schema. Specifically, we'll use the following plugins

- github.com/ImFlog/schema-registry-plugin - For interacting with Schema Registry i.e. testing schema compatibility, registering schemas, and configuring schema compatibility
- github.com/davidmc24/gradle-avro-plugin - Used for Java code generation from Avro schema (`.avsc`) files.
- github.com/google/protobuf-gradle-plugin - Used for Java code generation from Protobuf schema (`.proto`) files
- github.com/eirnym/js2p-gradle - Used for Java code generation for schemas using the JSON Schema specification.

NOTE **It's important to note the distinction between schema files written in JSON such as Avro schemas and those files using the JSON Schema format (json-schema.org/). In the case of Avro files they are written as json, but follow the Avro specification. With the JSON Schema files they follow the official specification for JSON Schemas.**

By using the gradle plugins for Avro, Protobuf and JSON Schema, you don't need to learn how to use the individual tools for each component, the plugins handle all the work. We'll also use a gradle plugin for handling most of the interactions with Schema Registry.

Let's get started by uploading a schema using a gradle command instead of a REST API command in the console.

UPLOADING A SCHEMA FILE

The first thing we'll do is use gradle to register a schema. We'll use the same Avro schema from the REST API commands section. Now to upload the schema, make sure to change your current directory (`CD`) into the base directory of project and run this gradle command:

```
./gradlew streams:registerSchemasTask
```

After running this command you should see something like `BUILD SUCCESSFUL` in the console. Notice that all you needed to enter on the command line is the name of the gradle task (from the schema-registry-plugin) and the task registers all the schema inside the `register { }` block in the `streams/build.gradle` file.

Now let's take a look at the configuration of the Schema Registry plugin in the `streams/build.gradle` file.

Listing 3.11 Configuration for Schema Registry plugin in streams/build.gradle

```
schemaRegistry {     ❶
    url = 'http://localhost:8081'   ❷


register {
        subject('avro-avengers-value',   ❸
                'src/main/avro/avenger.avsc',   ❹
                'AVRO')   ❺

     //other entries left out for clarity
    }

  // other configurations left out for clarity
}
```

❶ Start of the Schema Registry configuration block in the build.gradle file

❷ Specifying the URL to connect to Schema Registry

❸ Registering a schema by subject name

❹ Specifying Avro schema file to register

❺ The type of the schema you are registering

In the `register` block you provide the same information, just in a format of a method call vs. a URL in a REST call. Under the covers the plugin code is still using the Schema Registry REST API via a `SchemaRegistryClient`. As side note, in the source code you'll notice there are several entries in the `register` block. You'll use all of them when go through the examples in the source code.

We'll cover using more gradle Schema Registry tasks soon, but let's move on to generating code from a schema.

GENERATING CODE FROM SCHEMAS

As I said earlier, one of the best advantages of using the Avro and Protobuf platforms is the code generation tools. Using the gradle plugin for these these tools takes the convenience a bit further by abstracting away the details of using the individual tools. To generate the objects represented by the schemas all you need to do is run this gradle task:

Listing 3.12 Generating the model objects

```
./gradlew clean build
```

Running this gradle command generates Java code for all the types Avro, Protobuf, and JSON Schema for the schemas in the project. Now we should talk about where you place the schemas in the project. The default locations for the Avro and Protobuf schemas are the `src/main/avro` and `src/main/proto` directories, respectively. The location for the JSON Schema schemas is the `src/main/json` directory, but you need to explicitly configure this in the `build.gradle` file:

Listing 3.13 Configure the location of JSON Schema schema files

```
jsonSchema2Pojo {

  source = files("${project.projectDir}/src/main/json")   ❶
  targetDirectory = file("${project.buildDir}/generated-main-json-java")  ❷
  // other configurations left out for clarity
}
```

❶ The `source` configuration specifies where the generation tools can locate the schemas

❷ The `targetDirectory` is were tool writes the generated Java objects

NOTE **All examples here refer to the schemas found in the `streams` sub-directory unless otherwise specified.**

Here you can see the configuration of the input and output directories for the `js2p-gradle` plugin. The Avro plugin, by default, places the generated files in a sub-directory under the `build` directory named `generated-main-avro-java`.

For Protobuf we configure the output directory to match the pattern of JSON Schema and Avro in the `Protobuf` block of the `build.gradle` file like this:

Listing 3.14 Configure Protobuf output

```
protobuf {
    generatedFilesBaseDir = "${project.buildDir}
      /generated-main-proto-java"   ❶

    protoc {
        artifact = 'com.google.protobuf:protoc:3.15.3'   ❷
    }
}
```

❶ The output directory for the Java files generated from Protobuf schema

❷ Specifying the location of the protoc compiler

I'd to take a quick second to discuss annotation two for a moment. To use Protobuf you need to have the compiler `protoc` installed. By default the plugin searches for a `protoc` executable. But we can use a pre-compiled version of `protoc` from Maven Central, which means you don't have to explicitly install it. But if you prefer to use your local install, you can specify the path inside the `protoc` block with `path = path/to/protoc/compiler`.

So we've wrapped up generating code from the schemas, now it's time to run an end-to-end example

END TO END EXAMPLE

At this point we're going to take everything you've learned so far and run a simple end-to-end example. So far, you have registered the schemas and generated the Java files you need from them. So your next steps are to:

- Create some domain objects from the generated Java files
- Produce your created objects to a Kafka topic
- Consume the objects you just sent from the same Kafka topic

While parts two and three from the list above seem to have more to do with clients than Schema Registry, I want to think about it from this perspective. You're creating instances of Java objects created from the schema files, so pay attention to fields and notice how the objects conform to the structure of the schema. Secondly, focus on the Schema Registry related configuration items, serializer or deserializer and the URL for communicating with Schema Registry.

NOTE **In this example you will use a Kafka Producer and Kafka Consumer, but I won't cover any of the details of working with them. If you're unfamiliar with the producer and consumer clients that's fine. I'll go into detail about producers and consumers in the next chapter. But for now just go through the examples as is.**

If you haven't already registered the schema files and generated the Java code, let's do so now. I'll put the steps here again and make sure you have run `docker-compose up -d` to ensure your Kafka broker and Schema Registry are running.

Listing 3.15 Register schemas and generate Java files

```
./gradlew streams:registerSchemasTask   ❶

./gradlew clean build   ❷
```

❶ Register the schema files

❷ Build the Java objects from schemas

Now let's focus on the Schema Registry specific configurations. Go to the source code and take a look at the `bbejeck.chapter_3.producer.BaseProducer` class. For now we only want to look at the following two configurations, we'll cover more configurations for the producer in the next chapter:

```
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    keySerializer);   ❶
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");   ❷
```

❶ Specifying the serializer to use

❷ Setting the location of Schema registry

The first configuration sets the `Serializer` the producer will use. Remember, the `KafkaProducer` is decoupled from the type of the `Serializer`, it simply calls the `serialize` method and gets back an array of bytes to send. So the responsibility for providing the correct `Serializer` class is up to you.

In this case we're going to work with objects generated from an Avro schema, so you use the `KafkaAvroSerializer`. If you look at the `bbejeck.chapter_3.producer.avro.AvroProducer` class (which extends the `BaseProducer`) you see it pass the `KafkaAvroSerializer.class` to the parent object constructor. The second configuration specifies the HTTP endpoint that the `Serializer` uses for communicating with Schema Registry. These configurations enable the interactions described in the illustration "Schema registry ensures consistent data format between producers and consumers" above.

Next, let's take a quick look at creating an object:

Listing 3.16 Instantiating a an object from the generated code

```
var blackWidow = AvengerAvro.newBuilder()
                .setName("Black Widow")
                .setRealName("Natasha Romanova")
                .setMovies(List.of("Avengers", "Infinity Wars",
                  "End Game")).build();
```

OK, you're thinking now, "this code creates an object, what's the big deal?". While it could be a minor point, but it's more what you can't do here that I'm trying to drive home. You can only populate the expected fields with the correct types, enforcing the contract of producing records in the expected format. Of course you could update the schema and regenerate the code.

But by making changes, you have to register the new schema and the changes have to match the current compatibility format for the subject-name. So now can see now how Schema Registry enforces the "contract" between producers and consumers. We'll cover compatibility modes and the allowed changes in an upcoming section.

Now let's run the following gradle command to produce the objects to `avro-avengers` topic.

Listing 3.17 Running the AvroProducer

```
./gradlew streams:runAvroProducer
```

After running this command you'll see some output similar to this:

```
  DEBUG [main] bbejeck.chapter_3.producer.BaseProducer - Producing records
 [{"name": "Black Widow", "real_name": "Natasha Romanova", "movies":
["Avengers", "Infinity Wars", "End Game"]},
{"name": "Hulk", "real_name": "Dr. Bruce Banner", "movies":
["Avengers", "Ragnarok", "Infinity Wars"]},
{"name": "Thor", "real_name": "Thor", "movies":
["Dark Universe", "Ragnarok", "Avengers"]}]
```

After the application produces these few records it shuts itself down.

IMPORTANT **It's important to make sure to run this command exactly as shown here including the preceding : character. We have three different gradle modules for our Schema Registry exercises. We need to make sure the command we run are for the specific module. In this case the : executes the main module only, otherwise it will run the producer for all modules and the example will fail.**

Now running this command doesn't do anything exciting, but it demonstrate the ease of serializing by using Schema Registry. The producer retrieves the schema stores it locally and sends the records to Kafka in the correct serialized format. All without you having to write any serialization or domain model code. Congratulations you have sent serialize records to Kafka!

TIP **It could be instructive to look a the log file generated from running this command. It can be found in the `logs/` directory of the provided source code. The log4j configuration overwrites the log file with each run, so be sure to inspect it before running the next step.**

Now let's run a consumer which will deserialize the records. But as we did with the producer, we're going to focus on the configuration required for deserialization and working with Schema Registry:

Listing 3.18 Consumer configuration for using Avro

```
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    KafkaAvroDeserializer.class);   ❶
consumerProps.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
    true);   ❷
consumerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081");   ❸
```

❶ Using Avro deserialization

❷ Configuring to use a SpecificAvroReader

❸ The host:port for Schema Registry

You'll notice that in the second annotation you are setting the `SPECIFIC_AVRO_READER_CONFIG` to `true`. What does the `SPECIFIC_AVRO_READER_CONFIG` setting do? Well to answer that question let's take a slight detour in our conversation to discuss working with Avro, Protobuf, and JSON Schema serialized objects.

When deserializing one of the Avro, Protobuf, or JSON Schema objects there is a concept of deserializing the specific object type or a non-specific "container" object. For example, with the `SPECIFIC_AVRO_READER_CONFIG` set to true, the deserializer inside the consumer, will return an object of type `AvroAvenger` the **specific** object type.

However had you set the `SPECIFIC_AVRO_READER_CONFIG` to `false`, the deserializer returns an object of type `GenericRecord`. The returned `GenericRecrod` still follows the same schema, and has the same content, but the object itself is devoid of any type awareness, it's as the name implies simply a generic container of fields. The following example should make clear what I'm saying here:

Listing 3.19 Specific Avro records vs. GenericRecord

```
AvroAvenger avenger = // returned from consumer with
  //SPECIFIC_AVRO_READER_CONFIG=true
avenger.getName();
avenger.getRealName();     ❶
avenger.getMovies();

GenericRecord genericRecord = // returned from consumer with
  //SPECIFIC_AVRO_READER_CONFIG=false
if (genericRecord.hasField("name")) {
   genericRecord.get("name");
}

if (genericRecord.hasField("real_name")) {    ❷
    genericRecord.get("real_name");
}

if (GenericRecord.hasField("movies")) {
    genericRecord.get("movies");
}
```

❶ Accessing fields on the specific object

❷ Accessing fields on the generic object

From this simple code example, you can see the differences between the specific returned type vs. the generic. With the `AvroAvenger` object in annotation one, we can access the available properties directly, as the object is "aware" of its structure and provides methods for accessing those fields. But with the `GenericRecord` object you need to query if it contains a specific field before attempting to access it.

NOTE **The specific version of the Avro schema is not just a POJO (Plain Old Java Object) but extends the `SpecificRecordBase` class.**

Notice that with the `GenericRecord` you need to access the field exactly as its specified in the schema, while the specific version uses the more familiar camel case notation.

The difference between the two is that with the specific type you know the structure, but with the generic type, since it could represent any arbitrary type, you need to query for different fields to determine its structure. You need to work with a `GenericRecord` much like you would with a `HashMap`.

However you're not left to operate completely in the dark. You can get a list of fields from a `GenericRecord` by calling `GenericRecord.getSchema().getFields()`. Then you could iterate over the list of `Field` objects and get the names by calling the `Fields.name()`. Additionally you could get the name of the schema with `GenericRecord.getSchema().getFullName();` and presumably at that point you would know which fields the record contained.

Updating a field you'd follow a similar approach: .Updating or setting fields on specific and generic records

```
avnenger.setRealName("updated name")
genericRecord.put("real_name", "updated name")
```

So from this small example you can see that the specific object gives you the familiar setter functionality but the the generic version you need to explicitly declare the field you are updating. Again you'll notice the `HashMap` like behavior updating or setting a field with the generic version.

Protobuf provides a similar functionality for working with specific or arbitrary types. To work with an arbitrary type in Protobuf you'd us a `DynamicMessage`. As with the Avro `GenericRecord`, the `DynamicMessage` offers functions to discover the type and the fields. With JSON Schema the specific types are just the object generated from the gradle plugin, there's no framework code associated with it like Avro or Protobuf. The generic version is a type of `JsonNode` since the deserializer uses the jackson-databind (github.com/FasterXML/jackson-databind) API for serialization and deserialization.

NOTE **The source code for this chapter contain examples of working with the specific and generic types of Avro, Protobuf and JSON Schema.**

So the question is when do you use the specific type vs. the generic? In the case where you only have one type of record in a Kafka topic you'll use the specific version. On the other hand, if you have multiple event types in a topic, you'll want to use the generic version, as each consumed record could be a different type. We'll talk more about multiple event types in a single topic later in this chapter, and again in the client and Kafka Streams chapters.

The final thing to remember is that to use the specific record type, you need to set the `kafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG` to `true`. The default for the `SPECIFIC_AVRO_READER_CONFIG` is false, so the consumer returns the `GenericRecord` type if the configuration is not set.

Now with the sidebar about different record types completed, let's resume walking through your first end-to-end example using Schema Registry. You've already produced some records using the schema you uploaded previously. Now you just need to start a consumer to demonstrate deserializing those records with the schema. Again, looking at the log files should be instructive as you'll see the embedded deserializer downloading the schema for the first record only as it gets cached after the initial retrieval.

I should also note that the following example using `bbejeck.chapter_3.consumer.avro.AvroConsumer` uses both the specific class type and the `GenericRecord` type. As the example runs, the code prints out the type of the consumed record.

NOTE	There are similar examples for Protobuf and JSON Schema in the source code.

So let's run the consumer example now by executing the following command from the root of the book source code project:

Listing 3.20 Running the AvroConsumer

```
./gradlew streams:runAvroConsumer
```

IMPORTANT	Again, the same caveat here about running the command with the preceding `:` character, otherwise it will run the consumer for all modules and the example will not work.

The `AvroConsumer` prints out the consumed records and shuts down by itself. Congratulations, you've just serialized and deserialized records using Schema Registry!

So far we've covered the types of serialization frameworks supported by Schema Registry, how to write and add a schema file, and walked through a basic example using a schema. During the portion of the chapter where you uploaded a schema, I mentioned the term `subject` and how it defines the scope of schema evolution. That's what you'll learn in the next section, using the different subject name strategies.

3.2 Subject name strategies

Schema Registry uses the concept of a subject to control the scope of schema evolution. Another way to think of the subject is a namespace for a particular schema. In other words, as your business requirements evolve, you'll need to make changes to your schema files to make the appropriate changes to your domain objects. For example, with our AvroAvenger domain object, you want to remove the real (civilian) name of the hero and add a list of their powers.

Schema Registry uses the subject to lookup the existing schema and compare the changes with the new schema. It performs this check to make sure the changes are compatible with the current compatibility mode set. We'll talk about compatibility modes in an upcoming section. The subject name strategy determines the scope of where schema registry makes its compatibility checks.

There are three types of subject name strategies, `TopicNameStrategy`, `RecordNameStrategy`, and `TopicRecordNameStrategy`. You can probably infer the scope of the name-spacing implied by the strategy names, but it's worth going over the details. Let's dive in and discuss these different strategies now.

NOTE **By default all serializers will attempt to register a schema when serializing, if it doesn't find the corresponding id in its local cache. Auto registration is a great feature, but in some cases you may need to turn it off with a producer configuration setting of `auto.register.schemas=false`. One example of not wanting auto registration is when you are using an Avro union schema with references. We'll cover this in more detail later in the chapter.**

3.2.1 TopicNameStrategy

The `TopicNameStrategy` is the default subject in Schema Registry. The subject name comes from the name of the topic. You saw the `TopicNameStrategy` in action earlier in the chapter when you registered a schema with the gradle plugin. To be more precise the subject name is `topic-name-key` or `topic-name-value` as you can have different types for the key and value requiring different schemas.

The `TopicNameStrategy` ensures there is only one data type on a topic, since you can't register a schema for a different type with the same topic name. Having a single type per topic makes sense in a lot of cases. For example, if you name your topics based on the event type they store. it follows that they will contain only one record type.

Another advantage of the `TopicNameStrategy` is with the schema enforcement limited to a single topic, you can have another topic using the same record type, but using a different schema. Consider the situation where two different departments use the same record type, but use different topic names. With the `TopicNameStrategy` these departments can register completely different schemas for the same record type, since the scope of the schema is limited to a particular topic.

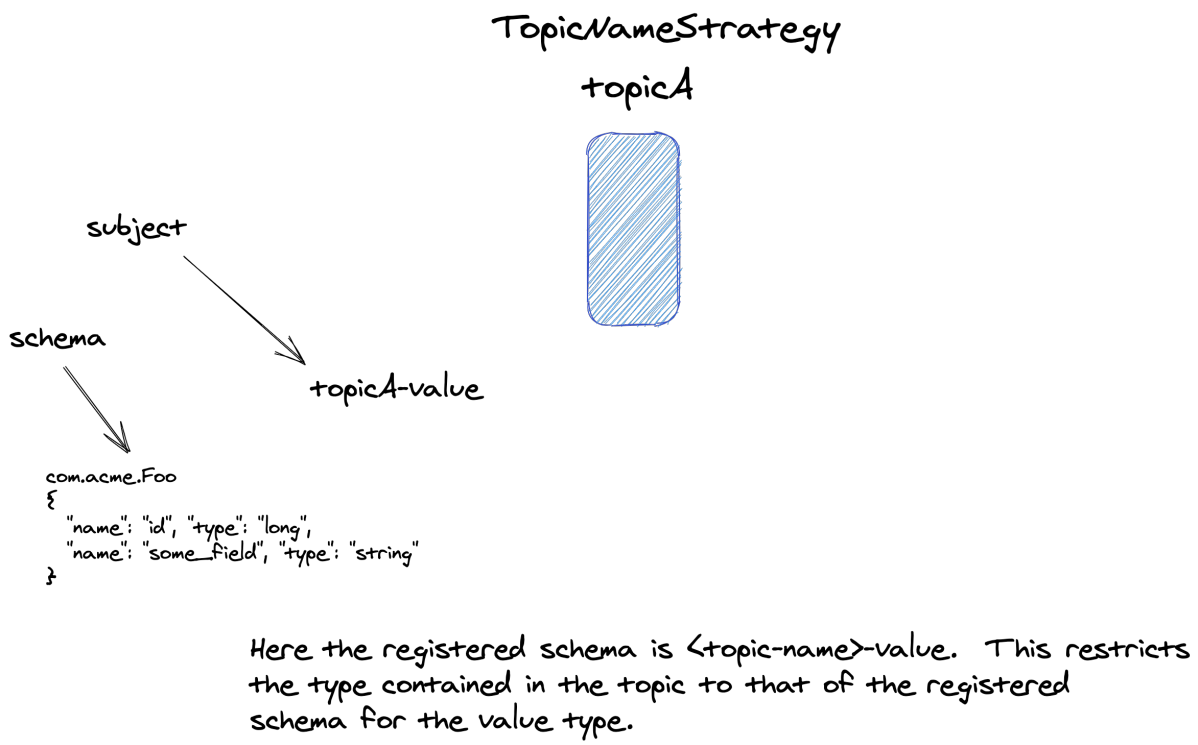

Figure 3.6 TopicNameStrategy enforces having the same type of domain object represented by the registered schema for the value and or the key

Since the `TopicNameStrategy` is the default, you don't need to specify any additional configurations. When you register schemas you'll use the format of `<topic>-value` as the subject for value schemas and `<topic>-key` as the subject for key schemas. In both cases you substitute the name of the topic for the `<topic>` token.

But there could be cases where you have closely related events and you want to produce those records into in one topic. In that case you'll want to chose a strategy that allows different types and schemas in a topic.

3.2.2 RecordNameStrategy

The `RecordNameStrategy` uses the fully qualified class name (of the Java object representation of the schema) as the subject name. By using the record name strategy you can now have multiple types of records in the same topic. But the key point is that there is a **logical** relationship between these records, it's just the physical layout of them is different.

When would you choose the `RecordNameStrategy`? Imagine you have different IoT (Internet of Things) sensors deployed. Some of sensors measure different events so they'll have different records. But you still want to have them co-located on the same topic.

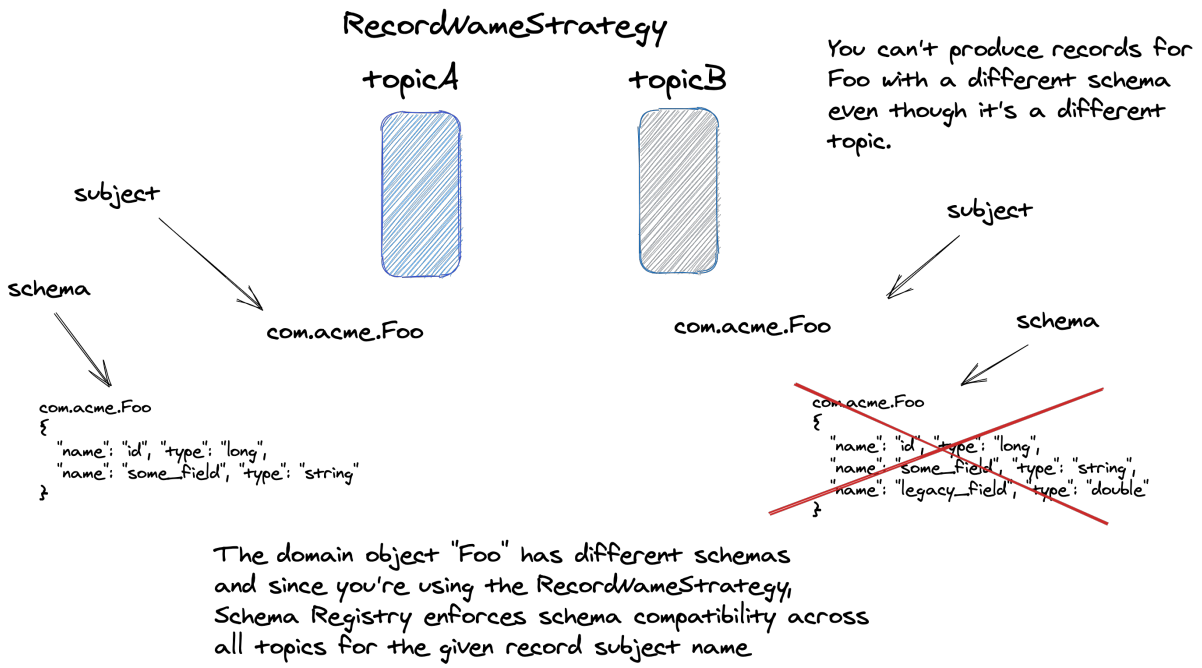

Figure 3.7 RecordNameStrategy enforces having the same schema for a domain object across different topics

Since there can be different types, the compatibility checks occur between schemas with the same record name. Additionally the compatibility check extends to all topics using a subject with the same record name.

To use the `RecordNameStrategy` you use a fully qualified class name for the subject when registering a schema for a given record type. For the `AvengerAvro` object we've used in our examples, you would configure the schema registration like this:

Listing 3.21 Schema Registry gradle plugin configuration for RecordNameStrategy

```
subject('bbejeck.chapter_3.avro.AvengerAvro','src/main/avro/avenger.avsc', 'AVRO')
```

Then you need to configure the producer and consumer with the appropriate subject name strategy. For example:

Listing 3.22 Producer configuration for RecordNameStrategy

```
Map<String, Object> producerConfig = new HashMap<>();
producerConfig.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
```

Listing 3.23 Consumer configuration for RecordNameStrategy

```
Map<String, Object> consumerConfig = new HashMap<>();
config.put(KafkaAvroDeserializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
config.put(KafkaAvroDeserializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 RecordNameStrategy.class);
```

NOTE **If you are only using Avro for serializing/deserializing the values, you don't need to add the configuration for the key. Also the key and value subject name strategies do not need to match, I've only presented them that way here.**

For Protobuf use the `KafkaProtobufSerializerConfig` and `KafkaProtobufDeserializerConfig` and for JSON schema use the `KafkaJsonSchemaSerializerConfig` and `KafkaJsonSchemaDeserializerConfig`

These configurations only effect how the serializer/deserializer interact with Schema Registry for looking up schemas. Again the serialization is decoupled from producing and consuming process.

One thing to consider is that by using only the record name, all topics must use the same schema. If you want to use different records in a topic, but want to only consider the schemas for that particular topic, then you'll need to use another strategy.

3.2.3 TopicRecordNameStrategy

As you can probably infer from the name this strategy allows for having multiple record types within a topic as well. But the registered schemas for a given record are only considered within the scope of the current topic. Let's take a look at the following illustration to get a better idea of what this means.

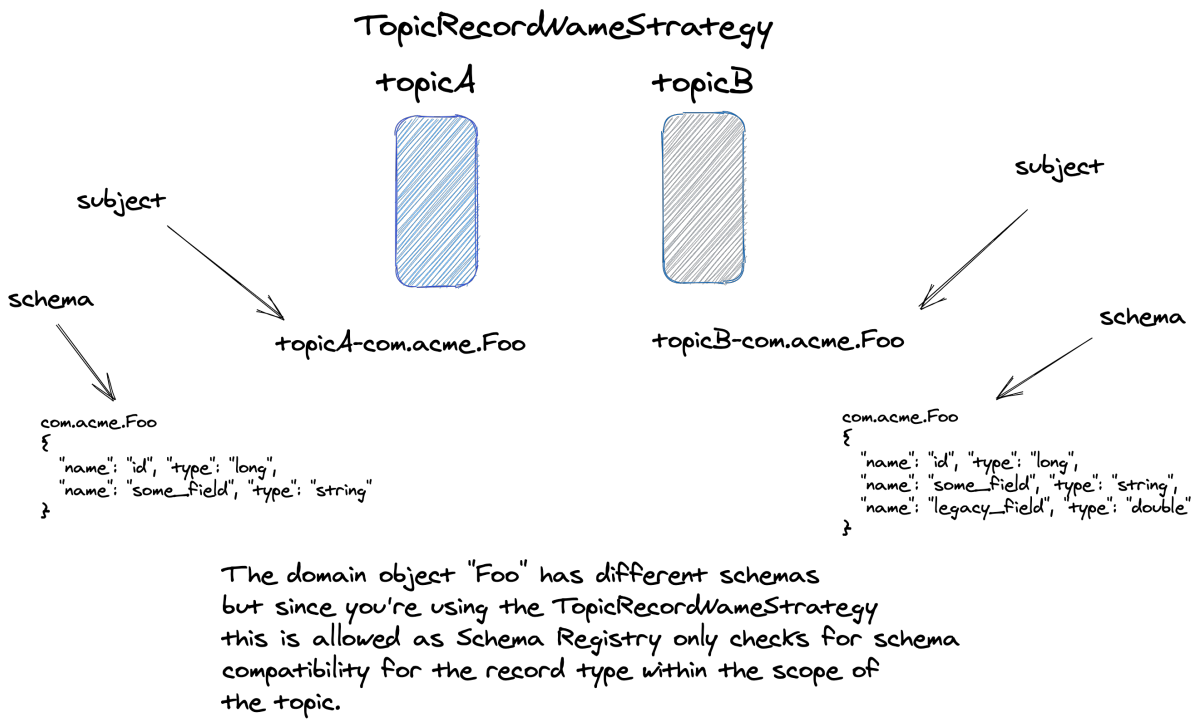

Figure 3.8 TopicRecordNameStrategy allows for having different schemas for the same domain object across different topics

As you can see from the image above `topic-A` can have a different schema for the record type `Foo` from `topic-B`. This strategy allows you to have multiple logically related types on one topic, but it's isolated from other topics where you have the same type but are using different schemas.

Why would you use the `TopicRecordNameStrategy`? For example, consider this situation:

You have one version of the `CustomerPurchaseEvent` event object in the `interactions` topic, that groups all customer event types (`CustomerSearchEvent`, `CustomerLoginEvent` etc) grouped together. But you have an older topic purchases, that also contains `CustomerPurchaseEvent` objects, but it's for a legacy system so the schema is older and contains different fields from the newer one. The `TopicRecordNameStrategy` allows for having these two topics to contain the same ***type*** but with different schema versions.

Similar to the `RecordNameStrategy` you'll need to do the following steps to configure the strategy:

Listing 3.24 Schema Registry gradle plugin configuration for TopicRecordNameStrategy

```
subject('avro-avengers-bbejeck.chapter_3.avro.AvengerAvro',
  'src/main/avro/avenger.avsc', 'AVRO')
```

Then you need to configure the producer and consumer with the appropriate subject name strategy. For example:

Listing 3.25 Producer configuration for TopicRecordNameStrategy

```
Map<String, Object> producerConfig = new HashMap<>();
producerConfig.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
producerConfig.put(KafkaAvroSerializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
```

Listing 3.26 Consumer configuration for TopicRecordNameStrategy

```
Map<String, Object> consumerConfig = new HashMap<>();
config.put(KafkaAvroDeserializerConfig.KEY_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
config.put(KafkaAvroDeserializerConfig.VALUE_SUBJECT_NAME_STRATEGY,
 TopicRecordNameStrategy.class);
```

NOTE **The same caveat about registering the strategy for the key applies here as well, you would only do so if you are using a schema for the key, it's only provided here for completeness. Also the key and value subject name strategies don't need to match**

Why would you use the `TopicRecordNameStrategy` over either the `TopicNameStrategy` or the `RecordNameStrategy`? If you wanted the ability to have multiple event types in a topic, but you need the flexibility to have different schema versions for a given type across your topics.

But when considering multiple types in a topic, both the `TopicRecordNameStrategy` and the `RecordNameStrategy` don't have the ability to constrain a topic to fixed set of types. Using either of those subject name strategies opens up the topic to have an unbounded number of different types. We'll cover how to improve on this situation when we cover schema references in an upcoming section.

Here's a quick summary for you to consider when thinking of the different subject name strategies. Think of the subject name strategy as a function that accepts the topic-name and record-schema as arguments and it returns a subject-name. The `TopicNameStrategy` only uses the topic-name and ignores the record-schema. `RecordNameStrategy` does the opposite; it ignores the topic-name and only uses the record-schema. But the `TopicRecordNameStrategy` uses both of them for the subject-name.

Table 3.1 Schema strategies summary table

Strategy	Multiple types in a topic	Different versions of objects across topics
TopicNameStrategy	Maybe	Yes
RecordNameStrategy	Yes	No
TopicRecordNameStrategy	Yes	Yes

So far we've covered the subject naming strategies and how Schema Registry uses subjects for

name-spacing schemas. But there's another dimension to schema management, how to evolve changes within the schema itself. How do you handle changes like the removal or addition of a field? Do you want your clients to have forward or backward compatibility? In the next section we'll cover exactly how you handle schema compatibility.

3.3 Schema compatibility

When there are schema changes you need to consider the compatibility with the existing schema and the producer and consumer clients. If you make a change by removing a field how does this impact the producer serializing the records or the consumer deserializing this new format?

To handle these compatibility concerns, Schema Registry provides four base compatibility modes `BACKWARD`, `FORWARD`, `FULL`, and `NONE`. There are also three additional compatibility modes `BACKWARD_TRANSITIVE`, `FORWARD_TRANSITIVE`, and `FULL_TRANSITIVE` which extend on the base compatibility mode with the same name. The base compatibility modes only guarantee that a new schema is compatible with immediate previous version. The transitive compatibility specifies that the new schema is compatible with **all** previous versions of a given schema applying the compatibility mode.

You can specify a global compatibility level or a compatibility level per subject.

What follows in this chapter is a description of the valid changes for a given compatibility mode along with an illustration demonstrating the sequence of changes you'd need to make to the producers the consumers. For a hands on tutorial of making changes to a schema, see Appendix-B: Schema Compatibility Workshop.

3.3.1 Backward compatibility

Backward compatibility is the default migration setting. With backward compatibility you update the consumer code first to support the new schema. The updated consumers can read records serialized with the new schema or the immediate previous schema.

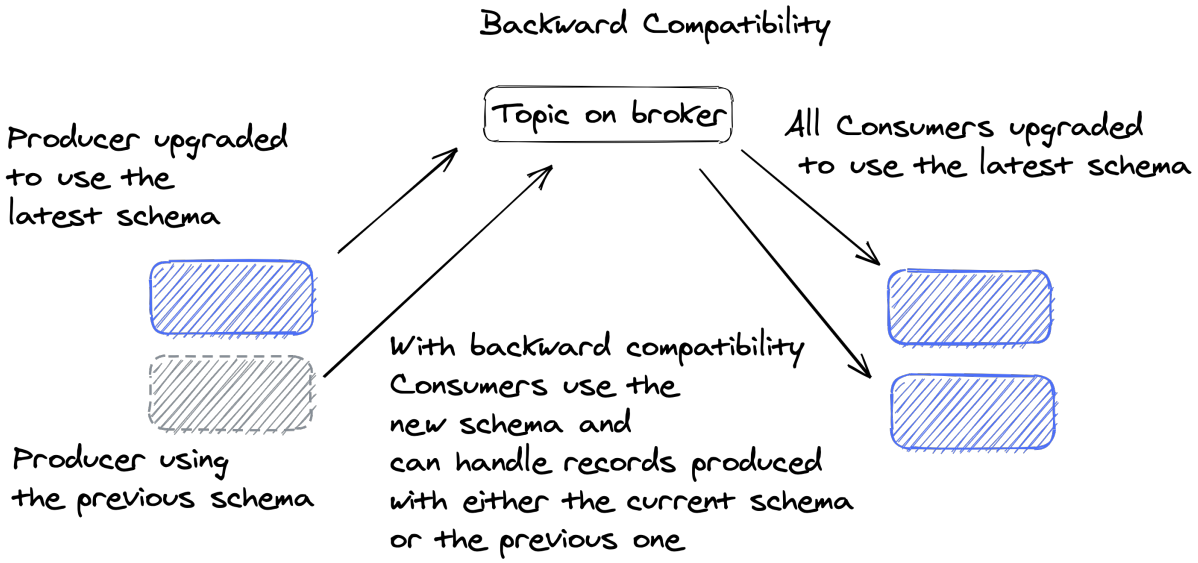

Figure 3.9 Backward compatibility updates consumers first to use the new schema then they can handle records from producers using either the new schema or the previous one

As shown in this illustration the consumer, can work with both the previous and the new schemas. The allowed changes with backwards compatibility are deleting fields or adding optional fields. An field is considered optional when the schema provides a default value. If the serialized bytes don't contain the optional field, then the deserializer uses the specified default value when deserializing the bytes back into an object.

3.3.2 Forward compatibility

Forward compatibility is a mirror image of backward compatibility regarding field changes. With forward compatibility you can add fields and delete **optional** fields.

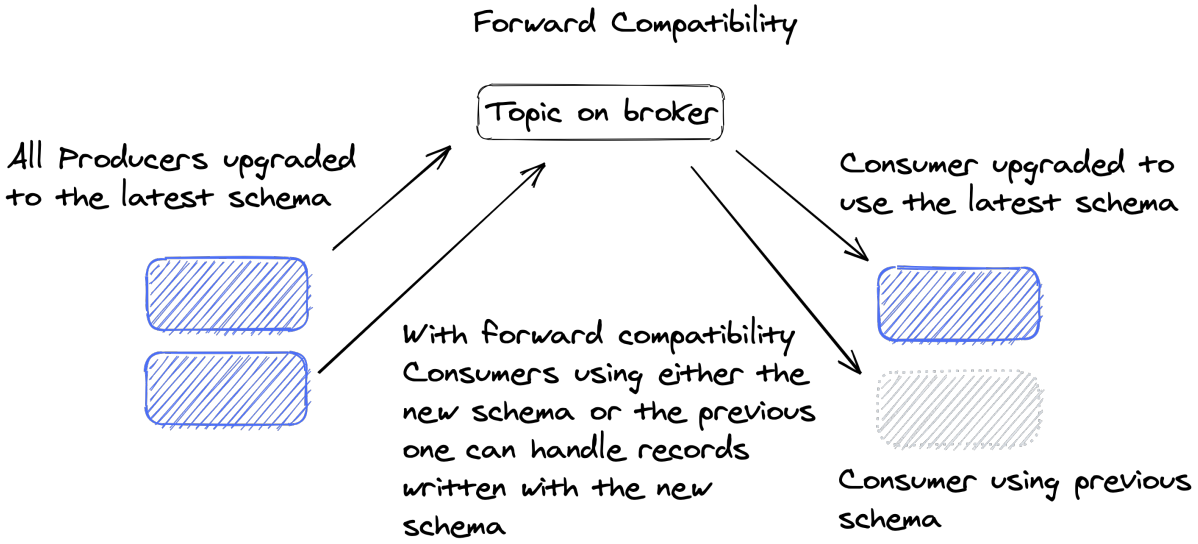

Figure 3.10 Forward compatibility updates producers first to use the new schema and consumers can handle the records either the new schema or the previous one

By upgrading the producer code first, you're making sure the new fields are properly populated and only records in the new format are available. Consumers you haven't upgraded can still work with the new schema as it will simply ignore the new fields and the deleted fields have default values.

At this point you've seen two compatibility types, backward and forward. As the compatibility name implies, you must consider record changes in one direction. In backward compatibility, you updated the consumers first as records could arrive in either the new or old format. In forward compatibility, you updated the producers first to ensure the records from that point in time are only in the new format. The last compatibility strategy to explore is the `FULL` compatibility mode.

3.3.3 Full compatibility

In full compatibility mode, you free to add or remove fields, but there is one catch. **Any changes** you make must be to **optional** fields only. To recap an optional field is one where you provide a default value in the schema definition should the original deserialized record not provide that specific field.

> **NOTE** **Both Avro and JSON Schema provide support for explicitly providing default values, with Protocol Buffers version 3 (the version used in the book) every field automatically has a default based in its type. For example number types are 0, strings are "", collections are empty etc.**

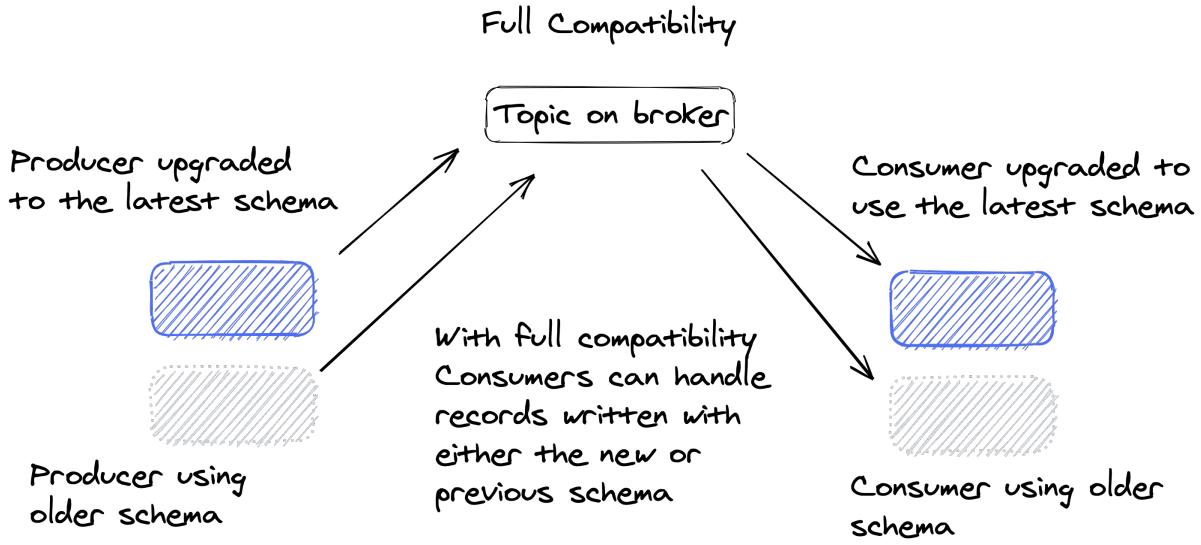

Figure 3.11 Full compatibility allows for producers to send with the previous or new schema and consumers can handle the records either the new schema or the previous one

Since the fields involved in the updated schema are optional, these changes are considered compatible for existing producer and consumer clients. This means that the upgrade order in this

case is up to you. Consumers will continue to work with records produced with the new or old schema.

3.3.4 No compatibility

Specifying a compatibility of `NONE` instructs Schema Registry to do just that, no compatibility checks. By not using any compatibility checks means that someone can add new fields, remove existing fields, or change the type of a field. Any and all changes are accepted.

Not providing any compatibility checks provides a great deal of freedom. But the trade-off is you're vulnerable to breaking changes that might go undetected until the worse possible time; in production.

It could be that every time you update a schema, you upgrade all producers and consumers at the same time. Another possibility is to create a new topic for clients to use. Applications can use the new topic without having the concerns of it containing records from the older, incompatible schema.

Now you've learned how you can migrate a schema to use a new version with changes within the different schema compatibility modes and for review here's a quick summary table of the different compatibility types

Table 3.2 Schema Compatibility Mode Summary

Mode	Changes Allowed	Client Update Order	Retro guaranteed compatibility
Backward	Delete fields, add optional fields	Consumers, Producers	Prior version
Backward Transitive	Delete fields, add optional fields	Consumers, Producers	All previous versions
Forward	Add fields, delete optional fields	Producers, Consumers	Prior version
Forward Transitive	Add fields, delete optional fields	Producers, Consumers	All previous versions
Full	Delete optional fields, add optional fields	Doesn't matter	Prior version
Full Transitive	Delete optional fields, add optional fields	Doesn't matter	All previous versions

But there's more you can do with schemas. Much like working with objects you can share common code to reduce duplication and make maintenance easier, you can do the same with schema references

3.4 Schema references

A schema reference is just what is sounds like, referring to another schema from inside the current schema. Reuse is a core principal in software engineering as the ability to leverage something you've already built solves two issues. First, you could potentially save time by not having to re-write some exiting code. Second, when you need to update the original work (which always happens) all the downstream components leveraging the original get automatically updated as well.

When would you want to use a schema reference? Let's consider you have an application providing information on commercial business and universities. To model the business you have a `Company` schema and for the universities you have a `College` schema. Now a company has executives and the college has professors. You want to represent both with a nested schema of a `Person` domain object. The schemas would look something like this:

Listing 3.27 College schema

```
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CollegeAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "professors", "type":
    {"type": "array", "items": {   ❶
      "namespace": "bbejeck.chapter_3.avro",
      "name":"PersonAvro",         ❷
      "fields": [
        {"name": "name", "type":"string"},
        {"name": "address", "type": "string"},
        {"name": "age", "type": "int"}
      ]
    }},
      "default": []
    }
  ]
}
```

❶ Array of professors

❷ The item type in array is a Person object

So you can see here you have a nested record type in your college schema, which is not uncommon. Now let's look at the company schema

Listing 3.28 Company schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CompanyAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "executives", "type":
    {"type": "array", "items": {    ❶
      "type":"record",
      "namespace": "bbejeck.chapter_3.avro",
      "name":"PersonAvro",          ❷
      "fields": [
        {"name": "name", "type":"string"},
        {"name": "address", "type": "string"},
        {"name": "age", "type": "int"}
      ]
    }},
      "default": []
    }
  ]
}
```

❶ Array of executives

❷ Item type is a PersonAvro

Again you have a nested record for the type contained in the schema array. It's natural to model the executive or professor type as a person, as it allows you to encapsulate all the details into an object. But as you can see here, there's duplication in your schemas. If you need to change the person schema you need to update every file containing the nested person definition. Additionally, as you start to add more definitions, the size and complexity of the schemas can get unwieldy quickly due to all the nesting of types.

It would be better to put a reference to the type when defining the array. So let's do that next. We'll put the nested `PersonAvro` record in its own schema file, person.avsc.

I won't show the file here, as nothing changes, we are putting the definition you see here in a separate file. Now let's take a look at how you'd update the `college.avsc` and `company.avsc` schema files:

Listing 3.29 Updated College schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CollegeAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "professors", "type":
    {"type": "array", "items": "bbejeck.chapter_3.avro.PersonAvro"},   ❶
      "default": []
    }
  ]
}
```

❶ This is the new part it's a reference to the person object

IMPORTANT **When using schema references, the referring schema you provide must be the same type. For example you can't provide a reference to an Avro schema or JSON Schema inside Protocol Buffers schema, the reference must be another Protocol Buffers schema.**

Here you've cleaned things up by using a reference to the object created by the `person.avsc` schema. Now let's look at the updated company schema as well

Listing 3.30 Updated Company schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "CompanyAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "executives", "type":
      {
        "type": "array", "items": "bbejeck.chapter_3.avro.PersonAvro"},   ❶
        "default": []
       }
  ]
}
```

❶ This is the new part also it's a reference to the person object

Now both schemas refer to the same object created by the person schema file. For completeness let's take a look at how you implement a schema reference in both JSON Schema and Protocol Buffers. First we'll look at the JSON Schema version:

Listing 3.31 Company schema reference in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Exchange",
  "description": "A JSON schema of a Company using Person refs",
  "javaType": "bbejeck.chapter_3.json.CompanyJson",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "executives": {
      "type": "array",
      "items": {
        "$ref": "person.json"   ❶
      }
    }
  }
}
```

❶ The reference to the Person object schema

The concept with references in JSON Schema is the same, but you provide an explicit `$ref` element pointing to the referenced schema file. It's assumed that the referenced file is located in the same directory as the referring schema.

Now let's take a look at the equivalent reference with Protocol Buffers:

Listing 3.32 Company Schema reference in Protocol Buffers

```
syntax = "proto3";

package bbejeck.chapter_3.proto;

import "person.proto";    ❶

option java_outer_classname = "CompanyProto";

message Company {
  string name = 1;
  repeated Person executives = 2;   ❷
}
```

❶ Import statement for the referenced schema

❷ Referring to the Person proto

With Protocol Buffers you have a very minor extra step of providing an import referring the the proto file containing the referenced object.

But now the question is how will the (de)serializers know how to serialize and deserialize the object into the correct format? You've removed the definition from inside the file, so you need to get a reference to the schema as well. Fortunately, Schema Registry provides for using schema references.

What you need to do is register a schema for the person object first, then when you register the schema for the college and company schemas, you provide a reference to the already registered person schema.

Using the gradle schema-registry plugin makes this a simple task. Here's how you would configure it for using schema references:

Listing 3.33 Gradle plugin reference configuration

```
register {

    subject('person','src/main/avro/person.avsc', 'AVRO')         ❶
    subject('college-value','src/main/avro/college.avsc', 'AVRO')
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "person", 1)   ❷
    subject('company-value','src/main/avro/company.avsc', 'AVRO')
        .addReference("bbejeck.chapter_3.avro.PersonAvro", "person", 1)   ❸
    }
```

❶ Register the person schema

❷ Register the college schema and add a reference to the person schema

❸ Register the company schema and add a reference to the person schema

So you first registered the `person.avsc` file, but in this case the subject is simply `person` because in this case it's not associated directly with any one topic. Then you registered both the college and company schemas using the `<topic name> - value` pattern as the college and company schemas are tied to topics with the same names and use the default subject name strategy (TopicNameStrategy) . The `addReference` method takes three parameters:

1. A name for the reference. Since you're using Avro it's the fully qualified name of the schema. For Protobuf it's the name of the proto file and for JSON schema it's the URL in the schema.
2. The subject name for the registered schema.
3. The version number to use for the reference.

Now with the references in place, you register the schemas and your producer and consumer client will be able to properly serialize and deserialize the objects with the references.

There are examples in the source code for running a producer and consumer with the schema references in action. Since you've already run the `./gradlew streams:registerSchemasTask` for the main module, you've already set up your references. To see using schema references in action you can run the following:

Listing 3.34 Tasks for schema references in action

```
./gradlew streams:runCompanyProducer
./gradlew streams:runCompanyConsumer

./gradlew streams:runCollegeProducer
./gradlew streams:runCollegeConsumer
```

3.5 Schema references and multiple events per topic

We've covered the different subject strategies `RecordNameStrategy`, and `TopicRecordNameStrategy` and how they allow for producing records of different types to a topic. But with the `RecordNameStrategy` any topic you produce to must use the same schema version for the given type. This means that if you want to make changes or evolve the schema, all topics must use the new schema. Using the `TopicRecordNameStrategy` allows for multiple events in a topic and it scopes the schema to a single topic, allowing you to evolve the schema independent of other topics.

But with both approaches you can't control the number of different types produced to the topic. If someone wants to produce a record of a different type that is not wanted, you don't have any way to enforce this policy.

However there is a way to achieve producing multiple event types to a topic ***and*** restrict the types of records produced to the topic by using schema references. By using `TopicNameStrategy` in conjunction with schema references, it allows all records in the topic to be constrained by a single subject. In other words, schema references allow you to have multiple types, but only those types that the schema refers to. This is best understood by walking through an example scenario

Imagine you are an online retailer and you've developed system for precise tracking of packages you ship to customers. You have a fleet of trucks and planes that take packages anywhere in the country. Each time a package handled along its route its scanned into your system generating one of three possible events represented by these domain objects: - `PlaneEvent`, `TruckEvent`, or a `DeliveryEvent`.

These are distinct events, but they are closely related. Also since the order of these events is important, you want them produced to the same topic so you have all related events together and in the proper sequence of their occurrence. I'll cover more about how combining related events in a single topic helps with sequencing in chapter 4 when we cover clients. Now assuming you've already created schemas for the PlaneEvent, TruckEvent, and the DeliveryEvent you could create an schema like this to contain the different event types:

Listing 3.35 Avro schema all_events.avsc with multiple events

```
[
  "bbejeck.chapter_3.avro.TruckEvent",      ❶
  "bbejeck.chapter_3.avro.PlaneEvent",
  "bbejeck.chapter_3.avro.DeliveryEvent"
]
```

❶ An Avro union type for the different events

The `all_events.avsc` schema file is an Avro `union`, which is an array of the possible event types. You use a `union` when a field, or, in this case a schema, could be of more then one type.

Since you're defining all the expected types in a single schema, your topic can now contain multiple types, but it's limited to only those listed in the schema. When using schema references in this format with Avro, it's critical to always set `auto.register.schemas=false` and `use.latest.version=true` in you Kafka producer configuration. Here's the reason why you need to use these configurations with the given settings.

When the Avro serializer goes to serialize the object, it won't find the schema for it, since it's in the union schema. As a result it will register the schema of the individual object, overwriting the union schema. So setting the auto registration of schemas to `false` avoids the overwriting of the schema problem. In addition, by specifying `use.latest.version=true`, the serializer will

retrieve the latest version of the schema (the union schema) and use that for serialization. Otherwise it would look for the event type in the subject name, and since it won't find it, a failure will result.

> **TIP** **When using the `oneOf` field with references in Protocol Buffers, the referenced schemas are automatically registered recursively, so can go ahead and use the `auto.register.schemas` configuration set to `true`. You can also do the same with JSON Schema `oneOf` fields.**

Let's now take a look at how you'd register the schema with references:

Listing 3.36 Register the all_events schema with references

```
subject('truck_event','src/main/avro/truck_event.avsc', 'AVRO')    ❶
subject('plane_event','src/main/avro/plane_event.avsc', 'AVRO')
subject('delivery_event','src/main/avro/delivery_event.avsc', 'AVRO')

subject('inventory-events-value', 'src/main/avro/all_events.avsc','AVRO')   ❷
    .addReference("bbejeck.chapter_3.avro.TruckEvent", "truck_event", 1)   ❸
    .addReference("bbejeck.chapter_3.avro.PlaneEvent", "plane_event", 1)
    .addReference("bbejeck.chapter_3.avro.DeliveryEvent", "delivery_event", 1)
```

❶ Registering the individual schemas referenced in the all_events.avsc file

❷ Registering all_events schema

❸ Adding the references of the individual schemas

As you saw before in the schema references section, with Avro you need to register the individual schemas before the schema containing the references. After that you can register the main schema with the references to the individual schemas.

When working with Protobuf there isn't a `union` type but there is a `oneOf` which is essentially the same thing. However with Protobuf you can't have a `oneOf` at the top-level, it must exist in an Protobuf message. For your Protobuf example, consider that you want to track the following customer interactions logins, searches, and purchases as separate events. But since they are closely related and sequencing is important you want them in the same topic. Here's the Protobuf file containing the references:

Listing 3.37 Protobuf file with references

```
syntax = "proto3";

package bbejeck.chapter_3.proto;

import "purchase_event.proto";    ❶
import "login_event.proto";
import "search_event.proto";

option java_outer_classname = "EventsProto";

message Events {

  oneof type {    ❷
    PurchaseEvent purchase_event = 1;
    LoginEvent login_event = 2;
    SearchEvent search_event = 3;
  }
  string key = 4;
}
```

❶ Importing the individual Protobuf messages

❷ The oneOf field which could be one of the three types listed

You've seen a Protobuf schema earlier in the chapter so I won't go over all the parts here, but the key thing for this example is the `oneOf` field `type` which could be a `PurchaseEvent`, `LoginEvent`, or a `SearchEvent`. When when you register a Protobuf schema it has enough information present to recursively register all of the referenced schemas, so it's safe to set the `auto.register` configuration to `true`.

You can structure your Avro references in a similar manner:

Listing 3.38 Avro schema with references using an outer class

```
{
  "type": "record",
  "namespace": "bbejeck.chapter_3.avro",
  "name": "TransportationEvent",    ❶

  "fields" : [
    {"name": "event", "type"[     ❷
      "bbejeck.chapter_3.avro.TruckEvent",    ❸
      "bbejeck.chapter_3.avro.PlaneEvent",
      "bbejeck.chapter_3.avro.DeliveryEvent"
    ]}
  ]
}
```

❶ Outer class name

❷ Field named "event"

❸ Avro union for the field type

So the main difference with this Avro schema vs. the previous Avro schema with references is

this one has outer class and the references are now a field in the class. Also, when you provide an outer class with Avro references like you have done here, you can now set the `auto.register` configuration to `true`, although you still need to register the schemas for the referenced objects ahead of time as Avro, unlike Protobuf, does not have enough information to recursively register the referenced objects.

There are some additional considerations when it comes to using multiple types with producers and consumers. I'm referring to the generics you use on the Java clients and how you can determine to take the appropriate action on an object depending on its concrete class name. I think these topics are better suited to discuss when we cover clients, so we'll cover that subject in the next chapter.

At this point, you've learned about the different schema compatibility strategies, how to work with schemas and using references. In all the examples you've run you've been using the built in serializers and deserializers provided by Schema Registry. In the next section we'll cover the configuration for the (de)serializers for producers and consumers. But we'll only cover the configurations related to the (de)serializers and not general producer and consumer configuration, those we'll cover in the next chapter.

3.6 Schema Registry (de)serializers

I've covered in the beginnings of the chapter,that when producing records to Kafka you need to serialize the records for transport over the network and storage in Kafka. Conversely, when consuming records you deserialize them so you can work with objects.

You need to configure the producer and consumer with the classes required for the serialization and deserialization process. Schema Registry provides a serializer, deserializer, and a Serde (used in Kafka Streams) for all three (Avro, Protobuf, JSON) supported types.

Providing the serialization tools is a strong argument for using Schema Registry that I spoke about earlier in the chapter. Freeing developers from having to write their own serialization code speeds up development and increases standardization across an organization. Also using a standard set of serialization tools reduces errors as reduces the chance that one team implements their own serialization framework.

NOTE **What's a Serde? A Serde is a class containing both a serializer and deserializer for a given type. You will use Serdes when working with Kafka Streams because you don't have access to the embedded producer and consumer so it makes sense to provide a class containing both and Kafka Streams uses the correct serializer and deserializer accordingly. You'll see Serdes in action when we start working with Kafka Streams in a later chapter.**

In the following sections I'm going to discuss the configuration for using Schema Registry aware serializers, deserializers. One important thing to remember is you don't configure the serializers directly. You set the configuration for serializers when you configure either the `KafkaProducer` or `KafkaConsumer`. If following sections aren't entirely clear to you, that's OK because we'll cover clients (producers and consumer) in the next chapter.

3.6.1 Avro

For Avro records there is the `KafkaAvroSerializer` and `KafkaAvroDeserializer` classes for serializing and deserializing records. When configuring a consumer, you'll need to include an additional property, `KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG=true` indicating that you want the deserializer to create a `SpecificRecord` instance. Otherwise the deserializer returns a `GenericRecord`.

Let's take a look at snippets of how you add these properties to both the producer and consumer. Note the following example only shows the configurations required for the serialization. I've left out the other configurations for clarity. We'll cover configuration of producers and consumers in chapter 4.

Listing 3.39 Required configuration for Avro

```
// producer properties
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);  ❶
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaAvroSerializer.class);  ❷
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❸

//consumer properties these are set separately on the consumer
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class);  ❹
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaAvroDeserializer.class);  ❺
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG,
  true);  ❻
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❼
```

❶ The serializer for the key

❷ The serializer for the value

❸ Setting the URL for the serializer

❹ The deserializer for the key

❺ The deserializer for the value

❻ Indicating to construct a specific record instance

❼ Setting the URL for the deserializer

Next, let's take a look at the configuration for working with Protobuf records

3.6.2 Protobuf

For working with Protobuf records there are the `KafkaProtobufSerializer` and `KafkaProtobufDeserializer` classes.

When using Protobuf with schema registry, it's probably a good idea to specify both the `java_outer_classname` and set `java_multiple_files` to `true` in the protobuf schema. If you end up using the `RecordNameStrategy` with protobuf then you ***must*** use these properties so the deserializer can determine the type when creating an instance from the serialized bytes.

If you remember from earlier in the chapter we discussed that when using Schema Registry aware serializers, those serializers will attempt to register a new schema. If your protobuf schema references other schemas via imports, the referenced schemas are registered as well. Only protobuf provides this capability, when using Avro or JSON referenced schemas are not loaded automatically.

Again if you don't want auto registration of schemas, you can disable it with the following configuration `auto.shema.registration = false`.

Let's look at a similar example of providing the relevant Schema Registry configurations for working with protobuf records.

Listing 3.40 Required configuration for Protobuf

```
// producer properties
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);  ❶
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaProtobufSerializer.class);  ❷
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❸

// consumer properties again set separately on the consumer
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class);  ❹
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaProtobufDeserializer.class);  ❺
props.put(KafkaProtobufDeserializerConfig.SPECIFIC_PROTOBUF_VALUE_TYPE,
  AvengerSimpleProtos.AvengerSimple.class);  ❻
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");  ❼
```

❶ The key serializer

❷ The protobuf value serializer

❸ Providing the URL for Schema Registry for the consumer

❹ The key deserializer

❺ The protobuf value deserializer

❻ The specific class the deserializer should instantiate

❼ The location of Schema Registry for the producer

As with the Avro deserializer, you need to instruct it to create a specific instance. But in this case you configure the actual class name instead of setting a boolean flag indicating you want a specific class. If you leave out the specific value type configuration the deserializer returns a type of `DynamicRecord`. We covered working with the `DynamicRecord` in the protobuf schema section.

The `bbejeck.chapter_3.ProtobufProduceConsumeExample` class in the book source code demonstrates the producing and consuming a protobuf record.

Now we'll move on the final example of configuration of Schema Registry's supported types, JSON schemas.

3.6.3 JSON Schema

Schema Registry provides the `KafkaJsonSchemaSerializer` and `KafkaJsonSchemaDeserializer` for working with JSON schema objects. The configuration should feel familiar to both Avro and the Protobuf configurations.

NOTE **Schema Registry also provides `KafkaJsonSerializer` and `KafkaJsonDeserializer` classes. While the names are very similar these (de)serializers are meant for working with Java objects for conversion to and from JSON, without a JSON Schema. While the names are close, make sure you are using the serializer and deserializer with `Schema` in the name. We'll talks about the generic JSON serializers in the next section.**

Listing 3.41 Required configuration for JSON Schema

```
// producer configuration
producerProps.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");   ❶
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
  StringSerializer.class);   ❷
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
  KafkaJsonSchemaSerializer.class);   ❸

// consumer configuration
props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  "http://localhost:8081");   ❹
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
  StringDeserializer.class);   ❺
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
  KafkaJsonSchemaDeserializer.class);   ❻
props.put(KafkaJsonDeserializerConfig.JSON_VALUE_TYPE,
  SimpleAvengerJson.class);   ❼
```

❶ Providing the URL for Schema Registry for the producer

❷ The key serializer

❸ The JSON Schema value serializer

❹ Providing the URL for Schema Registry for the producer

❺ The key deserializer

❻ Specifying the JSON Schema value deserializer

❼ Configuring the specific classes this deserializer will create

Here you can see a similarity with the protobuf configuration in that you need to specify the class the deserializer should construct from the serialized form in annotation number 7 in this example. If you leave out the specify value type then the deserializer returns a `Map`, the generic form of a JSON schema deserialization. Just a quick note the same applies for keys. If your key is a JSON schema object, then you'll need to supply a `KafkaJsonDeserializerConfig.JSON_KEY_TYPE` configuration for the deserializer to create the exact class.

There is a simple producer and consumer example for working with JSON schema objects in the `bbejeck.chapter_3.JsonSchemaProduceConsumeExample` in the source code for the book. As with the other basic producer and consumer examples, there are sections demonstrating how to work with the specific and generic return types. We outlined the structure of the JSON schema generic type in the JSON schema section of this chapter.

Now we've covered the different serializer and deserializer for each type of serialization supported by Schema Registry. Although using Schema Registry is recommended, it's not required. In the next section we'll outline how you can serialize and deserialize your Java objects without Schema Registry.

3.7 Serialization without Schema Registry

In the beginning of this chapter, I stated that your event objects, or more specifically their schema representations, are a contract between the producers and consumers of the Kafka event streaming platform. Schema Registry provides a central repository for those schemas hence providing enforcement of this schema contracts across your organization. Additionally, the Schema Registry provided serializers and deserializers provide a convenient way of working with data without having to write your own serialization code.

Does this mean using Schema Registry is required? No not at all. In some cases, you may not have access to Schema Registry or don't want to use it. Writing your own custom serializers and deserializers isn't hard. Remember, producers and consumers are decoupled from the (de)serializer implementation, you only provide the classname as a configuration setting. Although it's good to keep in mind that by using Schema Registry you can use the same schemas across Kafka Streams, Connect and ksqlDB.

So to create your own serializer and deserializer you create classes that implement the

`org.apache.kafka.common.serialization.Serializer` and `org.apache.kafka.common.serialization.Deserializer` interfaces. With the `Serializer` interface there is only one method you ***must*** implement `serialize`. For the `Deserializer` it's the `deserialize` method. Both interfaces have additional default methods (`configure`, `close`) you can override if you need to.

Here's a section of a custom serializer using the `jackson-databind` `ObjectMapper`:

Listing 3.42 Serialize method of a custom serializer

```
// details left out for clarity
@Override
public byte[] serialize(String topic, T data) {
    if (data == null) {
        return null;
    }
    try {
        return objectMapper.writeValueAsBytes(data);   ❶
    } catch (JsonProcessingException e) {
        throw new SerializationException(e);
    }
}
```

❶ Converting the given object to a byte array

Here you call `objectMapper.writeValueAsBytes()` and it returns a serialized representation of the passed in object.

Now let's look at an example for the deserializing counterpart:

Listing 3.43 Deserialize method of a custom deserializer

```
// details left out for clarity
@Override
public T deserialize(String topic, byte[] data) {
    try {
        return objectMapper.readValue(data, objectClass);   ❶
    } catch (IOException e) {
        throw new SerializationException(e);
    }
}
```

❶ Converting the bytes back to an object specified by the objectClass parameter

The `bbejeck.serializers` package contains the serializers and deserializers shown here and additional ones for Protobuf. You can use these serializers/deserializers in any of the examples presented in this book but remember that they don't use Schema Registry. Or they can serve as examples of how to implement your own (de)serializers.

In this chapter, we've covered how event objects or more specifically, their schemas, represent contract between producers and consumers. We discussed how Schema Registry stores these schemas and enforces this implied contract across the Kafka platform. Finally we covered the

supported serialization formats of Avro, Protobuf and JSON. In the next chapter, you'll move up even further in the event streaming platform to learn about Kafka clients, the `KafkaProducer` and `KafkaConsumer`. If you think of Kafka as your central nervous system for data, then the clients are the the sensory inputs and outputs for it.

3.8 Summary

- Schemas represent a contract between producers and consumers. Even if you don't use explicit schemas, you have an implied one with your domain objects, so developing a way to enforce this contract between producers and consumers is critical.
- Schema Registry stores all your schemas enforcing data governance and it provides versioning and three different schema compatibility strategies - backward, forward and full. The compatibility strategies provide assurance that the new schema will work with it's immediate predecessor, but not necessarily older ones. For full compatibility across all versions you need to use backward-transitive, forward-transitive, and full-transitive. Schema Registry also provides a convenient REST API for uploading, view and testing schema compatibility.
- Schema Registry supports three type of serialization formats Avro, Protocol Buffers, and JSON Schema. It also provides integrated serializers and deserializers you can plug into your KafkaProducer and KafkaConsumer instances for seamless support for all three supported types. The provided (de)serializers cache schemas locally and only fetch them from Schema Registry when it can't locate a schema in the cache.
- Using code generation with tools such as Avro and Protobuf or open source plugins supporting JSON Schema help speed up development and eliminate human error. Plugins that integrate with Gradle and Maven also provide the ability to test and upload schemas in the developers normal build cycle.