B Schema compatibility workshop

In this appendix, you'll take a guided walk through of updating schemas in different compatiblity modes. You'll change the schemas compatibiltiy mode, make changes, test those changes and finally run updated producers and consumers to see the different compatibilty modes in action. I've already made all the changes, you just need to read along and run the provided commands. There are three sub-projects sr-backward, sr-forward, and sr-full. Each sub-project contains producers, consumers and schemas updated and configured for the representative compatiblity mode.

> **NOTE** **There is a lot of overlap with code and the build.gradle files between the sub-projects. This is intentional as I wanted each module isolated. The focus of using these modules is for learning about evolving schemas in Schema Registry and the related changes you need to make to Kafka Producers and Consumers, not how to set up the ideal Gradle project!**

In this section I'll only go into how Schema Registry ensures compatibility between clients. For schema compatibility rules of the serilaiztion frameworks themselves, you'll want to look at each one specifially. Avro schema resolution rules are available here https://avro.apache.org/docs/current/spec.html#Schema+Resolution. Protobuf provides backward compatibility rules in the langage specification found here https://developers.google.com/protocol-buffers/docs/proto3.

Let's go over the different compatibility modes now. For each compatibility mode you'll see the changes made to the scheama and you'll run the a few steps to needed to sucessfuly migrate a schema.

I'd like to point out that for the sake of clarity each schema migration for the different compatibility modes has it's own gradle sub-module in the source code for the book. I did this as each Avro schema file has changes resulting in different Java class structures when you build the code. Instead of having you rename files, I opted for a structure where each migration type can

stand on its own. In a typical development evrionment you will not follow this practice. You'll modify the schema file, generate the new Java code and update the producers and consumers in the same project.

All of the schema migration examples will modify the original `avenger.avsc` schema file. Here's the original schema file for reference so it's easier to see the changes made for each schema migration.

Listing B.1 The original Avenger Avro schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "real_name", "type": "string"},
    {"name": "movies", "type":
                     {"type": "array", "items": "string"},
      "default": []
    }
  ]
}
```

NOTE **For working through schema evolution and the comaptibility types, I've created three sub-modules in the source code, `sr-backward`, `sr-forward`, and `sr-full`. These sub-modules are self contained and intentionally contain duplicated code and setup. The modules have updated schemas, producers and consumers for each type of compatibility mode. I did this to make the learning process easier, as you can look at the changes and run new examples without stepping on the previous ones.**

B.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.

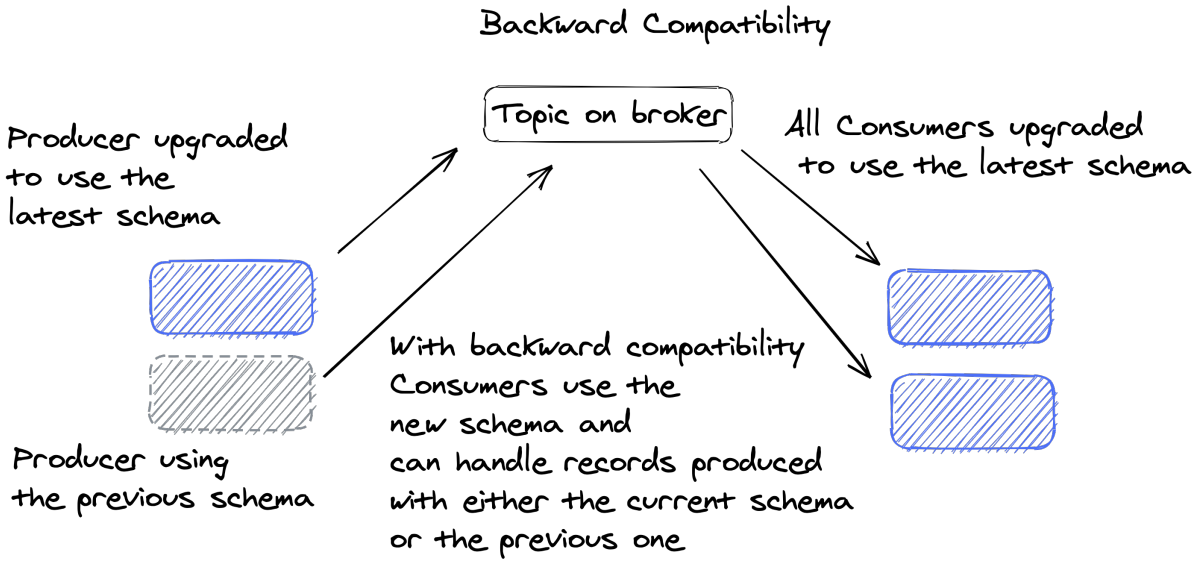

Figure B.1 Backward compatitbility updates consumers first to use the new schema then they can handle records from producers using either the new schema or the previous one

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 specifed default value when deserializing the bytes back into an object.

Before we get started, lets run the producer with the original schema. That way after the next step you'll have records using both the old schema and the new one and you'll be able to see backwards compatibility in action. Make sure you've started docker with `docker-compose up -d`, then run the following commands:

Listing B.2 Producing records with the original schema

```
./gradlew streams:registerSchemasTask   ❶
./gradlew streams:runAvroProducer   ❷
```

❶ Making sure you've registered the original avengers.avsc schema

❷ Run a producer with the original schema

Now you'll have records with the original schema in the topic. When you complete the next step, having these records available will make it clear how backwards compatibility works as the consumer will be able to accept records using the old and the updated schema.

So let's update the original schema and delete the `real_name` field and add a `powers` field with default value.

NOTE The schema file and code can be found in `sr-backward` sub-module in the `src/main/avro` directory in the source code.

Listing B.3 Backwards compatible updated schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "powers", "type":                    ❶
                    {"type": "array", "items": "string"},
      "default": []                                ❷
    },
    {"name": "movies", "type":
                    {"type": "array", "items": "string"},
      "default": []
    }
    ]
}
```

❶ The `powers` field replaced the deleted `real_name` field

❷ Provding a default value of an empty powers list for backwards compatibility.

Now that you have updated the schema, you'll want to test that the schema is compatible before uploading it to Schema Registry. Fortunately testing a new schema is a simple process. You'll use the `testSchemasTask` in the `sr-backward` module from the gradle plugin for testing compatibility. So let's test the compatibility first by running this command from the root of the project:

Listing B.4 Testing a new schema is backwards compatible

```
./gradlew :sr-backward:testSchemasTask
```

IMPORTANT For you to run the example successfully, you need to run the command exacty as its displayed here including the leading `:` character.

The result of running the `testSchemasTask` should be `BUILD SUCESSFUL` which means that the new schema is backwards compatible with the existing one. The `testSchemasTask` makes a call to Schema Registry to compare the proposed new schema against the current one to ensure it's comaptible. Now that we know the new schema is valid, let's ahead and register oit with the following command:

Listing B.5 Registering the new schema

```
./gradlew :sr-backward:registerSchemasTask
```

Again the result of running the register command print a `BUILD SUCESSFUL` on the console. Before we move on to the next step let's run a REST API command to view the latest schema for the `avro-avengers-value`:

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

Running this command should yield results resembling the following:

```
{
  "name" : "avro-avengers-value",
  "version" : 2,
  "schema" : "{\"type\": \"record\", \"namespace\": \"bbejeck.chapter_3\", \"name\":
\"AvengerAvro\"..."
}
```

From the results, you can see the increase in the version from 1 to 2 as you've loaded a new schema. With this changes in place, you'll need to update your clients, starting with the consumer. With compatibility of `BACWARDS` you want to update the consumer first to handle any records produced using the new schema.

For example you originally expected to work with the `real_name` field, but you deleted it in the schema, so you want to remove references to it in the new schema. You also added the `powers` field, so you'll want to be able to work with that field. That also implies you've generated new model objects.

Earlier in the chapter when you ran the `clean, build` command it generated the correct objects for all of our modules. So you should not have to do that now.

Take note that since we are in `BACKWARDS` compatiblity mode, if your updated consumer were to getrecords in the previous format, then it won't blow-up. The updated ignores the `real_name` field, and the `powers` field uses the default value.

After you have updated the consumer, then you'll want to update your producer applications to use the new schema. The `AvroProducer` in the `sr-backward` submodule has had the updates applied already. Now run the following command to producer records using the new schema.

Listing B.6 Producing records with the new schema

```
./gradlew :sr-backward:runAvroProducer
```

You'll see some text scroll by followed by the familiar `BUILD SUCESSFUL` text. If you remember, just a few minutes ago you ran the produce command from the original sub-module adding records in the previous schema. So now that you've run the producer using the new schema, you have a mix of old and new schema records in the topic. But our consumer example should be able to handle both types, since we are in the `BACKWARDS` compatibility mode.

Now when you run the consumer you should be able to see the records produced with the

previous schema as well as the records produced with the new schema. Run the following command to execute the updated consumer:

Listing B.7 Consuming records against new schema

```
./gradlew :sr-backward:runAvroConsumer
```

In the console you should see the first results printing `with powers []`. The empty value indicates those are the older records using the default value, since the original records did not have a `powers` field on the object.

NOTE **For this first compatibility example, your consumer read all the records in the topic. This happened because we used a new `group.id` for the consumer in the `sr-backwards` module and we've configured it to read from the earliest available offset, if none were found. For the rest of the compatiblity examples and modules, we'll use the same `group.id` and the consumer will only read newly produced records. I'll go into full details on the `group.id` configuration and offset behavior in chapter four.**

B.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. Let's go ahead and update the schema again, creating `avenger_v3.avsc` which you can find in the `sr-forward/src/main/avro` directory.

Listing B.8 Foward compatible Avenger schema

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "powers", "type": {
               "type": "array", "items": "string"},
             "default": []
    },
    {"name": "nemeses","type": {                    (1)
              "type": "array","items": "string"
      }
    }
  ]
}
```

(1) Added a new field, `nemeses`

In this new version of the schema, you've removed the `movies` field which defaults to an empty list and added a new field `nemeses`. In forward compatibility you would upgrade the producer

client code first.

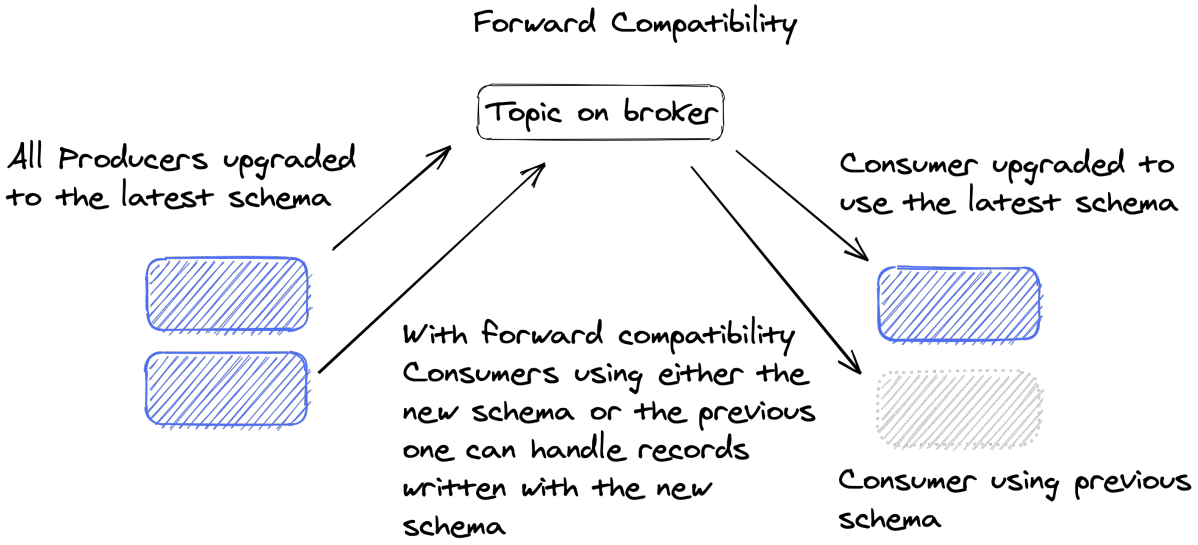

Figure B.2 Forward compatitbility updates producers first to use the new schema and consumers can handle the records either the new schema or the previous one

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.

Now you need to change the compatibility mode from `BACKWARD` to `FORWARD`. In the `sr-forward` sub-module the configuration for the Schema Registry plugin has this section setting the compatibility:

Listing B.9 Comatibility in build.gradle for sr-forward sub-module

```
config {
     subject('avro-avengers-value', 'FORWARD')
 }
```

Now with the configuration set, to change the compatibility mode, run this command:

Listing B.10 Changing the compatbility mode to FORWARD

```
./gradlew :sr-forward:configSubjectsTask
```

As we've seen before, the result of this command produces a `BUILD SUCCESSFUl` result on the console. If you want to confirm the compatibility mode for your subject, you can use this REST API command:

Listing B.11 REST API to view configured compatibility mode for a subject

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

The `jq` at the end of the curl command formats the returned JSON and you should see something like:

Listing B.12 Formatted configuration response

```
{
   compatibility: FORWARD
}
```

Now that you have configured the `avro-avengers-value` subject with forward compatibility, go ahead and test the new schema by running the following command:

Listing B.13 Testing a new schema is foward compatible

```
./gradlew :sr-forward:testSchemasTask
```

This command should print a `BUILD SUCESSFUL` on the console, then you can register the new schema:

Listing B.14 Register the new forward compatible schema

```
./gradlew :sr-forward:registerSchemasTask
```

Then run a producer already updated to send records in the new format with this command:

Listing B.15 Run producer updated for records in the new schema format

```
./gradlew :sr-forward:runAvroProducer
```

Now that you've run the producer with an updated schema let's first run the consumer that **is not** updated:

Listing B.16 Run a consumer not yet updated for the new schema changes

```
./gradlew :sr-backward:runAvroConsumer
```

The results of the command show how that with forward compatibility even if the consumer is **not updated** it can still handle records written using the new schema. Now we need to produce some records again for the ***updated*** consumer:

Listing B.17 Run producer again

```
./gradlew :sr-forward:runAvroProducer
```

Now run the consumer that is updated for the new schema:

Listing B.18 Run consumer updated for new schema

```
./gradlew :sr-forward:runAvroConsumer
```

In both cases, the consumer runs sucessfully, but the details in the console are different due to having upgraded the consumer to handle the new schema.

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 fomrat. In forward compatiblity, 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.

B.3 Full compatibility

In full compatiblity mode, you free to add or remove fields, but there is one catch. **Any changes** you make must be to **optional** fields only.

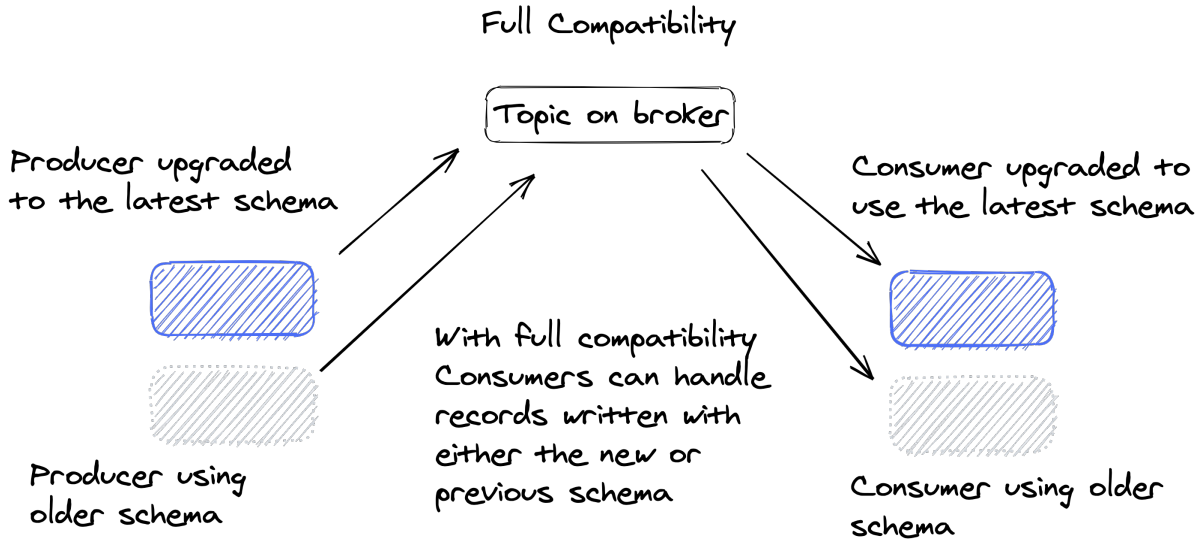

Figure B.3 Full compatitbility allows for producers to send with the previous or new schema and consumers can handle the records either the new schema or the previous one

Since the fields involved in the updated schema are optional, these changes are considered compatibable 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.

Let's take a look at an schema to work with `FULL` compatibility:

Listing B.19 Full compatibility schema avengers_v4.avsc

```
{
  "namespace": "bbejeck.chapter_3.avro",
  "type": "record",
  "name": "AvengerAvro",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "yearPublished", "type": "int", "default": 1960 },   ❶
    { "name": "realName", "type": "string", "default": "unknown" },   ❷
    { "name": "partners", "type": {
                           "type": "array","items": "string"},    ❸
                "default": []
    },
    {"name": "nemeses", "type": {
                         "type": "array", "items": "string"},
              "default": []
    }
  ]
}
```

❶ Added new optional field yearPublished

❷ Added back optional field realName

❸ Added new field partners

Before you update the schema, let's produce a set of records one more time so that we can have a batch of records in the format prior to our next schema change. I'll explain why we are doing this in an upcoming section.

Listing B.20 Run producer again to create a batch of records in the format before we migrate the schema

```
./gradlew :sr-forward:runAvroProducer
```

This will give us a batch of records to read with an updated consumer. But first let's change the compatibility, this time to `FULL`:

Listing B.21 Change the compatibility to FULL

```
./gradlew :sr-full:configSubjectsTask
```

And to keep consistent with our process, let's test the compatibility of the schema before we migrate it:

Listing B.22 Test schema for full compatibility

```
./gradlew :sr-full:testSchemasTask
```

With the migrated schema compatibility tested, let's go ahead and register it

Listing B.23 Register the FULL compatibility schema

```
./gradlew :sr-full:registerSchemasTask
```

With the a new version of the schema registered, let's have some fun with the order of records we produce and consume. Since all of the updates to the schema involve optional fields the order in which we update the producers and consumers doesn't matter.

A few minutes ago, I had you create a batch of records in the previous schema format. I did that to demonstrate that we can use an updated consumer in `FULL` compatiblity mode to read older records. Remember before with `FORWARD` compatibility it was essential to ensure the updated consumers would only see records in the new format.

Now let's run an updated consumer to read records using the previous schema. But let's watch what happens now:

Now run the updated consumer

Listing B.24 Consuming with the updated consumer

```
./gradlew :sr-full:runAvroConsumer
```

And it runs just fine! Now let's flip the order of operations and run the updated producer:

Listing B.25 Producing records with the new schema

```
./gradlew :sr-full:runAvroProducer
```

And now you can run the consumer that we haven't updated yet for the new record format:

Listing B.26 Consuming new records with a consumer not updated

```
./gradlew :sr-forward:runAvroConsumer
```

As you can see from playing with the different versions of producers and consumers with `FULL` compatibility, when you update the producer and consumer is up to you, the order doesn't matter.