TIBCO® Messaging - Schema Repository for Apache Kafka

User Guide

Version 2.0.0 | March 2025

Contents

About this Product

TIBCO Messaging - Schema Repository for Apache Kafka is ideal for implementing application projects (including proof of concept efforts), for testing, and for deploying applications in a production environment.

This release is the latest in a long history of TIBCO products that use the power of Information Bus® technology to enable truly event-driven IT environments. To find out more about how TIBCO® Messaging software and other TIBCO products are powered by TIBCO technology, please visit us at www.tibco.com.

TIBCO Messaging - Schema Repository for Apache Kafka for Apache Kafka is part of the TIBCO Messaging software.

TIBCO Schema Repository Overview

The TIBCO Schema Repository is a RESTful service layer for you to register and manage *schemas* for mutually compatible message formatting between producer and consumer applications. Using an external message format allows for easier development and more robust operation.

The schemas are expressed with the Apache Avro JSON schema representation used with Apache Kafka. The Avro schema representation is widely used, flexible, and compact. Avro handles data structures, field names, and types for structured and maintainable message exchange.

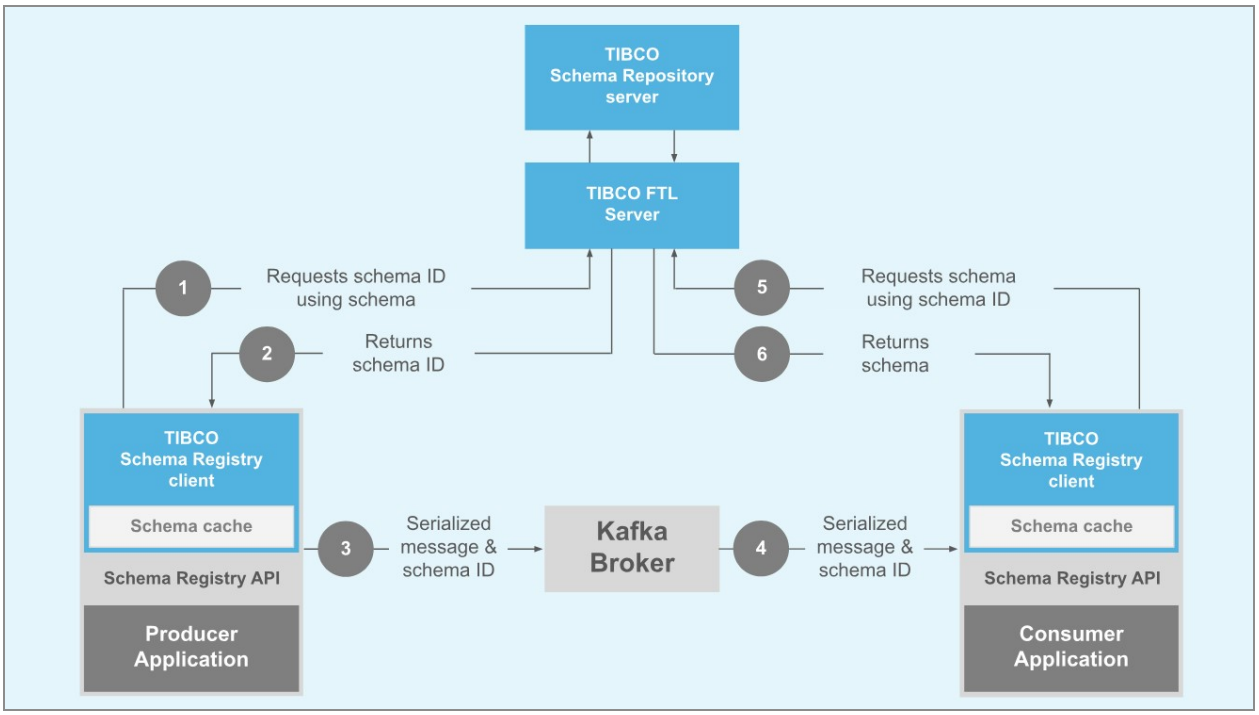
By default, the TIBCO Schema Repository controls access to the functionality of the schema repository by supporting OAUTH2-based roles to control access to specific REST-based operations. For example, applications can be restricted to using only specific schemas or prevented from making changes to existing schemas with control available all the way down to the subject level.

TIBCO FTL® provides storage and is required to run the Schema Repository (see the TIBCO Product Documentation).

Architecture and Message Flow

The Schema Repository is used by producer and consumer apps to create and reference schemas to ensure common representation for non-trivial message formats. The producer and consumer apps send and receive messages through the Kafka broker. The Kafka broker and the Schema Repository do not directly communicate. The TIBCO FTL server is used for storage.

Figure 1: Schema Repository Architecture



The following message flow is shown in this figure:

1. If a schema ID for the fields and types in the message to be sent was not cached locally from a prior message, the producer contacts the Schema Repository to either retrieve the schema ID for an existing schema or create a schema or version of an existing schema.
2. If the schema was previously registered, the Schema Repository will return the schema ID to the producer. Otherwise, the Schema Repository will validate the

schema by checking its compatibility with other schemas registered with the same subject name, register the schema as a new schema or a version of an existing schema, and return the schema ID for the newly registered schema to the producer. This schema ID is cached locally by the client.

> ⚠ **Warning:** If the schema is not compatible, a serialization exception occurs, an error is returned, and the schema is not registered. The consumer is unaware of the error. The developer must resolve the issue.

3. The producer sends the serialized message payload and schema ID to a Kafka broker.
4. The consumer receives the serialized message payload and schema ID from a Kafka broker.
5. If a schema for the schema ID was not cached locally from a prior message, the consumer contacts the Schema Repository to retrieve the schema for the schema ID from the received message.
6. The Schema Repository returns the schema that corresponds to the schema ID. This schema is cached locally by the client.

> ⚠ **Warning:** If the schema ID no longer exists, a deserialization exception is returned to the consumer. For example, the schema ID may have been deleted before all messages with the broker have been consumed.

Terms: Topic, Subject, ID, and Version

The Schema Repository creates and maintains schemas, schema IDs, and versions to work with the Kafka platform.

Kafka *topics* are ordered logs of *messages*. Each message is a key-value pair where the key is an identifier and the value is the content. The message key, the message value, or both can be serialized.

Each topic has a *topic name* that is unique across a Kafka server cluster since a specific Schema Repository (or a set of Schema Repositories) is associated with only a single Kafka cluster. Topic naming is a joint effort between developers and administrators. Administrators create topics. Developers use the topics and often give the administrators a list of topics to create.

A *subject name* is used to register schemas for a producer. Topics and `subjects` are logically unrelated. For example, one producer publishes to the topic `foo` and another producer publishes to the topic `bar` and both use the subject name `baz` in the Schema Repository to register their schemas.

A serializer registers a schema in the Schema Repository by the configured *subject name*, including `key.subject.name` and `value.subject.name`, which are Java properties set programmatically in Java code. The subject name identifies a *namespace* in the Schema Repository for the storage of a schema *ID*, *version*, and metadata. For configuration details, see Apache Avro Client Library Configuration Options, key.subject.name. If the subject name is not configured, the serializer registers a schema by the topic name (`%t-key`).

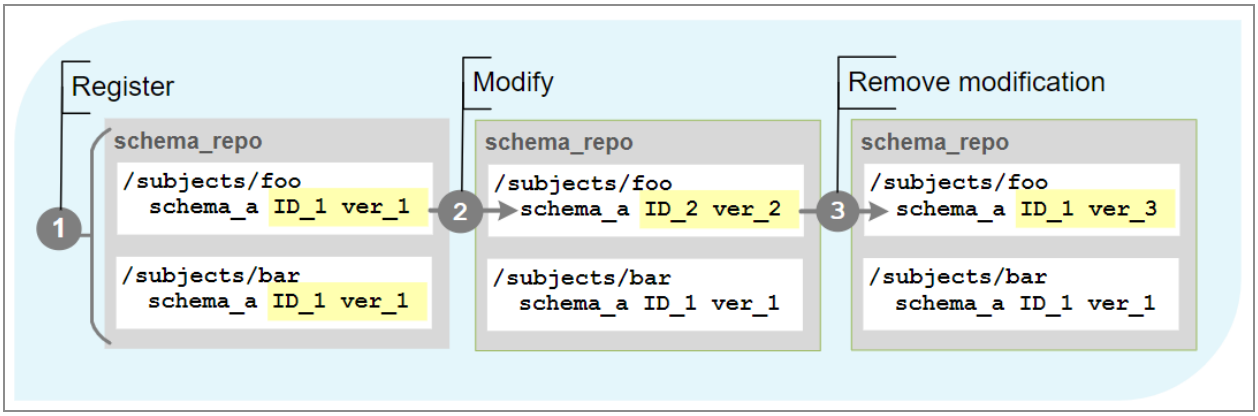
Note: It is difficult to change a name once it is in place, so it is a good practice to keep names generic.

The Schema Repository assigns each schema name a unique schema ID and version. A schema's ID is a monotonically increasing integer, starting at 1, for every new *_unique_ Avro* schema registered in the Schema Repository, regardless of what subject the schema is registered under. Schema IDs are unique across the cluster.

Example: Schemas IDs and Versions

The following figure shows an example of how IDs and versioning work.

Figure 2: Schema IDs and Versions



The following steps are followed in the figure:

1. You register `schema_a` under both `/subjects/foo` and `/subjects/bar`. It is the same schema and has the same schema ID.
2. You modify the schema by adding a field to it and register the new modified schema to only `/subjects/foo`. The `_version_` of `/subjects/foo` goes up to version 2, and the new modified schema gets schema `_ID_`. When the schema was modified, the Schema Repository checked to make sure the evolved schema was compatible, created an ID and version, then added the evolved schema.
3. If you then remove the field just added and register the schema under `/subjects/foo` again, then the `_version_` of `/subjects/foo` goes up to 3, but the `_ID_` of the schema there goes back to 1, because it is the same schema as was originally registered and still being used under `/subjects/bar` as well.

As shown, `_versions_` are really versions of `_subjects_` in the Schema Repository, not really versions of schemas. Both IDs and versions increase monotonically. If a schema is deleted, neither versions nor IDs are re-used for other schemas in the future.

Compatibility Types

In the Schema Registry, compatibility policies are created at the schema metadata level and define the evolution rules for each schema. If the Schema Repository determines a schema is not compatible, an error is returned.

Important: You set compatibility policies when you add a schema. Once a compatible policy is set, it should not be changed. The default is `BACKWARD`.

You can configure compatibility using any of the following values.

- **`BACKWARD:`** (Default) A newly registered schema version must be compatible with the last registered version. This means that a consumer using the newly registered version of a schema is able to deserialize data written by producers that used the previous version.

  Changes allowed:

  - Deleting fields
  - Adding optional fields

- **`BACKWARD_TRANSITIVE:`** A newly registered schema version must be compatible with all previous registered versions. This means that a consumer using the newly registered version of a schema is able to deserialize data written by producers that used any previous version.

  Changes allowed:

  - Deleting fields
  - Adding optional fields

- **`FORWARD:`** The last registered schema version must be compatible with a newly registered version. This means that a consumer using the last registered version of a schema is able to deserialize data written by producers using the newly registered version of a schema.

  Changes allowed:

  - Adding fields

    - Deleting optional fields

- **`FORWARD_TRANSITIVE:`** All previously registered schema versions must be compatible with a newly registered version. This means that a consumer using any previously registered version of a schema is able to deserialize data written by producers using the most newly registered version of a schema.

  Changes allowed:

    - Adding fields
    - Deleting optional fields

- **`FULL:`** A newly registered schema must be both forward and backward compatible. This means that a consumer using the newly registered version of a schema is able to deserialize data written by producers that used the previous version, and a consumer using the last registered version of a schema is able to deserialize data written by producers using the newly registered version of a schema.

  Changes allowed:

    - Adding optional fields
    - Deleting optional fields

- **`FULL_TRANSITIVE:`** A newly registered schema must be both forward transitive and backward transitive compatible. This means that a consumer using any version of a schema will be able to process data produced with any other version of the same schema.

  Changes allowed:

    - Adding optional fields
    - Deleting optional fields

- **`NONE:`** No compatibility policy exists, and there is no compatibility checking.

  Changes allowed:

    - All changes are allowed

Important: If you set transitive configurations (FULL_TRANSITIVE, BACKWARD_TRANSITIVE, or FORWARD_TRANSITIVE), no error is produced. However, the compatibility check is only against the last schema, not all schemas.

Client Configuration Properties

Important: The properties you can set for the producer and consumer apps include but are not limited to the properties covered in this section. The information given in this topic provides examples for your reference. For your requirements, consider all the configuration properties in the section Apache Avro Client Library Configuration Options.

Minimum Configuration Properties

Minimum configuration properties common to both producers and consumers follow. See the descriptions and code example below.

The `ftl.servers` property includes a pipe-delimited list of the realm servers that you want to connect to. For details, see Access Configuration Properties.

The `ftl.trust.file` setting points to the certificate file. The `ftl.trust.type` option specifies if you want to use a `"file"`, a `"string"`, or `"everyone"`.

In the example below, the trust type is `file` followed by a path to the TIBCO FTL server certificate file. By trusting the TIBCO FTL server, you transitively trust the `tibschemad` certificate as well for client communications. The `tibschemad` certificate is generated at run time and signed by the TIBCO FTL server. This approach is handy because you do not need to trust the `tibschemad` certificate itself. For details, see Security Configuration Properties.

```
...
properties.setProperty("ftl.servers", "https://localhost:8585");

properties.setProperty("ftl.trust.type", "file");

properties.setProperty("ftl.trust.file", "/path/to/ftl-trust.pem");
```

```
// create the producer using the properties just set above
KafkaProducer<Object, GenericRecord> producer = new KafkaProducer<>
(properties);
```

The following code example uses the `ftl.servers` property to point to the TIBCO FTL server. Trust is established with the TIBCO FTL server and, transitively, `tibschemad`. The TIBCO FTL server automatically redirects to the client where `tibschemad` is listening.

```
Properties properties = new Properties();


properties.setProperty("bootstrap.servers", "localhost:9092");


properties.setProperty("ftl.servers",
"https://127.0.0.1:8585|https://127.0.0.1:8686|https://127.0.0.1:8787");


properties.setProperty("ftl.trust.type", "file");


properties.setProperty("ftl.trust.file", "/path/to/ftl-server/ftl-
trust.pem");
```

Producer Configuration Properties

For producers, set the serializer for both the key and value properties.

```
properties.setProperty("key.serializer",
"com.tibco.messaging.kafka.avro.AvroSerializer");


properties.setProperty("value.serializer",
"com.tibco.messaging.kafka.avro.AvroSerializer");
```

Consumer Configuration Properties

For consumers, set the deserializer for both the key and value properties.

Note: This should be complementary to the serializer specified in the producer. Intervendor interoperability for different serializer and deserializer pairs is not defined.

```
properties.setProperty("key.deserializer",
"com.tibco.messaging.kafka.avro.AvroDeserializer");


properties.setProperty("value.deserializer",
"com.tibco.messaging.kafka.avro.AvroDeserializer");


properties.setProperty("group.id", "0" /* Or whatever consumer group you
want here. */);
```

Interoperability and Evolution

Schemas change over time. With the TIBCO Schema Repository, developers can modify schemas and be assured the modification does not break another messaging application. For example, you can add or delete fields or change data types. The Schema Repository conducts compatibility checks to ensure schemas safely evolve without accidentally breaking producer and consumer applications or causing other error conditions. The Schema Repository defines compatibility settings, which control when and how new schemas and versions are created and used. Schema versions are accessible to ensure applications use the correct message format.

As you evolve message content, carefully consider field names. Application logic is driven by a number of factors when receiving a message, such as the existence of a field (some are optional), the type of field, and the contents of the field. Named fields can be referenced within a message, even as other fields in a message are added, deleted, or changed. Add new fields for new information in a message rather than reusing an existing field. For example, you would not change the contents of a message to put "addresstwo" information into a field originally intended for "nametwo" information. Doing so would add complexity to handle both cases in a reasonable and consistent way. A better approach is to add a field for the "addresstwo" information and not use the "nametwo" field if it is not needed. This way, the existence and contents of a field can reliably be associated with the business logic without dealing with overloaded fields having multiple interpretations.

Fault Tolerance

When running the Schema Repository with the store type ftlkv, the TIBCO FTL server is used for back-end storage, so tibschemad is stateless.

Deploy `tibschemad` as you would any stateless service. It is suggested that you run multiple instances of `tibschemad` on two or more machines so the REST interface provides safety, including failover and fault tolerance (FT).

Multiple `tibschemad` instances can be active at a time. Each `tibschemad` registers with the TIBCO FTL server that started it and serves any requests that are made to that TIBCO FTL server. The TIBCO FTL server monitors a periodic heartbeat and restarts the `tibschemad` service if the instance fails. The `tibschemad` instances should use persistence servers in the same cluster to ensure that they use the same data to respond to requests.

A health check can be run with `GET https://ftl_server_host:port/schema/v1/ping`.

Serializers and Deserializers

The Schema Repository includes a library for serializing and deserializing messages. A producer application uses a schema to serialize a message for transmission. The consumer application deserializes the incoming message using the schema ID referenced in the message to:

- Validate the message has been serialized using a recognized schema
- Identify the fields in the message by field name and data type

The resulting message content is made available to the consumer.

In the application code, developers reference the Avro serializer for the producer to encode multi-field messages (and possibly also register a schema in the Schema Repository). Developers reference the deserializer to ensure the consumer is using the correct known schema before decoding the message for processing.

- Producer: `com.tibco.messaging.kafka.avro.AvroSerializer`
- Consumer: `com.tibco.messaging.kafka.avro.AvroDeserializer`

Important: The correct and consistent set of serializer/deserializer (SerDes) routines must be used to convert the message data in each direction. This means that both the producer and the consumer and the entire set of messaging applications must use SerDes routines from the same vendor. Interoperability between different vendors is happenstance and cannot be counted on for continued interoperability.

The Schema Repository supports these primitive types:

```
Boolean
```

```
byte[]
```

```
Double
```

```
Float
```

```
Integer
```

```
Long
```

```
String
```

The Schema Repository supports the following Avro logical types:

```
UUID
```

```
Date
```

```
TimeMillis
```

```
TimeMicros
```

```
TimestampMillis
```

```
TimestampMicros
```

```
LocalTimestampMillis
```

```
LocalTimestampMicros
```

For details on Avro schemas, as well as serializing and deserializing, see the Apache Avro documentation.

Schema Repository Daemon and JAR files

The Schema Repository has the following main components.

- The daemon `tibschemad` listens to the API requests.
- A client Java ARchive (JAR) file manages interactions with the Schema Repository and also contains the Avro serializer/deserializer (SerDe).

`tibschemad` Daemon

The `tibschemad` daemon, which listens for REST requests from producers or consumers and validates, stores, and distributes Avro message schemas, which are defined using a JSON representation. The OpenAPI specification for the REST interface is available at run time from `https://ftl_server_host:port/schema/v1/openapi.yaml`. For more information, see Using the REST API and Swagger UI.

Schema validation ensures accurate format conversion, routing, and data quality. The `tibschemad` daemon checks for backward compatibility as schemas evolve and versions schemas as appropriate.

Client JAR files with Avro Serializer/Deserializer (SerDe)

The provided client JAR files contain the Avro serialization library. Avro uses a JSON format for the serializer/deserializer (SerDe). These client JAR files aggregate methods to perform the following functions:

- Generate schemas based on message field names and types
- Contact the Schema Repository to automatically register a schema or new version of an existing schema for producers

> **Important:** You can configure the schema registration (`schema.registration.strategy`) to `auto`, `none`, or `latest`. For more information, see Apache Avro Client Library Configuration Options.

- Automatically retrieve schemas by the associated schema ID and version for consumers
- Serialize and deserialize Avro-formatted messages for producers and consumers respectively

The following JAR files are included:

- `tibftl-kafka-avro-2.0.0.jar`: This is the default client JAR file that includes serializers, deserializers, and related functions. To get the proper dependencies in your environment, refer to the `build.gradle` file in the `${TIBCO_HOME}/akd/repo/samples` directory.
    - implementation group: 'com.squareup.retrofit2', name: 'retrofit', version: '2.9.0'
    - implementation group: 'com.squareup.okio', name: 'okio', version: '2.9.0'
    - implementation group: 'com.squareup.retrofit2', name: 'converter-gson', version: '2.11.0'
    - implementation group: 'com.networknt', name: 'json-schema-validator', version: '1.0.87'
    - implementation group: 'io.github.erdtman', name: 'java-json-canonicalization': version: '1.1'
    - implementation group: 'org.msgpack', name: 'jackson-dataformat-msgpack', version: '0.9.9'
    - implementation group: 'com.kjetland', name: 'mbknor-jackson-jsonschema_2.13', version: '1.0.39'
    - implementation group 'org.apache.avro', name: 'avro', version: '1.12.0'
    - implementation group: 'org.slf4j', name: 'slf4j-simple', version: '2.0.16'
    - implementation group: 'org.bouncycastle', name: 'bcpkix-lts8on', version: '2.73.7'
    - implementation group: 'org.bouncycastle', name: 'bcprov-lts8on', version: '2.73.7'

- implementation group: 'com.squareup.okhttp3', name: 'okhttp', version: 4.12.0'
- implementation group: 'com.auth0', name: 'java-jwt', version: '4.4.0'

The JAR files also contain an `AvroConverter` class that can be used with the Kafka Connect framework. You can write messages to disk on your Kafka broker if you want the on-disk representation of a Kafka topic to be in serialized Avro binary format. For example, you can get an TIBCO FTL formatted message then write it to disk in Avro format.

Important: If you are using the Avro client JAR file in your producer or consumer apps, make sure the `kafka-clients-x.y.z.jar` and the dependencies listed above are on the classpath.

Using Test Apps

> **Caution:** The sample producer and consumer apps are provided for testing only and should not be used in a production environment. The samples may not detect and handle all error conditions. They are designed for illustrative and reference purposes and are not intended to be directly used in a production environment.

Select the Test App to Use

Sample apps are included to show how to use the Avro serializer and deserializer at a very basic level. The following sample apps are located in the `akd/repo/samples` directory.

- `AvroProducer` and `AvroConsumer`
- `JsonProducer` and `JsonConsumer`
- `TibProducer` and `TibConsumer`: Samples include security.

Example: AvroProducer and AvroConsumer Examples

The `AvroProducer` and `AvroConsumer` samples are very limited in functionality; properties such as the FTL server URLs cannot be set, and only HTTP connections are supported. The following are excerpts from the included `AvroProducer` and `AvroConsumer` sample applications.

AvroProducer

```
Properties properties = new Properties();
.
.
// We want to send messages in TIBCO's Avro format, so we will use
TIBCO's AvroSerializer
```

```
// class from 'tibftl-kafka-avro-2.0.0.jar' to serialize both the keys
and the values of
// outgoing messages.
properties.setProperty("key.serializer",
"com.tibco.messaging.kafka.avro.AvroSerializer");
properties.setProperty("value.serializer",
"com.tibco.messaging.kafka.avro.AvroSerializer");

.
.
.

KafkaProducer<Object, GenericRecord> producer = new KafkaProducer<>
(properties);
```

AvroConsumer

For a consumer app, the same options can be applied, except you use `key.deserializer` and `value.deserializer` in place of `key.serializer` and `value.serializer`.

```
Properties properties = new Properties();

.
.
.

// Required list of Kafka brokers.
properties.setProperty("bootstrap.servers", "localhost:9092");

// We expect to receive messages in TIBCO's Avro format, so we will use
TIBCO's AvroDeserializer
// class from 'tibftl-kafka-avro-2.0.0.jar' to deserialize both the keys
and the values of
// incoming messages.
properties.setProperty("key.deserializer",
"com.tibco.messaging.kafka.avro.AvroDeserializer");
properties.setProperty("value.deserializer",
"com.tibco.messaging.kafka.avro.AvroDeserializer");

// List of TIBCO FTL server URLs.  The FTL server forwards Schema
Repository requests
// to the Schema Repository server and handles automatic failover of
Schema Repository servers.
```

```
// In this example, we assume that the FTL server is running locally and
listening on
// port 13131, and that the Schema Repository is also running and
configured with the same FTL
// servers list.
properties.setProperty("ftl.servers", "https://localhost:13131/");

.
.
.

KafkaConsumer<Object, GenericRecord> consumer = new KafkaConsumer<>
(properties);
```

Example: JsonProducer and JsonConsumer

The `JsonProducer` and `JsonConsumer` samples are very limited in functionality; properties such as the FTL server URLs cannot be set, and only HTTP connections are supported. The following are excerpts from the included `JsonProducer` and `JsonConsumer` sample applications.

JsonProducer

```
Properties properties = new Properties();
.
.
// We want to send messages in TIBCO's JSON format, so we will use
TIBCO's JsonSerializer
// class from 'tibftl-kafka-avro-2.0.0.jar' to serialize both the keys
and the values of
// outgoing messages.
properties.setProperty("key.serializer",
"com.tibco.messaging.kafka.json.JsonSerializer");
properties.setProperty("value.serializer",
"com.tibco.messaging.kafka.json.JsonSerializer");
.
.
.
KafkaProducer<Object, JsonMessageWithSchema> producer = new
KafkaProducer<>(properties);
```

JsonConsumer

```
Properties properties = new Properties();
.
.
// We expect to receive messages in TIBCO's JSON format, so we will use
TIBCO's JsonDeserializer
// class from 'tibftl-kafka-avro-2.0.0.jar' to deserialize both the keys
and the values of
// incoming messages.
properties.setProperty("key.deserializer",
"com.tibco.messaging.kafka.json.JsonDeserializer");
properties.setProperty("value.deserializer",
"com.tibco.messaging.kafka.json.JsonDeserializer");
properties.setProperty("json.value.type",
"com.fasterxml.jackson.databind.JsonNode");
.
.
.
KafkaConsumer<Object, JsonNode> consumer = new KafkaConsumer<>
(properties);
consumer.subscribe(Arrays.asList("JsonHelloWorld"));
```

Example: TibProducer and TibConsumer Examples (Secure)

The `TibProducer` and `TibConsumer` test apps test a secure setup using the username, password, and trust types. The details of these precompiled test apps follow.

TibProducer Test App

A sample command line to get to the `TibProducer` test app is shown below. Use the current version for your installation.

```
java com.tibco.messaging.kafka.samples.TibProducer -h

TibProducer [options]
    -bs, --bootstrap-server <String: broker to connect to>
        REQUIRED
        Address of Kafka broker to connect to.
```

```
        Example: localhost:9092
    -fs, --ftl-servers <String: List of URLs>
        REQUIRED
        Pipe-delimited list of FTL server URLs used to connect to the
schema
        repository. URLs in the list are tried in order until a
successful
        connection is made.
        Example: https://server-one:8081,https://server-two:8081
    -t, --topic <String: Kafka topic name>
        REQUIRED
        Kafka topic to produce messages to.
        Example: avro-messages
    -c, --count <Integer: number of messages>
        Number of messages to produce before exiting. If not set,
messages
        will be produced indefinitely until TibProducer is stopped.
        Example: 10
    -d, --delay <Integer: milliseconds between messages>
        Amount of time to pause, in milliseconds, between sending each
message.
        If 0, TibProducer will produce messages as fast as possible. For
simple
        performance testing, combine a delay of 0 with the --quiet flag.
        Default: 1000
    -srs, --schema-registration-strategy <String: registration strategy
type>
        Can be one of 'auto', 'latest', or 'none'.
        When set to 'auto', TibProducer will attempt to register its
Avro
        schema automatically with the schema repository if it is not
already
        registered as the latest version in the configured subject. If
set to
        'latest', TibProducer will use the latest schema version
registered in
        the configured subject as long as it is compatible even if it is
not an
        exact match. If set to 'none', TibProducer's schema must already
be
        registered under some version in the configured subject.
TibProducer
        will fail to produce any messages and exit if it is not.
        Default: auto
    -ksn, --key-subject-name <String: schema repository subject>
```

```
        Subject to use when registering TibProducer's key schema with
the
        schema repository. The following special character sequences are
        recognized and automatically expanded:
        %t (expands to topic name used)
        %r (expands to name of Avro record being produced)
        %% (expands to a literal percent character)
        Default: %t-key
    -vsn, --value-subject-name <String: schema repository subject>
        Subject to use when registering TibProducer's value schema with
the
        schema repository. The following special character sequences are
        recognized and automatically expanded:
        %t (expands to topic name used)
        %r (expands to name of Avro record being produced)
        %% (expands to a literal percent character)
        Default: %t-value
    -tf, --trust-file <String: path to trusted certificates file>
        REQUIRED if using a secure (https) FTL connection and
        --trust-everyone is not specified.
        The certificates to trust can either be in PEM format or in a
Java
        KeyStore file.
        Example: /opt/ftl-server/ftl-trust.pem
    -tfp, --trust-file-password <String: location of password>
        REQUIRED if the --trust-file option specifies a Java KeyStore
file
        that is password-protected. Specifies the password for the
KeyStore.
        Supply one of the following forms:
        stdin | env:<environment_variable_name> |
        file:<password_file_path> | pass:<password>
        Using 'pass' to specify a password directly on the command-line
is
        supported for convenience but should not be considered secure.
    -te, --trust-everyone <Boolean flag>
        REQUIRED if using a secure (https) FTL connection and --trust-
file
        is not specified.
        Trust any TLS certificate encountered. This is NOT secure and
may
        expose user credentials to unauthorized parties; it is supported
as a
        convenience for testing purposes.
    -ccf, --client-certificate-file <String: mTLS client certificate
```

```
file path>
        REQUIRED if using an FTL server running mTLS.
        The certificates can either be in PEM format or in a Java
KeyStore file.
        Example: cert.pem
    -cpkf, --client-private-key-file <String: mTLS private key file
path>
        REQUIRED if using an FTL server running mTLS.
        The private key must be in PEM format.
        Example: privkey.pem
    -cpkp, --client-private-key-password <String: mtlS private key
password>
        REQUIRED if the --client-private-key-file option specifies a
private key
        encrypted with a password. Specifies the password for the
private key file.
        Supply one of the following forms:
        env:<environment_variable_name> |
        file:<password_file_path> | pass:<password>
        Using 'pass' to specify a password directly on the command-line
is
        supported for convenience but should not be considered secure.
    -at, --auth-type <String: authorization type>
        The authorization type of the Schema Registry client, either
'oauth2' or 'none'.
        This value should match the authorization type of the Schema
Repository server.
        If no type is specified at startup, the client will default to
expecting OAUTH2 parameters.
        Example: oauth2
    -o2t, --oauth2-token <String: OAUTH2.0 access token>
        A JWT that provides access to resources in the Schema Repository
server.
        Example:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI
6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6y
JV_adQssw5c
    -o2ts, --oauth2-token-supplier <String: path to TokenSupplier
implementation>
        The period-separated path to a class that implements the
TokenSupplier interface.
        Example: com.tibco.messaging.kafka.samples.SampleTokenSupplier
    -o2taw, --oauth2-token-autorefresh-window <Integer: number of
seconds>
        The number of seconds before a token expires in which a new
```

```
token
        will automatically be retrieved using the token supplier
function
        when an outgoing request is made.
        Example: 60
    -o2cral, --oauth2-connection-retry-auth-limit <Integer: number of
retries>
        The number of times the client should attempt to connect to the
        Schema Repository server after an attempt fails with a 401
Unauthorized error.
        This number defaults to 20 and cannot exceed 20.        Example:
10
    -q, --quiet
        Do not print the contents of each message produced. Recommended
when
        using TibProducer to conduct simple performance or load tests.
    -h, --help
        Print this usage text and exit.
```

TibConsumer Test App

A sample command line to get to the `TibConsumer` test app is shown below. Use the current version for your installation.

```
java com.tibco.messaging.kafka.samples.TibConsumer -h

TibConsumer [options]
    -bs, --bootstrap-server <String: broker to connect to>
        REQUIRED
        Address of Kafka broker to connect to.
        Example: localhost:9092
    -fs, --ftl-servers <String: List of URLs>
        REQUIRED
        Pipe-delimited list of FTL server URLs used to connect to the
schema
        repository. URLs in the list are tried in order until a
successful
        connection is made.
        Example: https://server-one:8081,https://server-two:8081
    -t, --topic <String: Kafka topic name>
        REQUIRED
        Kafka topic to consume messages from.
        Example: avro-messages
```

```
    -c, --count <Integer: number of messages>
        Number of messages to consume before exiting. If not set,
messages
        will be consumed indefinitely until TibConsumer is stopped.
        Example: 10
    -tf, --trust-file <String: path to trusted certificates file>
        REQUIRED if using a secure (https) FTL connection and
        --trust-everyone is not specified.
        The certificates to trust can either be in PEM format or in a
Java
        KeyStore file.
        Example: /opt/ftl-server/ftl-trust.pem
    -tfp, --trust-file-password <String: location of password>
        REQUIRED if the --trust-file option specifies a Java KeyStore
file
        that is password-protected. Specifies the password for the
KeyStore.
        Supply one of the following forms:
        stdin | env:<environment_variable_name> |
        file:<password_file_path> | pass:<password>
        Using 'pass' to specify a password directly on the command-line
is
        supported for convenience but should not be considered secure.
    -te, --trust-everyone <Boolean flag>
        REQUIRED if using a secure (https) FTL connection and --trust-
file
        is not specified.
        Trust any TLS certificate encountered. This is NOT secure and
may
        expose user credentials to unauthorized parties; it is supported
as a
        convenience for testing purposes.
    -ccf, --client-certificate-file <String: mTLS client certificate
file path>
        REQUIRED if using an FTL server running mTLS.
        The certificates can either be in PEM format or in a Java
KeyStore file.
        Example: cert.pem
    -cpkf, --client-private-key-file <String: mTLS private key file
path>
        REQUIRED if using an FTL server running mTLS.
        The private key must be in PEM format.
        Example: privkey.pem
    -cpkp, --client-private-key-password <String: mtlS private key
password>
```

```
        REQUIRED if the --client-private-key-file option specifies a
private key
        encrypted with a password. Specifies the password for the
private key file.
        Supply one of the following forms:
        env:<environment_variable_name> |
        file:<password_file_path> | pass:<password>
        Using 'pass' to specify a password directly on the command-line
is
        supported for convenience but should not be considered secure.
    -at, --auth-type <String: authorization type>
        The authorization type of the Schema Registry client, either
'oauth2' or 'none'.
        This value should match the authorization type of the Schema
Repository server.
        If no type is specified at startup, the client will default to
expecting OAUTH2 parameters.
        Example: oauth2
    -o2t, --oauth2-token <String: OAUTH2.0 access token>
        A JWT that provides access to resources in the Schema Repository
server.
        Example:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI
6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6y
JV_adQssw5c
    -o2ts, --oauth2-token-supplier <String: path to TokenSupplier
implementation>
        The period-separated path to a class that implements the
TokenSupplier interface.
        Example: com.tibco.messaging.kafka.samples.SampleTokenSupplier
    -o2taw, --oauth2-token-autorefresh-window <Integer: number of
seconds>
        The number of seconds before a token expires in which a new
token
        will automatically be retrieved using the token supplier
function
        when an outgoing request is made.
        Example: 60
    -o2cral, --oauth2-connection-retry-auth-limit <Integer: number of
retries>
        The number of times the client should attempt to connect to the
        Schema Repository server after an attempt fails with a 401
Unauthorized error.
        This number defaults to 20 and cannot exceed 20.        Example:
10
```

```
    -q, --quiet
        Do not print the contents of each message consumed. Recommended
when
        using TibConsumer to conduct simple performance or load tests.
    -h, --help
        Print this usage text and exit.
```

Using the REST API and Swagger UI

TIBCO's Avro serialization library is contained in the JAR file. The library directs REST requests via the TIBCO FTL server to the Schema Repository that is accessible at `ftl_server_host:port/schema/v1/`.

To access the online REST API documentation, start the Schema Repository, then use a browser to view `https://`*`ftl_server_host`*`:`*`port`*`/schema/v1/openapi.yaml`. You can use the Swagger UI to access `tibschemad` documentation on the REST API. You can paste in a bearer token to authorize with for schema repository servers running with OAUTH2.

Use the Swagger UI to perform live queries and monitor and update the Schema Repository. You can quickly try different requests and see the results in real time. This way, you interactively build and test REST requests before incorporating them into your applications and scripts.

Swagger serves up the pages and `curl` makes REST requests. For example, you can browse the Schema Repository or delete a schema. You can view the parameters, the request URL, the server response, the response header and body, example values, and more. The `curl` commands listed can be used to call the Schema Repository API.

Access to the Swagger UI

To access the Swagger UI, start the Schema Repository and use a browser to view `https://ftl_server_host:port/schema/v1/swagger`. For example, `https://localhost:8585/schema/v1/swagger/`.

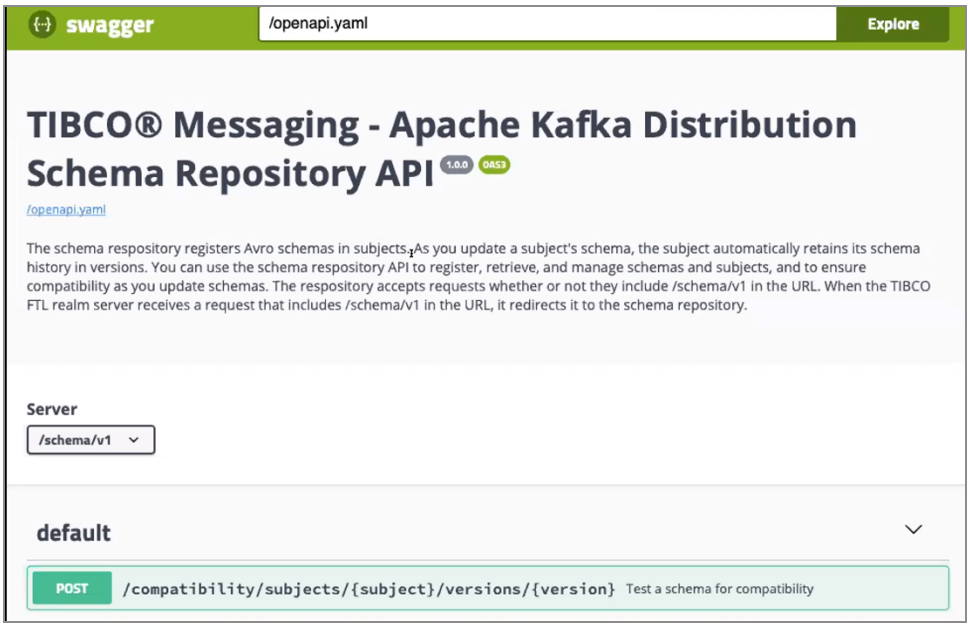
The Swagger UI shows the entire REST API. To use the API,

1. Scroll to the command of interest.
2. Click **Try it out**.
3. Add the appropriate options and parameters.
4. Click **Execute**.

The results are shown on the page for review.

REST API Examples Using Swagger

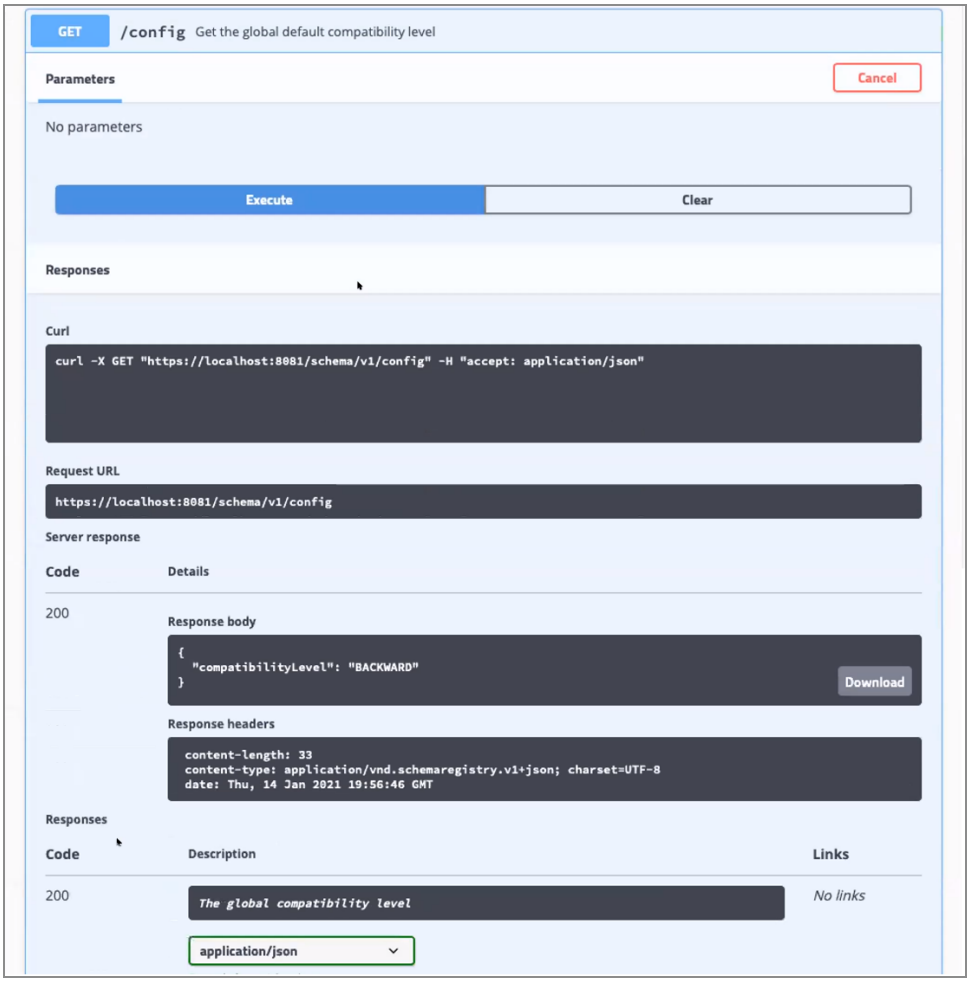
The following screen shows the `/openapi.yaml` which is the open API definition for this Schema Repository. It can be imported into an app if you want to use it directly rather than using the TIBCO Avro client JAR file.



Example: Get Global Default Compatibility Level

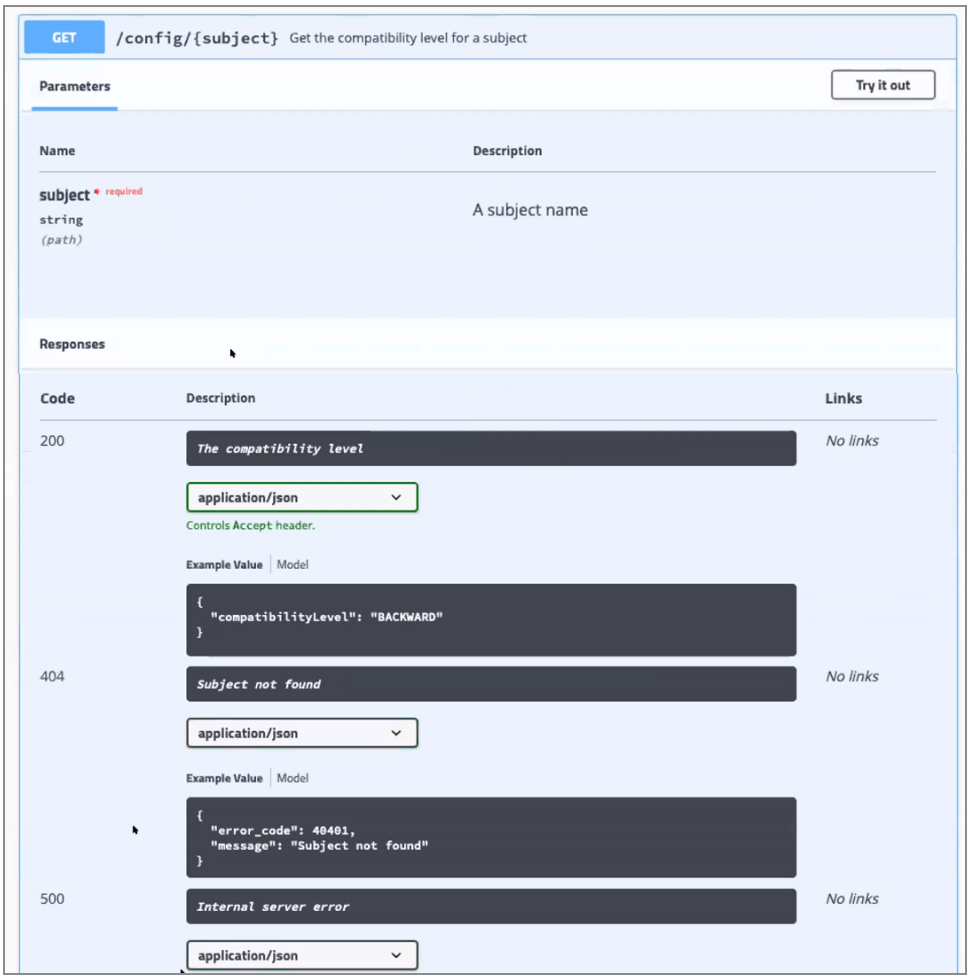
The following example shows how to retrieve the global schema compatibility level response. This is useful to configure how the Schema Repository generates new versions of schemas in terms of the compatibility type, which may be `BACKWARD`, `FORWARD`, `FULL`, `BACKWARD_TRANSITIVE`, `FORWARD_TRANSITIVE`, `FULL_TRANSITIVE`, or `NONE`. (For details on each type, see Compatibility Types.) In this example, a command takes no parameters and returns the results as part of the response body for parsing by the requesting application. The example shows a complete `curl` command line to request this information directly from the Schema Repository (not using the Swagger UI) together with:

- The request URL
- The return code (which varies in accordance to the request itself or the validity of any supplied parameters indicating success or non-success of the overall request)
- The response headers and body contents



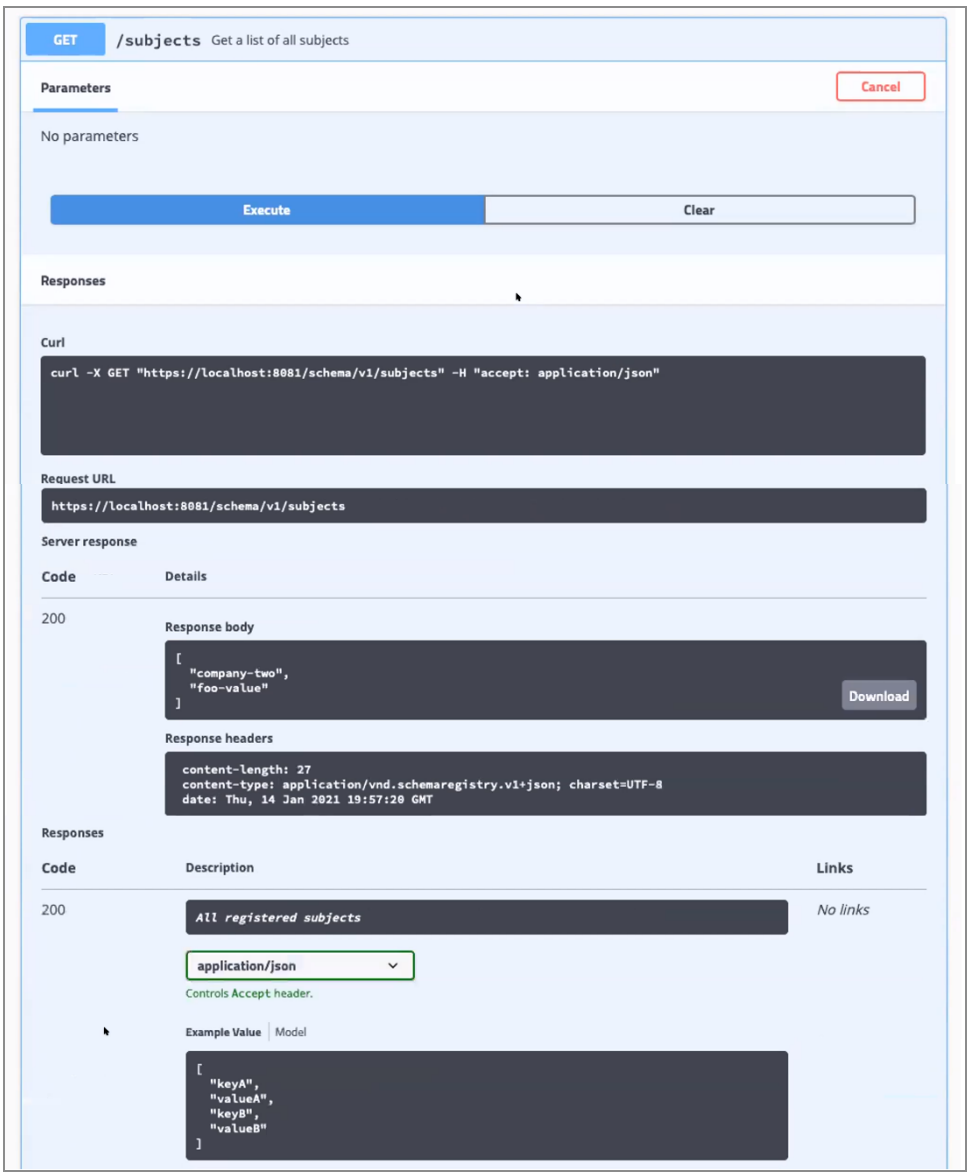
Example: Get Compatibility Level for a Subject

The following example shows how to use the Swagger UI to see descriptions of the API, what is expected, and what can be returned as an error. This is useful to get the compatibility settings for a specific subject rather than the overall global setting. The parameter `{subject}` must be specified for successful execution. You can use an invalid subject to see the error result.



Example: Get List of All Subjects

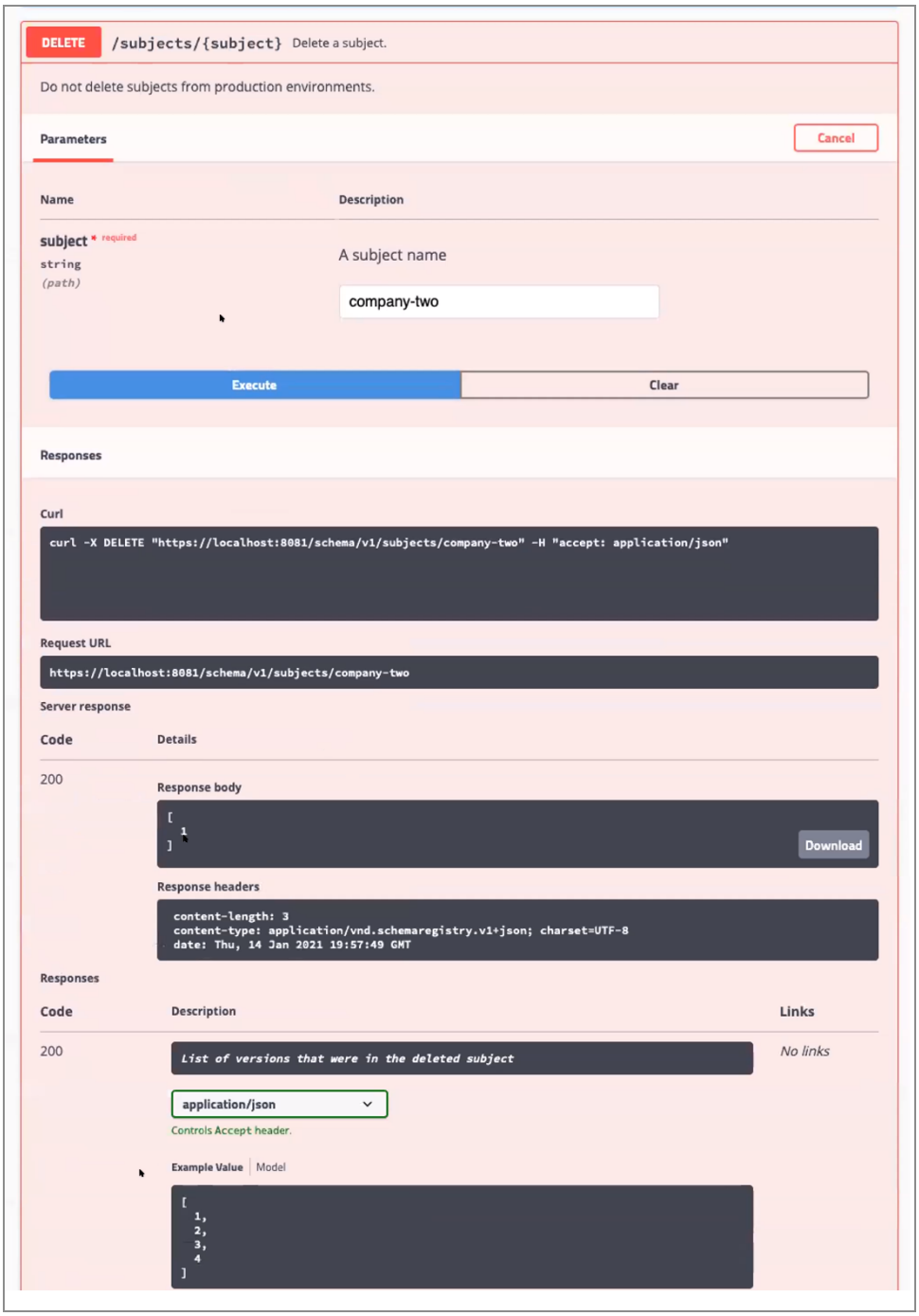
The following image illustrates getting a list of all schema subjects. This is a good example of where a list of items is returned in the body of response. Depending on the type of request, the body may be empty, contain a single value, or contain a list of values.



Example: Delete a Subject

To delete a subject, use the delete command and identify the subject name. When you click **Execute**, you see the number of versions deleted and other details. Deleting a schema removes the versioned instances, but the schema (MD5 hash) still exists and you can run a lookup on the deleted schema on the API.

Note: This performs a soft delete. For soft and hard delete information, see the DELETE commands in tibschemad Command Line Reference.


DELETE
/subjects/{subject}
Delete a subject.
Do not delete subjects from production environments.
Parameters
Cancel
Name
Description
subject * required
string
(path)
A subject name
company-two
Execute
Clear
Responses
Curl
curl -X DELETE "https://localhost:8081/schema/v1/subjects/company-two" -H "accept: application/json"
Request URL
https://localhost:8081/schema/v1/subjects/company-two
Server response
Code
Details
200
Response body
[
1
]
Download
Response headers
content-length: 3
content-type: application/vnd.schemaregistry.v1+json; charset=UTF-8
date: Thu, 14 Jan 2021 19:57:49 GMT
Responses
Code
Description
Links
200
List of versions that were in the deleted subject
No links
application/json
Controls Accept header.
Example Value
Model
[
1,
2,
3,
4
]

Password Security

This topic is relevant to the client using a password for mTLS. The `password` argument does not exist for the server.

Specifying a Password Argument

When you supply a password as a command-line argument, that argument is visible to casual observers. For example, command-line arguments appear in the output of the UNIX `ps` command.

`-p` *password* or **`--password`** *password*

```
Optional. Required for communication with a secure FTL server.

The repository authenticates itself to the FTL server using this
password. The password can be supplied using one of the following forms:

    • env:environment_var_name
    • file:password_file_path
    • pass:password
```

Form

You can supply password arguments in any of the following forms. Choose exactly one form.

`env:`*environment_var*

```
You must set an environment variable in the environment accessible to
the Schema Repository process. The value of that variable is the
password string. You must ensure that only authorized personnel have
access to that shell, where the environment variable is set.
```

file:*file_path*

You must create a text file that contains only the password itself, store that file on a file system accessible to the Schema Repository process, and ensure the security of that file in such a manner to prevent unauthorized users from viewing its contents.

pass:*password*

Warning: With the pass form, the password remains in the process command line, which is visible to any system user using the command to display running process information. Do *not* use this form for production deployments.

Tibschemad Authorization

The Schema Repository can be run without authorization (`auth.type none`) or using role-based authentication using OAUTH2 tokens (`auth.type oauth2`). The default authorization type is `oauth2`.

Note: The Schema Repository only supports access tokens in the form of signed JWTs with asymmetric validation keys. Unsigned JWTs, or JWTs with symmetric validation keys are not supported.

If `tibschemad` is run with authorization `none`, any client application can perform any REST-based operation provided by `tibschemad`. The concept of `role` is a `tibschemad`-defined parameters which must be part of the OAUTH2 token provided by a client application.

Access Tokens for Incoming Connections

Validation of REST requests against the rules defined in the policy file using the “role” specified in the client-supplied OAuth2 token. Depending on the role and the policies as specified in the policy file, access can be controlled at both an action and subject level thereby controlling the operations a client application may perform. If a client attempts to make a REST call which is not permitted for the role provided, `tibschemad` will return a response code of `403 (Forbidden)`. If a client attempts to make a REST call without a bearer token, `tibschemad` will return a response code of `401 (Unauthorized)`.

Example of `tibschemad`-defined parameters that must be part of the OAuth2 token.

Apache Avro Client Library Configuration Options

It is important to configure the Schema Repository correctly and manage changes so schemas are always application ready.

You can set the following Apache Avro client library configuration options programmatically with Java properties for either an Apache Kafka producer or consumer.

Access Configuration Properties

ftl.servers

```
Type:             String

Pipe-delimited list of FTL Server URLs.

 Default:       localhost:8081.
```

Security Configuration Properties

The `ftl.trust` properties are optional but are strongly recommended for security.

ftl.trust.type

```
Type:             String

FTL trust type ("file", "everyone", or "string", as per usual in other
products).
```

ftl.trust.file

```
Type:             String

 FTL trust file path.
```

ftl.trust.string

```
Type:             String

 FTL trust certificate PEM string.
```

Schema Registration Configuration Properties

schema.registration.strategy

```
Type:             String


 Valid values are:

   • auto (default) - If the latest version of the schema registered
     under the producer topic's subject does not match the schema of the
     message the producer is sending, or, if no schema is registered
     yet, then register a schema matching the producer's message before
     sending it.

   • none - Do not register a schema. If a matching schema does not exist
     for the producer's topic, then the producer fails to send.

   • latest - Do not register a schema. Instead, use the latest version
     of the producer topic's schema, even if it is not an exact match.
     If a schema does not exist for the producer's topic, then the
     producer fails to send.
```

Subject Configuration Properties

key.subject.name

Type: String

Subject in the Schema Repository that the client library registers new key schemas under.

Default: %t-key

Note: The application automatically replaces the special characters %t, %r, and %% with topic, name of schema, and literal % character respectively.

value.subject.name

Type: String

Subject in the Schema Repository that the client library registers new value schemas under.

Default: %t-value

Note: The application automatically replaces the special characters %t, %r, and %% with topic, name of schema, and literal % character respectively.

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The following documentation for this product is available on the TIBCO® Messaging - Schema Repository for Apache Kafka Product Documentation page:

- *TIBCO® Messaging - Schema Repository for Apache Kafka Release Notes*
- *TIBCO® Messaging - Schema Repository for Apache Kafka Installation*
- *TIBCO® Messaging - Schema Repository for Apache Kafka User Guide*

Other TIBCO Product Documentation

When working with TIBCO® Messaging - Schema Repository for Apache Kafka, you may find it useful to read the documentation of the following TIBCO product:

- TIBCO FTL® - Enterprise Edition: TIBCO's FTL server provides back-end Schema Repository storage.

How to Access Related Third-Party Documentation

When working with TIBCO® Messaging - Schema Repository for Apache Kafka, you may find it useful to read the documentation of the following third-party products:

- Apache Kafka documentation
- Apache Avro documentation

How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to TIBCO Community.

Legal and Third-Party Notices

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

TIBCO, the TIBCO logo, the TIBCO O logo, TIBCO FTL, TIBCO eFTL, TIB, and Information BUS are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (https://www.cloud.com/legal) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: https://scripts.sil.org/OFL

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file

for the availability of a specific version of Cloud SG software on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at https://www.cloud.com/legal.