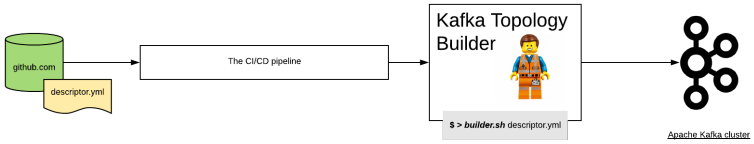
Kafka Topology Builder Documentation

Pere Urbon-Bayes

Feb 27, 2021

Contents

Welcome to the Kafka Topology Builder documentation, in this site we recollected notes and guides to provide the beginners, but as well reference for the most experienced on implementing a gitops approach for operations in Apache Kafka.



CHAPTER 1

Getting started

If you have landed in this page is because you aim to get started with the Kafka Topology Builder. To get you started we recommend:

- Want a quickstart ? checkout our demo in https://github.com/purbon/kafka-topology-builder-demo.
- New to gitops and kafka, check the *Core Concepts* page.
- If you are ready to jump in, start with the *How to setup a full workflow (by example)*.

CHAPTER 2

Installation

To install the Kafka Topology Builder and enable your teams to use a gitops approach when managing Apache Kafka you need:

- A CI/CD server, for example Jenkins, but any other will serve the purpose
- A git repository where the Topology description will be stored
- And for sure a Kafka cluster :-)

See the *How to setup a full workflow (by example)* section to learn more how to setup the required components to enable the full workflow.

if you already have this components, you can install the topology builder agent:

- As an RPM package for RedHat/CentOS linux distributions.
- As a DEB package for Debian based OS.
- As tar.gz source package.
- As well in the form of a docker image, available from docker hub.

You always can self build this package, all information is available from here. Users can download the latest "official" release artifacts directly from the download page here.

CHAPTER 3

Help?

If case you require any help regarding the usage or development of the Kafka Topology Builder, don't hesitate to contact our gitter community. In the future, when there is the request we might as well create a mailing list or other method of communication that help people using this project.

CHAPTER 4

Contents

4.1 Core Concepts

In this page you will find a description of the core concepts necessary to move on with implementing a gitops approach for automating Kafka operations.

4.1.1 What is gitops?

GitOps is a paradigm or a set of practices that empowers developers to perform tasks which typically fall under the purview of IT operations. GitOps requires us to describe and observe systems with declarative specifications that eventually form the basis of continuous everything.

Source https://www.cloudbees.com/gitops/what-is-gitops

So with this we bring into the table:

- Autonomy for teams, they can request and fulfil their own ops needs independently.
- A declarative approach, users describe what they need and not how it is implemented, facilitating the solution of problems, keeping detailed implementation abstracted.
- A centralised way to verify and control changes, providing an audit trail to fulfil change management requirements.
- Have a centralised source of truth where the current state of the platform is represented.

4.1.2 What do I need to use gitops?

To setup a gitops flow you need three basic components:

- A git server, anything will work from Github, Gitlab or Bitbucket
- A CI/CD pipeline, for example the master Jenkins!
- And the Kafka Topology Builder project

This building blocks will build up necessary integration steps:

- In the git server, ex. Gitlab, we're going to store the topology definition, so how we want the cluster to look like in terms of topics, acls, etc.
- The git server will manage as well the change management flow using Change Requests (or pull request :-P)
- Jenkins will be the muscle of the operation, in charge of retrieving the git content, do an initial verification and run the topology builder to apply the changes in Apache Kafka.
- And The Kafka Topology Builder project, the responsible of interpreting the configuration files (topologies) and apply the changes directly into Apache Kafka and related components.

4.1.3 How are the users / dev teams leveraging gitops?

This flow is in place, at most the organisations, in order to enable individual cluster users request the resources they need for their project, and do this without very tied formal processes or even worst a Jira ticket :-).

The traditional flow for this approach is:

- A user (dev team) has the need of more topics (for example)
- This same user, does a new branch in git, and apply the required changes in the configuration file
- Once happy with the required changes, it will be submitting a change request (PR) to merge this changes to the master branch

then (in more controlled environments) :

- An authorised agent of change, for example people from the operations team, will do a manual review of the changes. If required will request changes to the original requester
- An automatic verification, using the CI/CD pipeline, is run automatically once the change request is created
- Once the agent is happy with the change request, it will merge it into master
- From master a CI/CD job will be triggered automatically to perform the requested changes into the cluster

NOTE: In environments, or organisations, less restrictive you might like to approve the change request automatically if the CI/CD pipeline verification is greed.

It should be noted as well that thanks to the change request mechanism offered by Gitlab, Github and Bitbucket, the change management chain and audit is fulfilled.

4.2 What can you do with Kafka Topology Builder

The main purpose of the Kafka Topology Builder is to help automating boring administrative tasks that each team dealing with Kafka have to do. This tasks usually are around Topics, Access Control, but as well on handling Schemas and more.

In this chapter we will introduce the different things one can configure in each topology file(s) and how to take advantage of the tool the best.

4.2.1 Themes

Use of custom configuration plans

It is possible with Kafka Topology Builder to utilise custom configuration plans. These plans will allow you to summarize a set of default configuration properties in a reusable label, making easier for users and as well operators to configure each topic.

You might wonder, why are Plans an interesting thing, this are a few ideas where you can use it:

- To define custom service levels, where you have topics with different retentions or that accept different message sizes.
- To group default configuration, allowing you as Kafka Topology user and operator to have an smaller file size.

And I am sure there is going to be more.

How can you take advantage of Plans?

Defining the your plans

As an operator of the Kafka Topology Builder you can define a set of custom plans, this would be a file that look like this:

```
plans:
  gold:
    alias: "gold"
    config:
      retention.ms: "5000"
      max.message.bytes: "7340116"
  silver:
    alias: "silver"
    config:
      retention.ms: "6000"
      max.message.bytes: "524294"
```

In this file, the reader can see two plans, *gold* and *silver* with different default configuration values.

Using plans in the Topology

Once the plan is defined, you can use it in your Topology files. This process would look like this:

```
context: "contextOrg"
source: "source"
projects:
  - name: "foo"
    topics:
      - name: "foo"
        config:
          replication.factor: "1"
          num.partitions: "1"
      - name: "fooBar"
        plan: "silver"
        config:
          replication.factor: "1"
      - name: "barFoo"
        plan: "gold"
        config:
```

(continues on next page)

(continued from previous page)

```
          replication.factor: "1"
      - name: "barFooBar"
        plan: "gold"
        config:
          replication.factor: "1"
```

In this Topology the topics *fooBar*, *barFoo* and *barFooBar* will be using custom plans.

Things to consider:

- Configuration values defined in the plan take precedence over the config values defined in the topic.
- A topic can have only a plan, without providing any configuration.
- The Plan label is optional, topics can use, or not, plans.
- If plans are defined in the Topology, but no proper definition file is passed, the tool will complain.

What about from the CLI

As a user of the Kafka Topology Builder CLI, if interested to use Plans you can pass a file using the dedicated parameter. An example call will look like:

```
$>  kafka-topology-builder.sh --brokers localhost:9092  \
              --clientConfig example/topology-builder.properties \
              --topology example/descriptor-with-plans.yaml  \
              --allowDelete
```

Controlling the access

ACLs

In the topology descriptor files users can create permissions for different types of applications, Consumers, Producers, Kafka streams apps or Kafka Connectors. With this roles, users can easily create the permissions that map directly to their needs.

Consumers

As a user you can configure consumers for each project. Consumer have a principal and optionally a consumer group name. The consumer group ACL is by default defined for all groups (“*”). Users can customize this ACL by defining a *group* attribute for each consumer.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      consumers:
        - principal: "User:App0"
```

Consumer definition with principal “User:App0” and without an specific consumer group, for this configuration an ACL will be created to accept any consumer group.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      consumers:
        - principal: "User:App0"
          group: "foo
```

Consumer definition with principal "User:App0" and consumer group name "foo".

In the default mode the KTB will create dedicated ACL for each user and topic pair. For organisations that aim not to have dedicated pair of rules the KTB offer the option to optimise the number of ACLs using prefixed rules.

The optimised ACLs/RBAC can be enabled using the *topology.acls.optimized* configuration property.

Producers

As a user of KTB you can configure the required set of producers for your application.

Producers have a principal.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      producers:
        - principal: "User:App0"
```

In the default mode the KTB will create dedicated ACL for each user and topic pair. For organisations that aim not to have dedicated pair of rules the KTB offer the option to optimise the number of ACLs using prefixed rules.

The optimised ACLs/RBAC can be enabled using the *topology.acls.optimized* configuration property.

Streams

Users can also setup Kafka Streams applications. Each one of them will be composed of a principal and a list of topics that this principal needs to read and write. The principal is the user used by the streams app to connect to Kafka. You can also optionally specify the *applicationId*.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      streams:
        - principal: "User:App0"
          topics:
            read:
              - "topicA"
            write:
              - "topicB"
```

KTB will create the necessary ACLs for reading and writing topics, as well as ACLs needed by the app to create/manage internal topics. The ACLs for the consumer group and for internal topic creation are prefixed. The

resource name (prefix) is by default the topic name prefix in the project. For the example above the prefix will by default be "context.source.foo".

As you see in the next example this can be overridden by specifying an *applicationId* in the topology.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      streams:
        - principal: "User:App0"
          applicationId: "streamsApplicationId"
          topics:
            read:
              - "topicA"
            write:
              - "topicB"
```

When the *applicationId* is specified this is used as the resource prefix in the ACLs for consumer groups and internal topics for the streams app. In the above example the prefix will be "streamsApplicationId".

Connectors

In a similar fashion as with the previous roles, users can setup specific Kafka Connect setups. Each one of them will be composed of a principal, this would be the user used by the connector to connect to Kafka and a list of topics that this principal needs to read or write to, remember Connectors can either read (Sink) or write (Source) into Apache Kafka and they do it to topics.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      connectors:
        - principal: "User:Connect1"
          connectors:
            - "jdbc-sync"
            - "ibmmq-source"
          topics:
            read:
              - "topicA"
              - "topicB"
        - principal: "User:Connect1"
          group: "group"
          status_topic: "status"
          offset_topic: "offset"
          configs_topic: "configs"
          topics:
            write:
              - "topicA"
              - "topicB"
```

If you are having more than one Kafka Connect cluster you can specify a custom group, status, offset and config topics.

When using RBAC, you can add under each principal the connectors it can use and this principals will only have visibility over them.

Schema Registry

Under the platform section users can define the permissions required for handling Schema Registry clusters, optionally you can configure the topic name and group used for the communication.

```
---
  context: "context"
  platform:
      schema_registry:
        instances:
          - principal: "User:SchemaRegistry01"
            topic: "foo"
            group: "bar"
          - principal: "User:SchemaRegistry02"
            topic: "zet"
        rbac:
          Operator:
            - principal: "User:Hans"
            - principal: "User:Bob"
```

If you are using rbac, under the specific section users can attach their own cluster wide role principles.

What ACLs are created

Kafka Topology Builder will assign the following ACLs:

- each principal in the *consumers* list will get *READ* and *DESCRIBE* permissions on each topic. In addition *READ* access on every consumer group (by default) or the group specified in the topology.
- each principal in the *producers* list will get *WRITE* and *DESCRIBE* permissions on each topic. In addition if a *transactionId* is specified a WRITE and DESCRIBE ACL is created on the transactionId resource. And if either *transactionId* or *idempotence* is specified for the producer the IDEMPOTENT_WRITE ALLOW acl is created.
- each principal in the *streams* list will get
  - *READ* access on every topic in its *read* sub-object
  - *WRITE* access on every topic *write* sub-object
  - *ALL* access on every topic starting with the fully-qualified project name (by default) or the given applicationId. These are *PREFIXED* ACLs.
  - *READ* access on consumer groups starting with the fully-qualified project name (by default) or the given applicationId. These are *PREFIXED* ACLs.
- each principal for a connector will get
  - read and write access on the corresponding *status_topic*, *offset_topic*, and *config_topics* (*LITERAL* ACLs)
    - these fields default to *connect-status*, *connect-status*, and *connect-configs*. Hence access to these topics will be granted to the Connect principal if the fields are not explicitly given.
  - *CREATE* access on the cluster resource
  - *READ* access on every topic in the corresponding *topics.read* subobject
  - *WRITE* access on every topic in the corresponding *topics.write* subobject
  - *READ* access on the group specified in the corresponding *group* field * if no *group* is specified, rights to *connect-cluster* will be granted

- the principal for a *schema_registy* platform component will be given *DESCRIBE_CONFIGS*, *READ*, and *WRITE* access to each topic.
- the principal for a *control_center* platform component will be given:
  - *DESCRIBE* and *DESCRIBE_CONFIGS* on the cluster resource
  - *READ* on every consumer group starting with the corresponding *appId* (*PREFIXED* ACLs)
  - *CREATE*, *DESCRIBE*, *READ*, and *WRITE* access on each topic starting with the corresponding *appId* (*PREFIXED*)
  - *CREATE*, *DESCRIBE*, *READ*, and *WRITE* access on the *_confluent-metrics*, *_confluent-command*, and *_confluent-monitoring* topics

Which ACLs does the user running Kafka Topology Builder need?

The principal which the Kafka Topology Builder uses to authenticate towards the Kafka cluster should have the following rights:

- *ALTER* on the cluster resource to create and delete ACLs
- *DESCRIBE* on the cluster resource
- the following operations be allowed for topic resources prefixed with the current context:
  - *ALTER_CONFIGS*, *CREATE*, and *DESCRIBE*
  - *ALTER* when changing the number of partitions should be allowed
  - *DELETE* when topic deletion should be allowed

See https://docs.confluent.io/current/kafka/authorization.html for an overview of ACLs. When setting up the topology builder for a specific context, prefixed ACLs can be used for all topic-level operations.

When using Confluent Cloud, a *service account* with the proper rights to run the topology builder for the context *samplecontext* could be generated as follows using the Confluent Cloud CLI *ccloud*:

```
ccloud service-account create sa-for-ktb --description 'A service account for the␣
↪Kafka Topology Builder'
# note the Id for the service account, we will use 123456 below

ccloud kafka acl create --allow --service-account 123456 --cluster-scope --operation␣
↪ALTER
ccloud kafka acl create --allow --service-account 123456 --cluster-scope --operation␣
↪DESCRIBE
ccloud kafka acl create --allow --service-account 123456 --topic samplecontext --
↪prefix --operation ALTER_CONFIGS
ccloud kafka acl create --allow --service-account 123456 --topic samplecontext --
↪prefix --operation CREATE
ccloud kafka acl create --allow --service-account 123456 --topic samplecontext --
↪prefix --operation DESCRIBE
ccloud kafka acl create --allow --service-account 123456 --topic samplecontext --
↪prefix --operation ALTER
ccloud kafka acl create --allow --service-account 123456 --topic samplecontext --
↪prefix --operation DELETE
```

RBAC

Having multiple Kafka Connect clusters

A more than common scenario in many organisations is to have multiple Kafka Connect clusters. The Kafka Topology Builder will allow you to configure and manage them using a single Topology, using a descriptor yaml like this one:

```
---
  context: "context"
  projects:
    - name: "projectA"
      consumers:
        - principal: "User:App0"
        - principal: "User:App1"
      producers:
        - principal: "User:App3"
        - principal: "User:App4"
      connectors:
        - principal: "User:Connect1"
          group: "group"
          status_topic: "status"
          offset_topic: "offset"
          configs_topic: "configs"
          topics:
            read:
              - "topicA"
              - "topicB"
```

The reader can see with the previous YAML code block that *User:Connect1* will be authorized for a custom set of group, status, offset and configs topics. This future is very flexible as single topology files can be used to describe permission for multiple Connect clusters.

Access for specific Connectors

It is possible in RBAC to assign permission for a given principal to access a given set of Connectors. This is possible with the Kafka Topology Builder with a topology like the one below, where *User:Connect1* will have access to connectors *jdbc-sync* and *jdbc-source*.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      consumers:
        - principal: "User:App0"
        - principal: "User:App1"
      connectors:
        - principal: "User:Connect1"
          connectors:
            - "jdbc-sync"
            - "ibmmq-source"
          topics:
            read:
              - "topicA"
              - "topicB"
        - principal: "User:Connect2"
          topics:
            write:
```

(continues on next page)

(continued from previous page)

```
            - "topicC"
            - "topicD"
```

Access for specific Schemas

It is possible in RBAC to assign permission for a given principal to access a given set of Schemas. This is possible with the Kafka Topology Builder with a topology like the one below, where *User:App0* will have access to schemas in subjects *transactions* and *User:App1* to subject *contracts*.

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      consumers:
        - principal: "User:App0"
        - principal: "User:App1"
      streams:
        - principal: "User:App0"
          topics:
            read:
              - "topicA"
              - "topicB"
            write:
              - "topicC"
              - "topicD"
      schemas:
        - principal: "User:App0"
          subjects:
            - "transactions"
        - principal: "User:App1"
          subjects:
            - "contracts"
```

Cluster wide roles

In the RBAC module users can add cluster wide roles to principals. This roles can be attached to each one of the clusters available in the confluent platform.

This functionality will, as of the time of writing this documentation, work for Kafka, Kafka Connect and Schema Registry clusters. It might be extended in the future for other clusters in the platform.

```
---
  context: "context"
  source: "source"
  platform:
      kafka:
        rbac:
          SecurityAdmin:
            - principal: "User:Foo"
          ClusterAdmin:
            - principal: "User:Boo"
      kafka_connect:
```

(continues on next page)

(continued from previous page)

```
        rbac:
          SecurityAdmin:
            - principal: "User:Foo"
      schema_registry:
        instances:
          - principal: "User:SchemaRegistry01"
            topic: "foo"
            group: "bar"
          - principal: "User:SchemaRegistry02"
            topic: "zet"
        rbac:
          Operator:
            - principal: "User:Hans"
            - principal: "User:Bob"
```

In the previous example the reader can see how to add cluster wide roles into each of the available clusters, all roles go under the rbac label.

NOTE: The syntax support having multiple schema registry instance where the reader can configure specific *schema topics* and *groups*. This capability allows a high degree of personalisation for the permissions being generated.

Managing Principals

One of the common actions required when managing a cluster is the operation of principals (aka Users in kafka). The KTB can help you manage this principals by parsing the involved topologies.

NOTE This feature is currently experimental, please use it with care and let us know anything that is missing.

How does it work

As this feature is considered experimental for now, the reader should first enable it, this is done by adding the _topology.features.experimental_ property in the configuration.

```
topology.features.experimental=true (by default all experimental features are␣
↪disabled)
```

Once enabled, this feature will run in between the TopicManager and the BindingsManager, so right before the Bindings are created all required principals are present in the cluster.

This process is done by parsing the topology description and extracting the principals, once done the tool will calculate which principals are required to be create and which ones are no longer necessary and can be disabled. For sure always if the related delete option is enabled.

If desired by organisational purposes a user can decide to filter witch Service Account principals can be managed, this is done using the *topology.service.accounts.managed.prefixes* configuration setting. Check config for details.

Principals Manager Providers

As a user of KTB you can select witch providers to use, they could be for example Confluent Cloud or SASL/SCRAM (once #2 is implemented), the tool will contact the server and manage the user creation.

In the current version you can only use the Confluent Cloud provider.

Confluent Cloud Provider

If the reader is using the Confluent Cloud, you can manage the principals described in the Topology. This operations are dependant on the *ccloud* CLI, as a user you should have this tool available in your system.

Another requirement is you should be logged in more details here. Once this is done, the KTB will rely on CLI to manage the principals.

Enabling principal translation

As a user of KTB with Confluent Cloud, you can chose to use the Service Account ID or the Service Account Name as a label describing your user. If you want the users to be managed by Kafka Topology Builder you have to use the principals by your Service Account name, the Topology will look like this:

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      consumers:
        - principal: "User:ApplicationName"
```

internally this would be registered in Confluent Cloud and use internally the Service Account ID for managing all required ACLs.

As this is an experimental feature you will be required to enable it.

```
topology.translation.principal.enabled=true (by default all experimental features are␣
↪disabled)
```

Managing Schemas

Because not only from Topics and Access live the Kafka team, with the Kafka Topology Builder you can manage as well your schemas. This functionality can be very useful for administering your cross environment, but as well to register specific schemas per topic.

An example topology for managing schemas would look like this (*only the topics section*).

```
---
  context: "context"
  projects:
    - name: "projectA"
      topics:
        - name: "foo"
          schemas:
            value.schema.file: "schemas/foo-value.avsc"
          config:
            replication.factor: "1"
            num.partitions: "1"
        - name: "bar"
          dataType: "avro"
          subject.name.strategy: "TopicRecordNameStrategy"
          schemas:
            - key.schema.file: "schemas/foo-key.avsc"
              value.schema.file: "schemas/foo-value.avsc"
```

(continues on next page)

(continued from previous page)

```
            value.record.type: "foo"
          - key.schema.file: "schemas/bar-key.avsc"
            value.schema.file: "schemas/bar-value.avsc"
            value.record.type: "bar"
          - key.schema.file: "schemas/zet-key.avsc"
            value.schema.file: "schemas/zet-value.avsc"
            value.record.type: "zet"
        config:
          replication.factor: "1"
          num.partitions: "1"
```

If using an example like this, for the topic _bar_ there is going to be an schema registered for key and value. Currently the schemas are managed as files and need to be accessible for the KTB tool, for example inside the git repository.

A topology with absolute schema path would look like this:

```
---
  context: "context"
  projects:
    - name: "projectA"
      topics:
        - name: "foo"
          schemas:
            value.schema.file: "/kafka/schemas/foo-value.avsc"
          config:
            replication.factor: "1"
            num.partitions: "1"
```

NOTE: The path for the files is relative to the location of the of the topology.

NOTE: Keep in mind that for a use case like in the example above you have to define the value for *schema.registry.url* in your properties file.

Where should you put your schema files

We recommend keeping your schemas relative to the location of your topologies files as it is easier to map all things bundler near by than in distant locations.

However, if preferred for organisational purposes the reader can specify any absolute path in the topology and the tool will prefer it as an schema location.

NOTE: There is no way as of now to set a preferred location by configuration, so the user should write an absolute path where necessary in the topology file.

Multiple schemas support per topic

It is possible to setup multiple schemas per topic, this is a common use case when using a subject name strategy different than *TopicNameStrategy*.

If using multiple schemas, as it can be seen in the earlier example, the user must set the subject naming strategy for the topic to be one of the other options.

The user should as well set the *record.type* for each of the keys as this is a value required when configuring the final subject.

Support multiple formats

It is currently possible to support schema files with all formats currently available in the Confluent Schema Registry. As a user, you can set the format for each of the components in the schema section like this:

```
---
  context: "context"
  projects:
    - name: "projectA"
      topics:
        - name: "foo"
          config:
            replication.factor: "1"
            num.partitions: "1"
        - name: "bar"
          dataType: "avro"
          schemas:
            key.schema.file: "schemas/bar-key.avsc"
            key.format: "AVRO"
            value.schema.file: "schemas/bar-value.json"
            value.format: "JSON"
          config:
            replication.factor: "1"
            num.partitions: "1"
```

if the **format** keyword is not specified, the default value is *AVRO*.

Set the Schema Compatibility

In the Schema management section is possible to set the schema compatibility level like this:

```
---
  context: "context"
  projects:
    - name: "projectA"
      topics:
        - name: "foo"
          config:
            replication.factor: "1"
            num.partitions: "1"
        - name: "bar"
          dataType: "avro"
          schemas:
            value.schema.file: "schemas/bar-value.avsc"
            value.compatibility: "BACKWARD"
          config:
            replication.factor: "1"
            num.partitions: "1"
```

NOTE: The compatibility level will be set before summit the registered schema file, this is done like this to easy transitions and migrations.

Supported Schema Registry

There are multiple options when using an Schema Registry with Kafka, currently only the Confluent Schema Registry is supported.

Managing Topics and their Configuration

The first and foremost important thing we aim to manage is topics, setup the partitions count and replication factor and as well configure their specific characteristics.

Topic naming convention

Topic names will be chosen according to the scheme:

```
[context].[source].[project-name].[topic-name]
```

It is possible to give a finer structure to the topic names by specifying additional fields between the *company* and *projects* fields. Optionally, a *dataType* can be specified, which will be suffixed to the topic name. For example:

```
context: "context"
company: "company"
env: "env"
source: "source"
projects:
  - name: "projectA"
    topics:
      - name: "foo"
      - name: "bar"
        dataType: "avro"
```

will lead to topic names

```
context.company.env.source.projectA.foo
context.company.env.source.projectA.bar.avro
```

Partitions Count and Replication Factor

In following configuration the reader is seeing an example of how to create a topic with *replication.factor* and *num.partitions*, in that case all one.

```
---
  context: "context"
  projects:
    - name: "projectA"
      topics:
        - name: "foo"
          config:
            replication.factor: "1"
            num.partitions: "1"
```

Users can as well increase later the number of partitions and the Topology Builder will handle it properly.

Topic Level Consumers and Producers

It is possible to setup dedicated access control rules for specific topics instead of project scope. An example configuration would look like this:

```
---
  context: "context"
  projects:
    - name: "projectA"
      topics:
        - name: "foo"
          consumers:
            - principal: "User:App0"
          producers:
            - principal: "User:App1"
          config:
            replication.factor: "1"
            num.partitions: "1"
```

This type of Access Control rules allow the reader to setup dedicated access to single topics, without giving global project access.

Handling Configuration

For each topic, under the configuration attribute, it is possible to define the map of custom broker side configurations for the topic.

KTB is going to take care to apply the necessary changes and remove the ones that are not necessary anymore.

4.3 Understanding the descriptor files

As a descriptive tool the KTB uses a file (or set of files) to record how the cluster should look like, this files are call descriptors.

Because users might have complex deployments, the structure is flexible enough to accommodate them, we will cover them in this section.

4.3.1 The File format

Currently the tool supports reading YAML and JSON files. **Note** that all topologies will need to be using the same file format, a mixture is not supported.

In this guide we will use yaml as the core format, for JSON examples please refer to the examples directory.

The file type is configured using the *topology.file.type* configuration variable.

```
` topology.file.type=JSON `
```

4.3.2 Getting started

The KTB files use a common structure for describing the cluster entities, this attributes are setup around:

- **context** : It is commonly used to describe where a collection of entities are coming from. This value could be for example a team name, a line of business or simply the origin of the topics (the data center).

This attribute is used as a primary key to group all the entities.

- **project**: In each *context* there could be one or more projects, this attribute is the next level of personalisation as it is usually used to group all final entities such as permissions (acls/rbac), topics and schemas. Each Topology can have multiple of them.

In between the *context* and the *project* attribute the user can define a free list of attributes in form of a key and value. This list of attributes is going to be listed, in order, by default during the topic name composition in between the context and the project attribute.

A descriptor header like:

```
---
  context: "context"
  source: "source"
  projects:
    - name: "foo"
      topics:
        - name: "foo"
          config:
            replication.factor: "1"
            num.partitions: "1"
        - dataType: "avro"
          name: "bar"
          config:
            replication.factor: "1"
            num.partitions: "1"
```

will by default create topic names with the prefix *context.source.foo*, in detail this topology will create two topics.

- *context.source.foo.foo* with one partition and a replication factor of one.
- *context.source.foo.bar.avro* with a single partition and a single replica.

The separator and order of attribute can be personalised using the KTB configuration file. The relevant properties are:

- **Property**: *topology.topic.prefix.format*, to set the full topic naming format.
- **Property**: *topology.project.prefix.format*, to set the project level name format, it should be a subset of the previous one.
- **Property**: *topology.topic.prefix.separator*, to select a custom separator between attributes.

more details can be found in the *Important configuration values* section where the most important configuration details are explained.

4.3.3 Manage only topics, the optional files

Not all the attributes are mandatory in the descriptor file, it is currently possible to:

- Have a file with only topics, so no acls are defined using the abstractions provided by the consumers, producers, streams, etc attributes.
- Build a topology with partial acls, if you are not using any stream application, there is no need to define it, same for other access control properties.
- When defining a topic it is possible to use: * *dataType* when as a user it is aimed to specify the data type of the topic. * *schemas* if the reader is interested to register schemas for the topic.

4.4 How can you run Kafka Topology Builder

Kafka Topology Builder is available to be integrated using different artifacts:

- As a **CLI tool**, available to be installed using rpm, deb and tar.gz packages from github.
- As a **docker image**, available from docker hub.

NOTE: In the future it will also be available as a jar to be integrated in other libraries.

4.4.1 Running KTB as a CLI command

If you are using the CLI tool, you an use the *–help* command to list the different options available.

```
$> kafka-topology-builder.sh  --help
    usage: cli
      --allowDelete          Permits delete operations for topics and
                             configs. (deprecated, to be removed)
      --brokers <arg>        The Apache Kafka server(s) to connect to.
      --clientConfig <arg>   The AdminClient configuration file.
      --dryRun               Print the execution plan without altering
                              anything.
      --help                 Prints usage information.
      --quiet                Print minimum status update
      --topology <arg>       Topology config file.
      --version              Prints useful version information.
```

The most important ones are:

- *–brokers*: This is an optional parameter where the user can list the target Kafka cluster urls.
- *–clientConfig*: As other tools, the Kafka Topology Builder needs it's own configuration. In this parameter users can pass a file listing all different personalisation options.
- *–dryRun*: When as a user, you don't want to run the tool, but instead see what might happen. This option is very useful to evaluate changes before applying them to the cluster.
- *–topology*: This is where you will pass the topology file. It can be either a single file, or a directory. If a directory is used, all files within are going to be compiled into a single macro topology.
- *–version*: If you wanna know the version you are running.
- *–allowDelete*: By default the KTB will not make any destructive operations. If as a user, you allow the tool to update the cluster,

for example by deleting unnecessary ACLs, or topics, you need to pass this option.

4.4.2 Running the KTB as a docker image.

As explained earlier, users can run the tool as well directly as docker images. An example command for this function will look like this:

```
$> docker run -t -i \
      -v /Users/pere/work/kafka-topology-builder/example:/example \
      purbon/kafka-topology-builder:latest \
      kafka-topology-builder.sh \
      --brokers pkc-4ygn6.europe-west3.gcp.confluent.cloud:9092 \
```

(continues on next page)

(continued from previous page)

```
    --clientConfig /example/topology-builder-with-schema-cloud.properties \
    --topology /example/descriptor.yaml -quiet
```

CLI options are all available here. Available image tags can be found at docker hub.

4.5 Handling delete in the Kafka Topology Builder

As the reader might already be aware, the Kafka Topology Builder is capable of handling deletes for you. This way the project can ensure that the final state of the cluster is consistent with the declarative state in the topology descriptors.

However in some situations having delete enabled might not be what you are looking for. For this reason the tool allows you to control it in full granularity.

4.5.1 Delete flag in the CLI

The CLI provide you currently with a global DELETE flag, this would allow deletion of all resources controlled by the Kafka Topology builder. The user can set this flag (*–allowDelete*) from the CLI as described.

```
$> kafka-topology-builder.sh  --help
    usage: cli
      --allowDelete          Permits delete operations for topics and
                             configs. (deprecated, to be removed)
      --brokers <arg>        The Apache Kafka server(s) to connect to.
      --clientConfig <arg>   The AdminClient configuration file.
      --dryRun               Print the execution plan without altering
                              anything.
      --help                 Prints usage information.
      --quiet                Print minimum status update
      --topology <arg>       Topology config file.
      --version              Prints useful version information.
```

NOTE: This is a global flag, allowing/deny all delete operations.

By default the value for this variable is false, so the Kafka Topology Builder will **not** delete any resource.

4.5.2 Granular delete flags

There could be situations when the reader aim to :

- Control delete operations in a more granular way, for example allow delete of bindings (Acls/RBAC) but not topics.
- Control the delete via ENV variables. This could be very handy when doing CI/CD integrations and setting variables over the pipeline executions.

This can be done in KTB by using the capabilities provided by the configuration library in use. The tool allows the user to set:

- A configuration variable to allow/deny the delete operations.
- Use an ENV variable as a first priority citizen to handle this operation

NOTE ENV variables take priority over other ways to set a config value.

Topics deletion flag

The user can control topic deletion by:

- setting the *allow.delete.topics* configuration in the provided file to the tool.
- set the ENV variable *ALLOW_DELETE_TOPICS* when calling the tool from the CLI.

Bindings deletion flag

The user can control bindings deletion by:

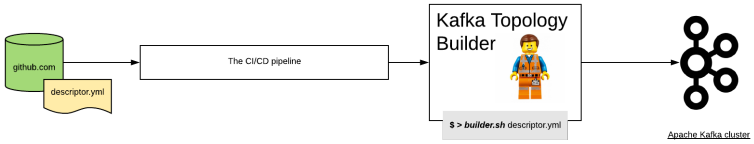
- setting the *allow.delete.bindings* configuration in the provided file to the tool.
- set the ENV variable *ALLOW_DELETE_BINDINGS* when calling the tool from the CLI.

4.6 How to setup a full workflow (by example)

This section describe the configuration steps need to setup a flow with the Kafka Topology Builder. For this example we're going to use:

- Github as the git server.
- Jenkins as the CI/CD pipeline.

Configuration is possible with other technologies such as Gitlab or Concourse for example.



4.6.1 Configure the git server

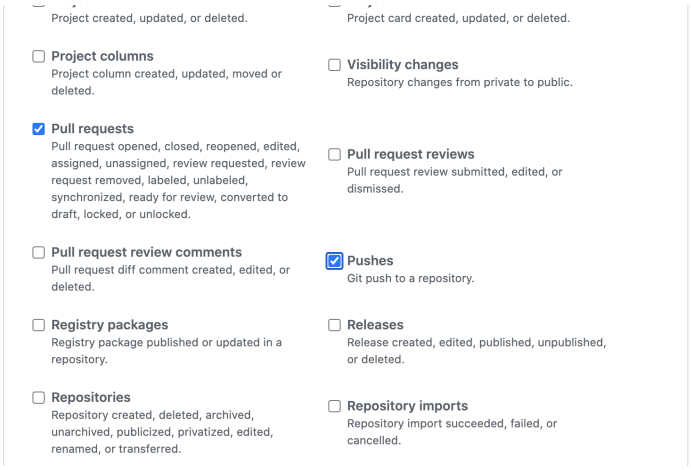
From the git server perspective you are going to need to setup the next steps:

- Block the master branch to push commits, so everyone is required to perform changes using a pull request.
- Configure a webhook to inform the CI/CD pipeline every time: * A pull request is created or changed. * A merge happened to the master branch.

For Github the webhook events to be selected are:

- *Pull requests*: _Pull request opened, closed, reopened, edited, assigned, unassigned, review requested, review request removed, labeled, unlabeled, synchronized, ready for review, converted to draft, locked, or unlocked._
- push event.

it should look like this:



4.6.2 Setup CI/CD

As mentioned earlier the specific setup for the CI/CD pipeline will vary. In this documentation we will enumerate a few of the steps necessary if you are using Jenkins.

How many pipelines/jobs do we need?

The first question is how many pipelines do you need? we recommend having two:

- The first dedicated to do change request (pull request) verifications
- A second one where we apply teh changes after merged

Setting up the PR verification pipeline

The pipeline responsible of running the test for each change request should look like this:

```
pipeline {

    agent {
     docker { image 'purbon/kafka-topology-builder:latest' }
    }

    stages {
        stage('verify-replication-factor') {
            steps {
                sh 'checks/verify-replication-factor.sh ${TopologyFiles} 3'
            }
        }
        stage('verify-num-of-partitions') {
             steps {
                sh 'checks/verify-num-of-partitions.sh ${TopologyFiles} 12'
            }
        }
    }
 post {
    success {
        withCredentials([string(credentialsId: 'my-github', variable: 'GITHUB_TOKEN
↪')]) {
```

(continues on next page)

(continued from previous page)

```
            sh './post-hook-success.sh'
        }
    }
    failure {
        withCredentials([string(credentialsId: 'my-github', variable: 'GITHUB_TOKEN
↪')]) {
            sh './post-hook-failure.sh'
        }
    }
 }
}
```

In the previous pipeline definition, using the Jenkins Pipeline DSL, we can notice a few relevant steps:

- Is using docker as an agent. We suggest this as a best practise, but it is possible as well to run this with any agent available that has access to a host where the Kafka Topology Builder is installed.
- There are a few verification, or test, steps. This are checks that run automatically for every Pull Request.
- In the pipeline the reader can see the topology files are passes as jenkins parameters, see *${TopologyFiles}*
- An important post step is configured where the pipeline will inform back to the git server the result of the verification. This step needs access to each server token, for the case of this pipeline a previously configured github token.

Setting up the main pipeline

The main pipeline should look like this:

```
pipeline {

    agent {
        docker { image 'purbon/kafka-topology-builder:latest' }
    }

    stage('run') {
      steps {
          withCredentials([usernamePassword(credentialsId: 'confluent-cloud ',
           usernameVariable: 'CLUSTER_API_KEY', passwordVariable: 'CLUSTER_API_SECRET
↪')]) {
            sh './demo/build-connection-file.sh > topology-builder.properties'
          }
          sh 'kafka-topology-builder.sh  --brokers ${Brokers}
          --clientConfig topology-builder.properties --topology ${TopologyFiles}'
        }
      }
    }
}
```

As the reader can see, the main responsibility of this pipeline is to apply the changes to the cluster by calling the kafka topology builder tool.

NOTE: The change request has been previously validated by an agent, and as well using the verifications pipeline.

As in the previous pipeline, we should note here the relevant steps:

- As a first step, the pipeline should take the parameters passed and build the properties file necessary for the topology builder to work.

- Next is, just call the *kafka-topology-builder* script with the required parameters (note, brokers and topology files are configured as job parameters) and let the changes go.

4.6.3 Using it in the day to day

As a development team, or basically as a user of the kafka infra, you will be provided with an one or multiple yaml files (the descriptors), usually they will be hosted in a shared repository but they could be as well in your own project repo.

Following your very same project setup, for example by using the GitLab flow where you will have environment branches you can expect to find a descriptors in each environment branch.

As a user, when you require a new topic, configuration or user permission, you will simply need to:

- create a new branch.
- alter the required files (the descriptors)
- create a pull request (PR) and request a review by a peer.
- Once the PR is approved, it will be merged.

Once the PR is merged, the peer jenkins job will pick up the files and apply the required changes directly to your shared infra.

4.6.4 Taking advantage of the Kafka topology Builder across environments

As introduced in the previous section, in any software project, there are many environments. This environments could be:

- test, where users run they specific story development tests.
- staging, or pre production, where the integration and smoke test are executed.
- production, where basically stuff runs.

If your project is following the Gitlab flow or anything similar, you might be having environment branch, one per each one in your setup.

Moving changes across environment will be as easy as following the same approach you are already taking for releasing commits across environments. This could be cherry pick, pull requests, etc.

NOTE: as a team, you might like to move towards a more controlled setup, going from complete freedom in the lower environment, to a more restricted setup as soon as you get into production. But with a flexible approach like gitops, continuous release is as well possible, the limits are only on yourself and the stability of your platform.

4.7 A collection of Kafka Topology Demos

If you're looking for building examples, you can find in this page provided examples build by community members.

4.7.1 A Jenkins CI/CD example

In case you're looking to use the Kafka Topology Builder in collaboration with Jenkins, you can find an example available for you here.

This project uses the Jenkins Pipeline future with the Docker execution agent to build and compose the two required pipelines, the one for pull requests and the one for applying the changes to the cluster.

4.7.2 A TravisCI integration

If you are are TravisCI user, you can find an example setup available for you here.

In this example you can see a similar setup as with the previously described Jenkins example, using docker and two different pipelines you can grasp a working example to start using the Kafka Topology Builder.

Thanks Nikoleta Verbeck for making this.

4.7.3 How to contribute a new page

If you are using the Kafka Topology Builder, integrating it with more CI/CD solutions let us know and we will add your example to this list. Examples help newcomers use it faster and are a great contribution to the project.

Thanks a lot.

4.8 Important configuration values

This page describe the most common configuration values for the Kafka Topology Builder, this values can be set within the topology-builder properties file.

4.8.1 Access control configuration

Configure the access control methodology.

Property: *topology.builder.access.control.class* **Default value**: "com.purbon.kafka.topology.roles.SimpleAclsProvider" **values**:

- RBAC: "com.purbon.topology.roles.RBACProvider"
- ACLs: "com.purbon.kafka.topology.roles.SimpleAclsProvider"

4.8.2 RBAC configuration

To configure RBAC, as a user you need to setup the access to your MDS server location, for this you need to setup the user and password to access it. An example configuration looks like this:

```
topology.builder.mds.server = "http://localhost:8090"
topology.builder.mds.user = "mds"
topology.builder.mds.password = "mds-secret"
```

A part from that, you need to setup the UUID for each of your clusters. This is one like this:

```
topology.builder.mds.kafka.cluster.id = "foobar"
topology.builder.mds.schema.registry.cluster.id = "schema-registry-cluster"
topology.builder.mds.kafka.connect.cluster.id = "connect-cluster"
```

4.8.3 Schema Management

If you plan to manage and deploy schemas with KTB, you must define the url to your Confluent Schema Registry as follows

```
schema.registry.url = "http://localhost:8081"
```

4.8.4 Topology Builder backend usage and selection

It is necessary for the topology builder to keep some state, for example for situations when the tool needs to decide what ACLs to remove. As well this property is important when the tool does not manage all topics in the cluster, so it is important to know it context.

The default implementation is a File, however is possible ot use other systems. To configure it you can use:

Configure the state management system. **Property**: *topology.builder.state.processor.class* **Default value**: "com.purbon.kafka.topology.backend.FileBackend" **values**:

- File: "com.purbon.kafka.topology.backend.FileBackend"
- Redis: "com.purbon.kafka.topology.backend.RedisBackend"

If you are using redis, you need to extend two other properties to setup the server location:

```
topology.builder.redis.host = "example.com"
topology.builder.redis.port = 6379
```

4.8.5 Customize the topic naming convention

A request, not either common, but necessary in some situations is to customize the topic naming convention. For this the Kafka Topology Builder offers the user the option to set it up using the configuration file.

This future accepts patterns using the jinja template formatting. *NOTE*: The properties used in the template need to exist in the topology as attributes.

As a user you can customize:

- **Property**: *topology.topic.prefix.format*, to set the full topic naming format.
- **Property**: *topology.project.prefix.format*, to set the project level name format, it should be a subset of the previous one.
- **Property**: *topology.topic.prefix.separator*, to select a custom separator between attributes.

4.8.6 Optimised number of ACLs and RBAC bindings

This property is used to reduce the number of ACLs, or RBAC bindings, created. In the normal operational mode, the KTB, will create direct pair of bindings for each user and topic. However for some organisations, it might be enough, to create an optimised list by using prefixed bindings.

Property: *topology.acls.optimized* **Default value**: "false"

An example configuration might look like this:

```
topology.acls.optimized=true
```

4.8.7 Internal topics prefixes

This is used to avoid deleting topics not controlled by topology builder.

Property: *kafka.internal.topic.prefixes* **Default value**: "_"

An example configuration might look like this:

```
kafka.internal.topic.prefixes.0=_
kafka.internal.topic.prefixes.1=topicPrefixA
kafka.internal.topic.prefixes.2=topicPrefixB
```

4.8.8 Topology level validations

It is now possible to define a list of validations to be applied to the desired Topology file.

As a user you can list the validations to be applied using the configuration property:

- **Property**: *topology.validations*

This property accepts the list of validations available in the class path. They will be applied in sequence as defined.

An example configuration might look like this:

```
topology.validations.0=topology.CamelCaseNameFormatValidation
topology.validations.1=topic.PartitionNumberValidation
```

Users can pull custom validation available from the class path.

4.8.9 Prevent ACL for topic creation for connector principal

By default KTB will create the ACLs needed for connectors to create their own topics (with CREATE ACL operation on the CLUSTER resource). You can override this behaviour by setting the config below to *false*. And instead create the needed topics with KTB.

Property: *topology.connector.allow.topic.create* **Default value**: true

An example configuration will look like this:

```
topology.connector.allow.topic.create=false
```

4.8.10 Retrieve topic management state from local controlled view

By default since it's creation KTB has been retrieving the state of topics from the target cluster, this means pulling the actual view directly from there (AK cluster) using AdminClient. However, this is not optimal when multiple teams use KTB as a multi tenant tool.

If you want to manage the current view of topics from the own KTB cluster state subsystem, you should use this property.

Property: *topology.state.topics.cluster.enabled* **Default value**: true

This property is fow the time being true as default (backwards compatible), however the local management system for topics will become the default in 2.0 and in 3.0 the management using the target cluster will be completely removed.

An example to use local topic management state will look like this:

```
topology.state.topics.cluster.enabled=false
```

4.8.11 Control allowed Service accounts to be managed by KTB

This property is used to control which Service Accounts are allowed to be managed by the KTB, this variable contains a list of allowed prefixes.

Property: *topology.service.accounts.managed.prefixes* **Default value**: "[]"

An example configuration might look like this:

```
topology.service.accounts.managed.prefixes.0=User:AService
topology.service.accounts.managed.prefixes.1=User:BService
```

If this prefix list is used, only service accounts that match the prefix will be ever processed, anything else will be ignored.