Tim van Baarsen

Following 171 Followers About

Programmatically create Kafka topics using Spring Kafka

Tim van Baarsen Apr 10, 2020 · 5 min read

Since the introduction of the AdminClient in the Kafka Clients library (version 0.11.0.0), we can create topics programmatically. Spring Kafka is leveraging the Kafka AdminClient to create Kafka topics programmatically even easier! Learn both about how to use it but also how to avoid some pitfalls 😉.

Spring Boot will autoconfigure a `AdminClient` Spring Bean in your application context which will automatically add topics for all beans of type `NewTopic`.

Spring Kafka version 2.3 introduced a `TopicBuilder` class to make the creation of such beans even more convenient!

used by Spring as a source of bean definitions.

All methods annotated with `@Bean` : will return an object that should be registered as a bean in the Spring application context. Spring Kafka will automatically add topics for all beans of type `NewTopic`

```
1  package io.stockgeeks.kafka.config;
2
3  import org.apache.kafka.clients.admin.NewTopic;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.Configuration;
6  import org.springframework.kafka.config.TopicBuilder;
7
8  @Configuration
9  public class KafkaTopicConfiguration {
10
11   @Bean
12   public NewTopic topicExample() {
13     return TopicBuilder.name("my-first-kafka-topic")
14       .partitions(6)
15       .replicas(3)
16       .build();
17   }
18 }
```

KafkaTopicConfiguration.java hosted with ♡ by GitHub view raw

Next to the name of the Kafka topic name you can specify:

- the number of partitions for the topic
- the number of replicas for the topic
- assign replicas

To create a **compact** Kafka **topic**:

```
1  package io.stockgeeks.kafka.config;
2
3  import org.apache.kafka.clients.admin.NewTopic;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.Configuration;
6  import org.springframework.kafka.config.TopicBuilder;
```

```
10
11    @Bean
12    public NewTopic compactTopicExample() {
13      return TopicBuilder.name("my-first-compact-kafka-topic")
14        .partitions(6)
15        .replicas(3)
16        .compact()
17        .build();
18    }
19  }
```

KafkaTopicConfiguration.java hosted with ♡ by GitHub view raw

Example of passing specific Kafka **topic configuration properties** (in this case to configure compression:

```
1   import org.apache.kafka.clients.admin.NewTopic;
2   import org.apache.kafka.common.config.TopicConfig;
3   import org.springframework.context.annotation.Bean;
4   import org.springframework.context.annotation.Configuration;
5   import org.springframework.kafka.config.TopicBuilder;
6
7   @Configuration
8   public class KafkaTopicConfiguration {
9
10    @Bean
11    public NewTopic topicWithCompressionExample() {
12      return TopicBuilder.name("kafka-topic-with-compression")
13        .partitions(6)
14        .replicas(3)
15        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
16        .build();
17    }
18  }
```

KafkaTopicConfiguration.java hosted with ♡ by GitHub view raw

Caveats and pitfalls

There are some caveats and pitfalls to take into account when creating topics programmatically. Let's go over them one by one:

- Not using Spring Boot?

- Replication factor in a single node Kafka Cluster for local development

Not using Spring Boot?

In case you are not using Spring Boot, you have to configure the KafkaAdmin bean yourself to automatically add topics for all beans of type `NewTopic`

```
1  package io.stockgeeks.kafka.config;
2
3  import org.apache.kafka.clients.admin.AdminClientConfig;
4  import org.springframework.beans.factory.annotation.Value;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7  import org.springframework.kafka.core.KafkaAdmin;
8
9  import java.util.HashMap;
10 import java.util.Map;
11
12 @Configuration
13 public class KafkaTopicConfiguration {
14
15   @Value(value = "${kafka.bootstrapServers:localhost:9092}")
16   private String bootstrapServers;
17
18   @Bean
19   public KafkaAdmin kafkaAdmin() {
20     Map<String, Object> configs = new HashMap<>();
21     // Depending on you Kafka Cluster setup you need to configure
22     // additional properties!
23     configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
24     return new KafkaAdmin(configs);
25   }
26 }
```

Automatically increases the number of partitions

From the Spring Kafka documentation:

> *If the broker supports it (1.0.0 or higher), the admin increases the number of partitions if it is found that an existing topic has fewer partitions than the* `NewTopic.numPartitions`.

- name **“already-existing-kafka-topic”**
- with three partitions

Show details about the topic using the command: `kafka-topics.sh` (Part of the Apache Kafka distribution)

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --zookeeper
zookeeper:2181 --topic already-existing-kafka-topic --describe
```

Output showing us the number of partitions is 3:

```
Topic: already-existing-kafka-topic     PartitionCount: 3
ReplicationFactor: 1    Configs:
        Topic: already-existing-kafka-topic     Partition: 0
Leader: 1001    Replicas: 1001  Isr: 1001
        Topic: already-existing-kafka-topic     Partition: 1
Leader: 1001    Replicas: 1001  Isr: 1001
        Topic: already-existing-kafka-topic     Partition: 2
Leader: 1001    Replicas: 1001  Isr: 1001
```

Now you configured six partitions for the topic using the TopicBuilder and start your application:

Although the topic already exists, the number of partitions of the topic is increased to six!

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --zookeeper
zookeeper:2181 --topic already-existing-kafka-topic --describe
```

Output showing the number of partitions increased:

```
Topic: already-existing-kafka-topic     PartitionCount: 6
ReplicationFactor: 1    Configs:
        Topic: already-existing-kafka-topic     Partition: 0
```

```
        Topic: already-existing-kafka-topic     Partition: 2
Leader: 1001    Replicas: 1001  Isr: 1001
        Topic: already-existing-kafka-topic     Partition: 3
Leader: 1001    Replicas: 1001  Isr: 1001
        Topic: already-existing-kafka-topic     Partition: 4
Leader: 1001    Replicas: 1001  Isr: 1001
        Topic: already-existing-kafka-topic     Partition: 5
Leader: 1001    Replicas: 1001  Isr: 1001
```

Be aware:

- If partitions are increased for a topic, and the producer is using a key to produce messages, the partition logic or ordering of the messages will be affected!
- The number of partitions for a Kafka topic can only be increased.

The default number of partitions and replication count in the TopicBuilder

By default, the values for both the **partitions** and **replicas** in the TopicBuilder are one! For a local development environment, this is a sensible default (because, in most cases, you just run a single node Kafka cluster). But production Kafka applications need topics with a proper number of partitions and replicas to be able to balance the load, scale, and be fault-tolerant.

There is no silver bullet for the number of partitions of your Kafka topic. It depends on your use case. On the Confluent blog, you can find a good read about how to choose the number of topic partitions.

If you would like to know more about the basics of partitions and replicas read my previous blog post: "Head First Kafka: The basics of producing data to Kafka explained using a conversation."

Replication factor in a single node Kafka cluster for local development

> *From the Kafka documentation: The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of servers for fault tolerance.*

The **replication factor** defines the number of copies for each message produced to a Kafka topic. The replication factor can be defined at the topic level (like we do here in the Java config). Replicas are distributed evenly among Kafka brokers in a cluster.

you will run into this error:

```
2020-04-10 10:27:59.925 ERROR 3013 --- [           main]
o.springframework.kafka.core.KafkaAdmin  : Could not configure
topics

org.springframework.kafka.KafkaException: Failed to create topics;
nested exception is
org.apache.kafka.common.errors.InvalidReplicationFactorException:
Replication factor: 3 larger than available brokers: 1.
```

Long story short: when you run a single node Kafka cluster for local development, there are no other brokers to replicate the data. So you can only configure a replication factor of one!

Since you don't want to configure a replication factor of one for non-development environments, you need to make the number of replicas configurable!

Takeaways from my blogpost

- Programmatically creating Kafka topics is powerful but be aware of the pitfalls.
- Plan the number of partitions and replicas for your topic ahead based on your use-case.
- For a local development single node Kafka cluster, you can only configure a replication factor of one.
- The use of the AdminClient might be restricted on your Kafka cluster that makes it not possible to programmatically creating Kafka topics

Keep on learning

Use useful links related to this blogpost:

- Javadoc Kafka AdminClient
- Spring Kafka Documentation about configuring topics
- Anatomy of a topic

Tap the 👏 button if you found this article useful!

Any questions or feedback? Reach out to me on Twitter: @TimvanBaarsen

Kafka Spring Spring Boot Spring Kafka Java