

Load Balancing-API Gateway-Kafka basics

MOSTAFA RASTGAR

JUNE 2019

Agenda

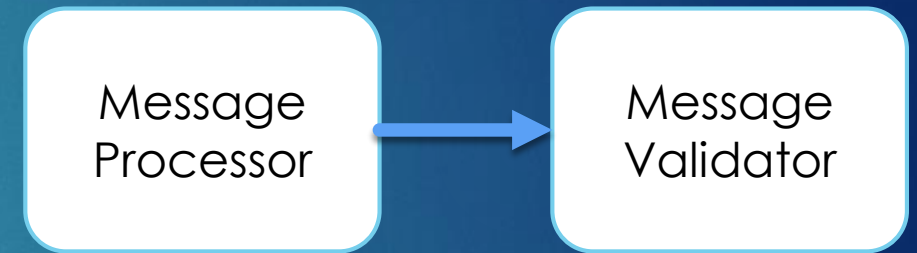
2

- ▶ Eureka Service Discovery
- ▶ Ribbon as a client-side load-balancer
- ▶ Zuul as an API Gateway
- ▶ Kafka basics

Eureka Service Discovery

3

- ▶ we should create two microservices and establish REST-based connection between them

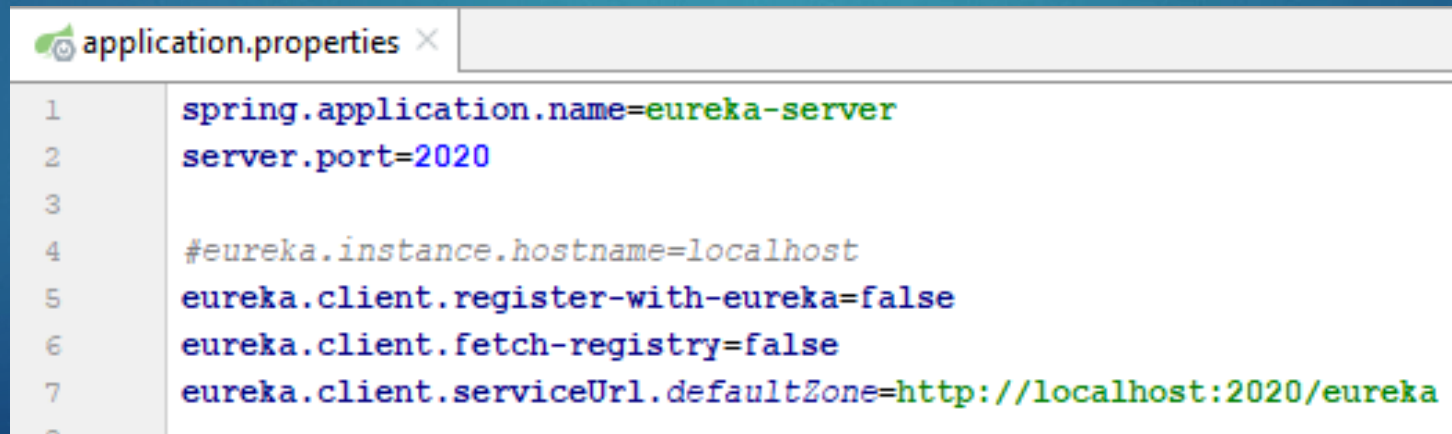


- ▶ That means every time there is a new instance of MV, we would need to change the configuration of MP. That's not cool
- ▶ we will use Eureka Naming Server to fix this problem
- ▶ Eureka is the Netflix Service Discovery Server and Client
- ▶ Bootstrapping Eureka Naming Server with Spring Initializer
 - ▶ @EnableEurekaServer
 - ▶ Launch Eureka Naming Server and visit the corresponding web console in: [http://localhost:\[PORT\]/](http://localhost:[PORT]/)

Eureka Service Discovery

4

- ▶ Include the 'spring-cloud-starter-netflix-eureka-server' starter in your eureka server
- ▶ To make your server secure, include 'spring-boot-starter-security' and provide HTTP Basic authentication
 - ▶ For more detail refer WebSecurityConfig class
- ▶ The other configurations are as follow:

A screenshot of a code editor window titled 'application.properties'. The window shows a list of configuration properties for a Spring Cloud Eureka Server. The properties are: 'spring.application.name=eureka-server', 'server.port=2020', a commented-out line '#eureka.instance.hostname=localhost', 'eureka.client.register-with-eureka=false', 'eureka.client.fetch-registry=false', and 'eureka.client.serviceUrl.defaultZone=http://localhost:2020/eureka'.

```
1 spring.application.name=eureka-server
2 server.port=2020
3
4 #eureka.instance.hostname=localhost
5 eureka.client.register-with-eureka=false
6 eureka.client.fetch-registry=false
7 eureka.client.serviceUrl.defaultZone=http://localhost:2020/eureka
8
```

Eureka Service Discovery

5

▶ Eureka Client

- ▶ To include the Eureka Client in your project, use the 'spring-cloud-starter-netflix-eureka-client' starter
- ▶ Use `@EnableDiscoveryClient` and `eureka.client.serviceUrl.defaultZone=http://[user:password@]eureka_host:eureka_port/eureka`
- ▶ When a client registers with Eureka, it provides meta-data about itself — such as host, port, health indicator URL, home page, and other details
- ▶ Eureka receives heartbeat messages from each instance belonging to a service
- ▶ The status page and health indicators for a Eureka instance default to `/info` and `/health` respectively, which are the default locations of useful endpoints in a Spring Boot Actuator application
- ▶ You need to change these if you use a non-default context path or servlet path. We should change two settings:
 - ▶ `eureka.instance.statusPageUrlPath=${server.servletPath}/info`
 - ▶ `eureka.instance.healthCheckUrlPath=${server.servletPath}/health`

Eureka Service Discovery

6

▶ Registering a Secure Application

- ▶ If your app wants to be contacted over HTTPS, you can set two flags in the `EurekaInstanceConfig`:
 - ▶ `eureka.instance.[nonSecurePortEnabled]=[false]`
 - ▶ `eureka.instance.[securePortEnabled]=[true]`
- ▶ Because of the way Eureka works internally, it still publishes a non-secure URL for the status and home pages unless you also override those explicitly
 - ▶ `eureka.instance.statusPageUrl=https://${eureka.hostname}/info`
 - ▶ `eureka.instance.healthCheckUrl=https://${eureka.hostname}/health`
 - ▶ `eureka.instance.homePageUrl=https://${eureka.hostname}/`

▶ Eureka's Health Checks

- ▶ By default, Eureka uses the client heartbeat to determine if a client is up
- ▶ Unless specified otherwise, the Discovery Client does not propagate the current health check status of the application, per the Spring Boot Actuator
- ▶ Consequently, after successful registration, Eureka always announces that the application is in 'UP' state
- ▶ The following example shows how to enable health checks for the client
 - ▶ `eureka.client.healthcheck.enabled=true`

Eureka Service Discovery

7

- ▶ self preservation and renew threshold
 - ▶ Every instance needs to renew its lease to Eureka Server with frequency of one time per 30 seconds, which can be define in 'eureka.instance.leaseRenewalIntervalInSeconds'
 - ▶ **Renews (last min):** represents how many renews received from Eureka instance in last minute
 - ▶ **Renews threshold:** the renews that Eureka server expects received from Eureka instance per minute
 - ▶ **SELF PRESERVATION MODE:** if Renews (last min) is less than Renews threshold, self preservation mode will be activated.

Eureka Service Discovery

8

- ▶ self preservation and renew threshold
 - ▶ **Question:** What is the purpose of the self preservation?
 - ▶ **Answer:** The SELF PRESERVATION MODE is design to avoid poor network connectivity failure. Connectivity between Eureka instance A and B is good, but B is failed to renew its lease to Eureka server in a short period due to connectivity hiccups, at this time Eureka server can't simply just kick out instance B. If it does, instance A will not get available registered service from Eureka server despite B is available. So this is the purpose of SELF PRESERVATION MODE, and it's better to turn it on.

Eureka Service Discovery

9

- ▶ self preservation and renew threshold
 - ▶ **Question:** why does a single Eureka server configured with `registerWithEureka: false` show up in the threshold count?
 - ▶ **Answer:** The minimal threshold 1 is written in the code. `registerWithEureka` is set to false so there will be no Eureka instance registers, the threshold will be 1. Moreover, in production environment, generally we deploy two Eureka servers and `registerWithEureka` will be set to true. So the threshold will be 2, and Eureka server will renew lease to itself twice/minute, so RENEWALS ARE LESSER THAN THRESHOLD won't be a problem.

Eureka Service Discovery

10

- ▶ self preservation and renew threshold
 - ▶ **Question:** For every client threshold count increases by +2. Is it the exact renew threshold?
 - ▶ **Answer:** `eureka.instance.leaseRenewalIntervalInSeconds` defines how many renews sent to server per minute, but it will multiply a factor 'eureka.server.renewalPercentThreshold' mentioned above, the default value is 0.85. Therefore, If you just want to deploy in demo/dev environment, you can set `eureka.server.renewalPercentThreshold` to 0.49, so when you start up a Eureka server alone, threshold will be 0
- ▶ **Summary:** The whole Eureka service settings have been configured with the most reliable default values. Therefore, it is highly recommended to leave them to their defaults

Ribbon as a client-side load-balancer

- ▶ Spring Netflix Eureka has a built-in client side load balancer called Ribbon. The corresponding starter is 'spring-cloud-starter-netflix-ribbon'
- ▶ In other words, spring-cloud-starter-netflix-eureka-client includes 'spring-cloud-starter-netflix-ribbon'
- ▶ Ribbon can automatically be configured by registering RestTemplate as a bean and annotating it with @LoadBalanced.

```
@LoadBalanced
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

- ▶ From now on, thanks to Eureka server, we can claim the appropriate service URL just by calling it with its registered service name such as: `restTemplate.getForEntity("http://[related service name]/[related web method]", [related response entity class]);`

Zuul as an API Gateway

12

- ▶ Netflix has more than 600 different microservices. Therefore, common aspects such as UI development, authentication, security, and monitoring - should be developed for each microservice team, so the same code has been replicated over 600 microservices?
- ▶ Moreover, changes in the authentication requirements will ripple over all services
- ▶ so this type of design is very error-prone and rigid

Zuul as an API Gateway

13

- ▶ Integration Patterns

- ▶ API Gateway Pattern

- ▶ Problem

- ▶ When an application is broken down to smaller microservices, there are a few concerns that need to be addressed:

1. How to call multiple microservices abstracting producer information.
2. On different channels (like desktop, mobile, and tablets), apps need different data to respond for the same backend service, as the UI might be different.
3. Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?
4. How to handle different type of Protocols some of which might not be supported by producer microservice.

- ▶ Solution

- ▶ An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above.

1. An API Gateway is the single point of entry for any microservice call.
2. It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.
3. It can fan out a request to multiple services and aggregate the results to send back to the consumer.
4. One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.
5. It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.
6. it can also offload the authentication/authorization responsibility of the microservice.

Zuul as an API Gateway

14

- ▶ Aggregator Pattern

- ▶ Problem

- ▶ We have talked about resolving the aggregating data problem in the API Gateway Pattern. However, we will talk about it here holistically. When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

- ▶ Solution

- ▶ The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:
 1. It is recommended if a composite microservice will make calls to all the required microservices, consolidate the data, and transform the data before sending back.
 2. An API Gateway can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.
 - ▶ It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

Zuul as an API Gateway

15

- ▶ Here, the Zuul (The Gatekeeper/Demigod) concept pops up.



Zuul as an API Gateway

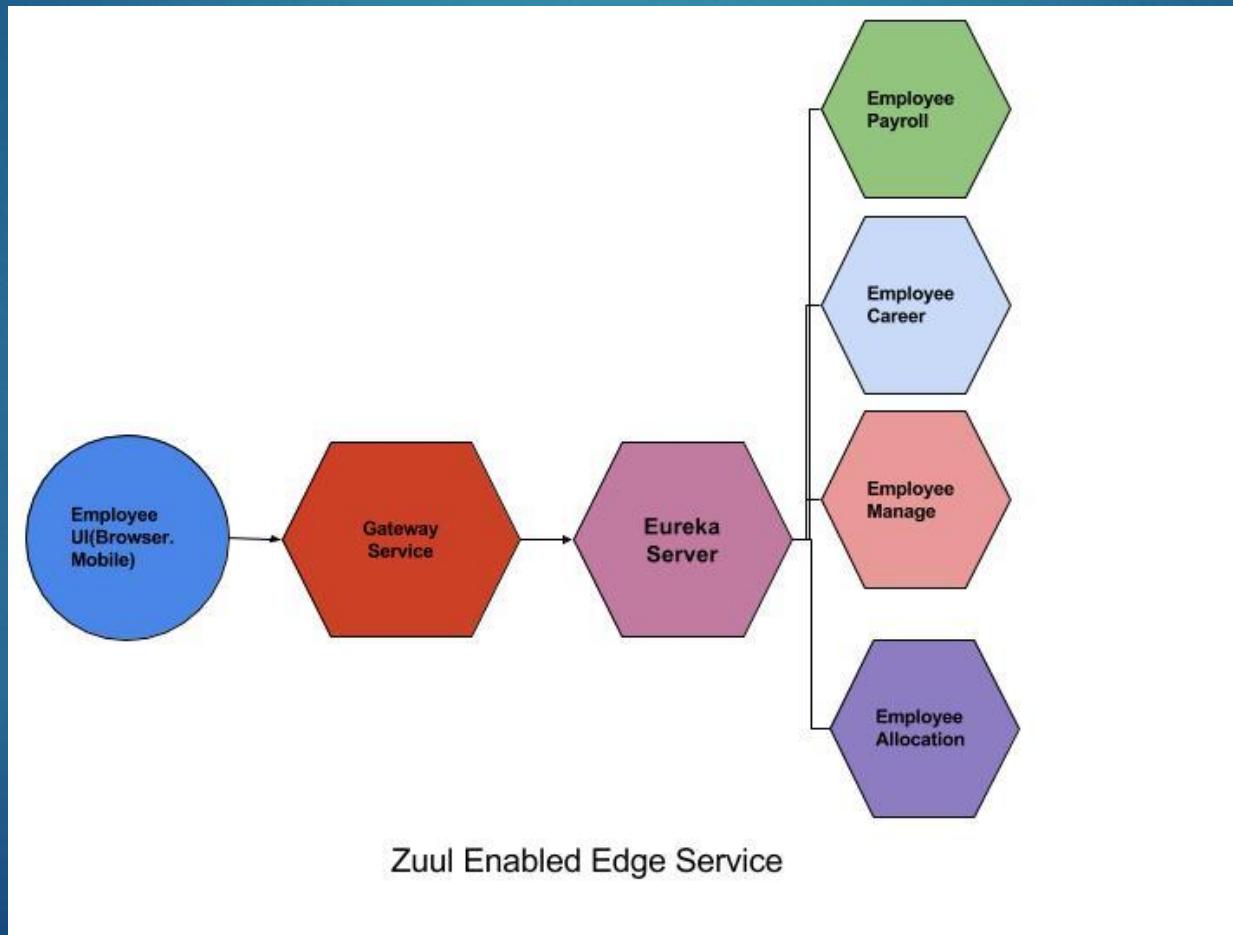
16

- ▶ Zuul acts as an API gateway or Edge service
- ▶ It receives all the requests coming from the outside of the microservices boundary then delegates the requests to internal microservices
- ▶ It also can adjust data regarding the various kinds of protocol related to different consumers
- ▶ The advantage of this type of design is that common aspects like authentication, and security can be put into a centralized service, so all common aspects will be applied on each request, and if any changes occur in the future, we just have to update the business logic of this Edge Service
- ▶ Also, we can implement any routing rules or any filter implementation

Zuul as an API Gateway

17

- We'll route requests to a REST Service discovered by Spring Cloud Eureka through Zuul Proxy



Zuul as an API Gateway

18

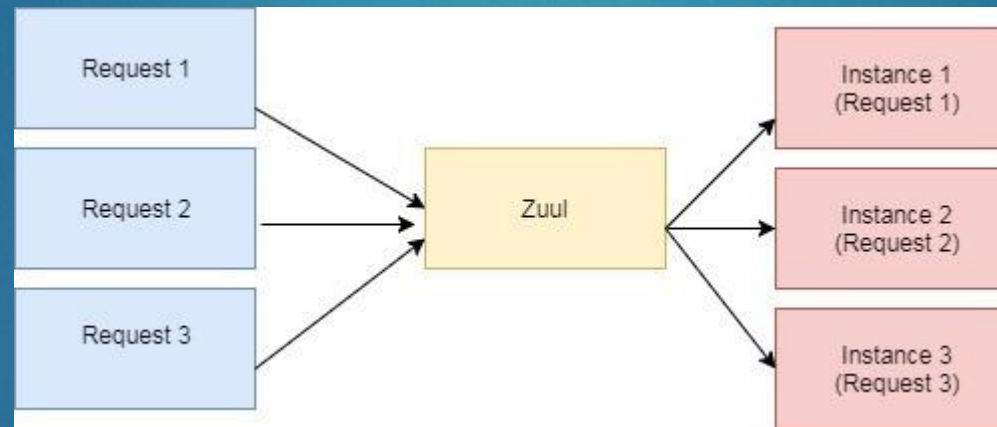
- ▶ How to use zuul?
 - ▶ add `@EnableZuulproxy` and `@EnableDiscoveryClient` on top of the zuul microservice application
- ▶ Important properties
 - ▶ `zuul.routes.employeeUI.serviceId=EmployeeDashBoard`
 - ▶ By this, we are saying if any request comes to the API gateway in form of `/employeeUI`, it will redirect to the `EmployeeDashBoard` microservice. So, if you hit the following URL: `http://[zuul ip]:[zuul port]/employeeUI/dashboard/1`, it will redirect to `http://[related microservice ip]:[related microservice port]/dashboard/1`.
 - ▶ `zuul.host.socket-timeout-millis=30000`
 - ▶ we instruct Spring Boot to wait for the response for 30000 ms until Zuul's internal Hystrix timeout will kick off and show you the error.

Zuul as an API Gateway

19

- ▶ Load Balancing with Zuul

- ▶ When Zuul receives a request, it picks up one of the physical locations available and forwards requests to the actual service instance

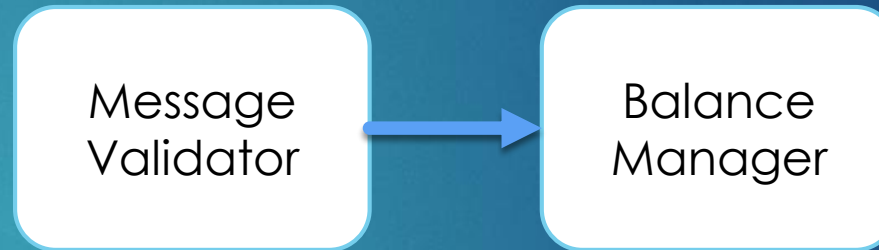


- ▶ Internally, Zuul uses Netflix Ribbon to look up for all instances of the service from the service discovery (Eureka Server)

Kafka basics

20

- ▶ We should stream payment orders to check available balances for each participants



- ▶ One of the best platforms (till now) is Kafka

Kafka basics

21

- ▶ What is Kafka?
 - ▶ Kafka is a distributed messaging system providing fast, highly scalable and redundant messaging through a pub-sub model
 - ▶ Kafka's distributed design gives it several advantages:
 - ▶ First, Kafka allows a large number of permanent or ad-hoc consumers
 - ▶ Second, Kafka is highly available and resilient to node failures and supports automatic recovery
 - ▶ In real world data systems, these characteristics make Kafka an ideal fit for communication and integration between components of large scale data systems

Kafka basics

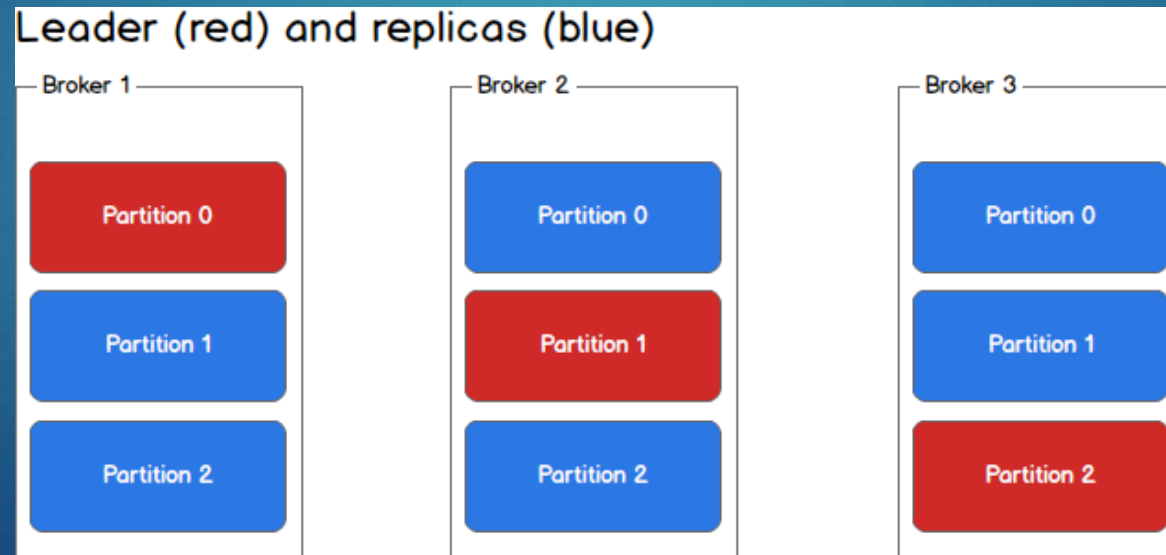
22

- ▶ Kafka Terminology
 - ▶ Topics
 - ▶ Producers
 - ▶ Consumers
 - ▶ Brokers
- ▶ Anatomy of a Kafka Topic
 - ▶ Kafka topics are divided into a number of partitions
 - ▶ Partitions allow you to parallelize a topic by splitting the data in a particular topic across multiple brokers
 - ▶ Consumers can also be parallelized so that multiple consumers can read from multiple partitions in a topic allowing for very high message processing throughput
 - ▶ Each message within a partition has an identifier called its offset
 - ▶ Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose, allowing consumers to join the cluster at any point in time they see fit
- ▶ Therefore, each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the message's topic, partition, and offset within the partition

Kafka basics

23

- ▶ Partitions and Brokers
 - ▶ Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic
 - ▶ All writes and reads to a topic go through the leader
 - ▶ If a leader fails, a replica takes over as the new leader



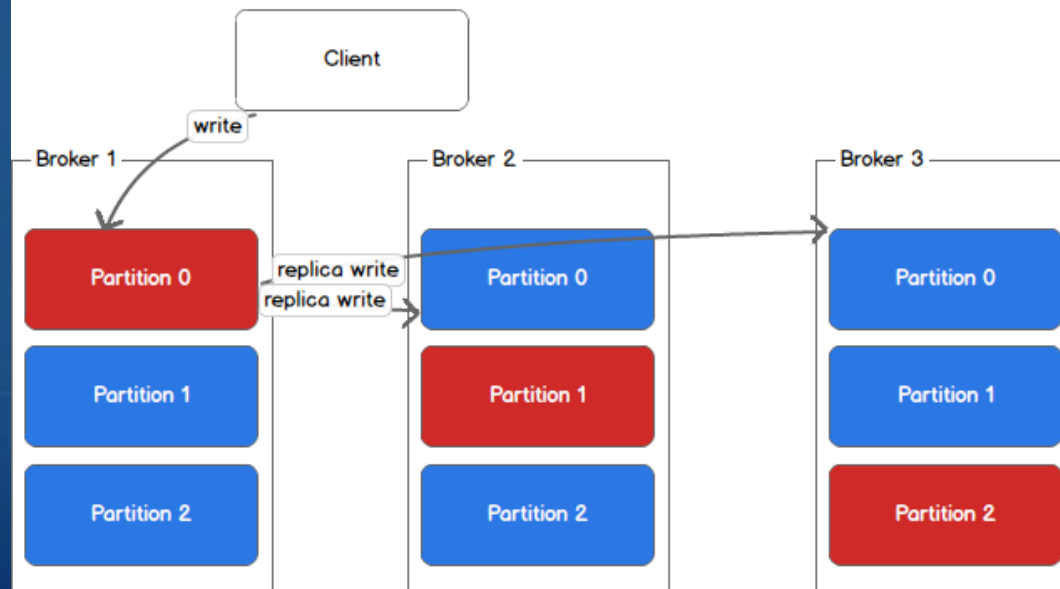
Kafka basics

24

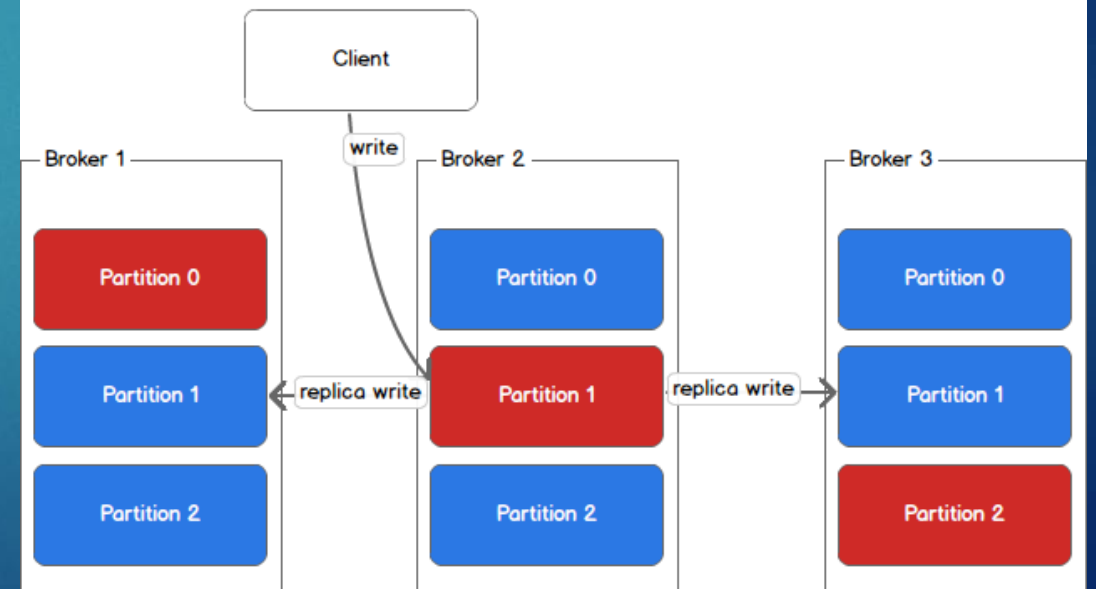
► Producers

- Producers write to a single leader
- this provides a means of load balancing production so that each write can be serviced by a separate broker and machine

Leader (red) and replicas (blue)



Leader (red) and replicas (blue)

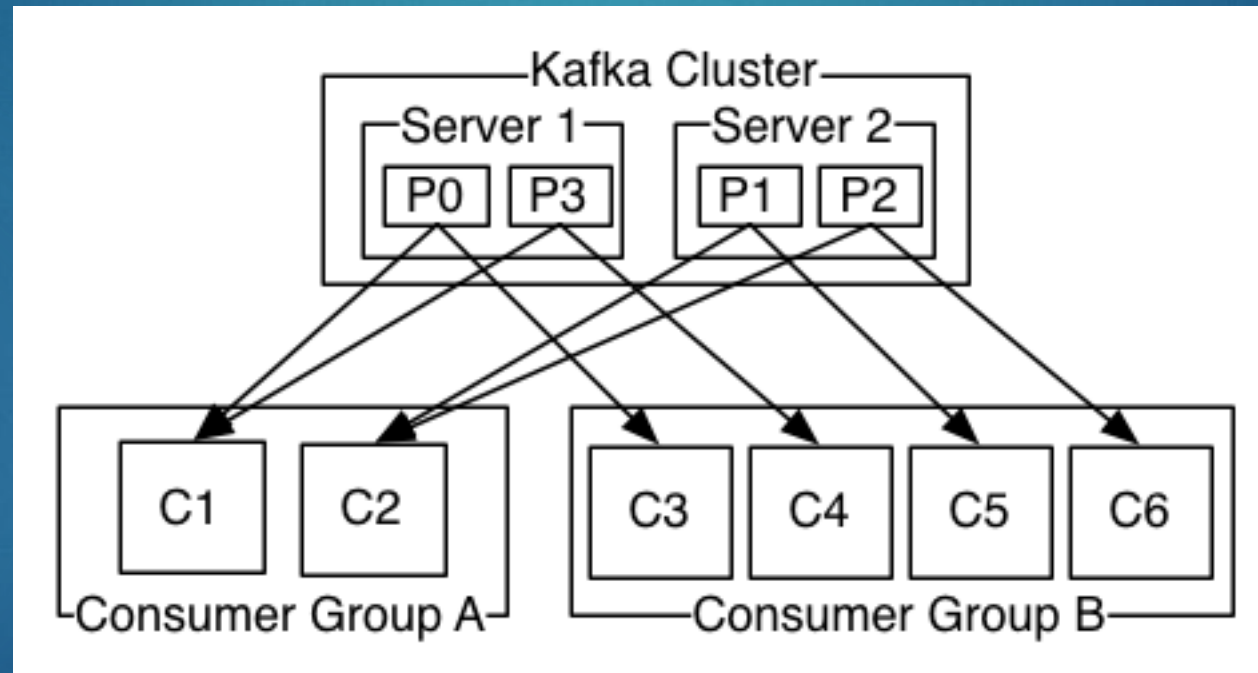


- ▶ Consumers and Consumer Groups
 - ▶ Consumers read from any single partition
 - ▶ allowing you to scale throughput of message consumption in a similar fashion to message production
 - ▶ Consumers can also be organized into consumer groups for a given topic
 - ▶ each consumer within the group reads from a unique partition and the group as a whole consumes all messages from the entire topic
 - ▶ Therefore:
 1. If you have more consumers than partitions then some consumers will be idle because they have no partitions to read from
 2. If you have more partitions than consumers then consumers will receive messages from multiple partitions
 3. If you have equal numbers of consumers and partitions, each consumer reads messages in order from exactly one partition

Kafka basics

26

- The whole ecosystem in a nutshell



► Consistency and Availability

► Kafka makes the following guarantees about data consistency and availability:

1. Messages sent to a topic partition will be appended to the commit log in the order they are sent
2. A single consumer instance will see messages in the order they appear in the log
3. A message is 'committed' when all in sync replicas have applied it to their log, and
4. Any committed message will not be lost, as long as at least one in sync replica is alive

► Handling Writes

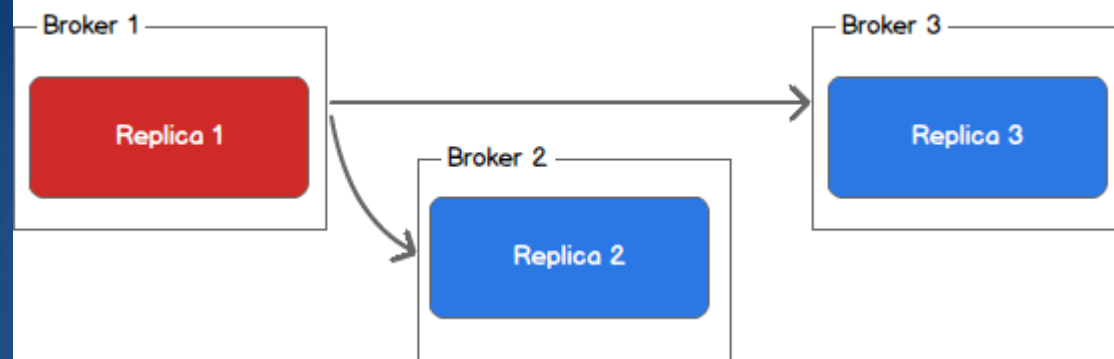
- When communicating with a Kafka cluster, all messages are sent to the partition's leader
- The leader is responsible for writing the message to its own in sync replica and, once that message has been committed, is responsible for propagating the message to additional replicas on different brokers
- Each replica acknowledges that they have received the message and can now be called in sync
- When every broker in the cluster is available, consumers and producers can happily read and write from the leading partition of a topic without issue. Unfortunately, either leaders or replicas may fail and we need to handle each of these situations

Kafka basics

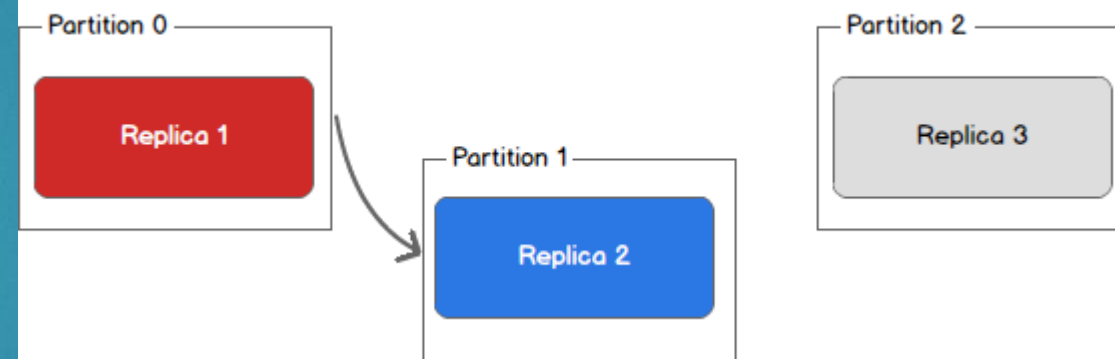
29

► Handling Failure

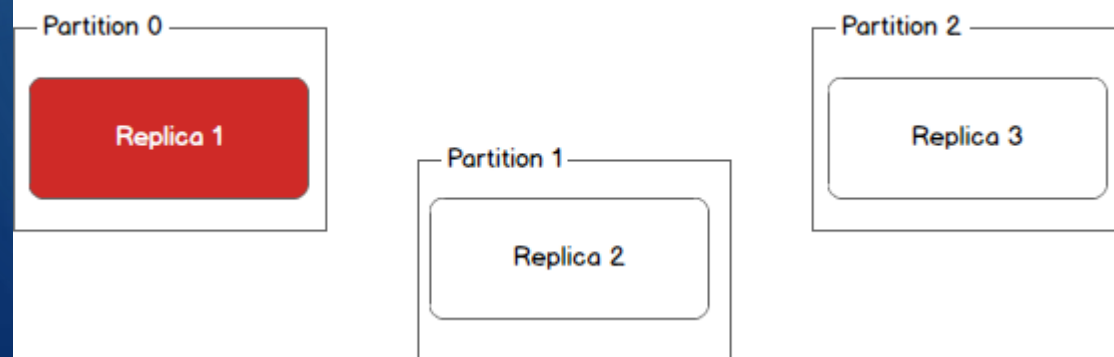
Leader (red) writes to replicas (blue)



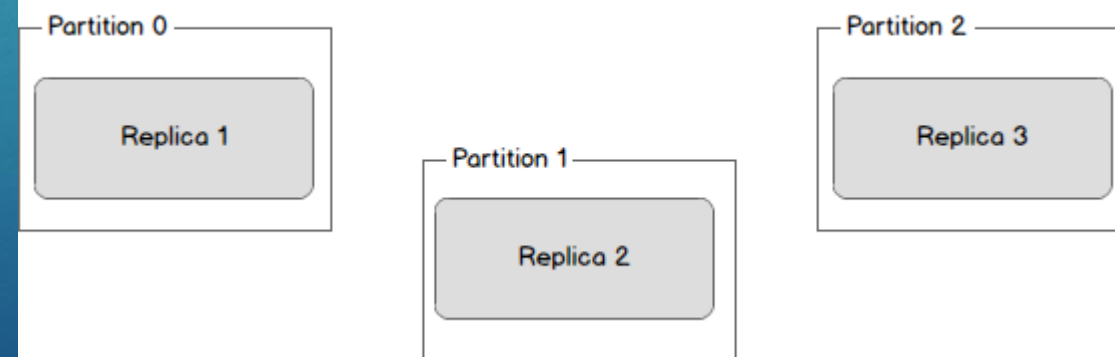
Leader (red) writes to live replicas (blue)



Leader (red) writes to live replicas (blue)



All replicas failed

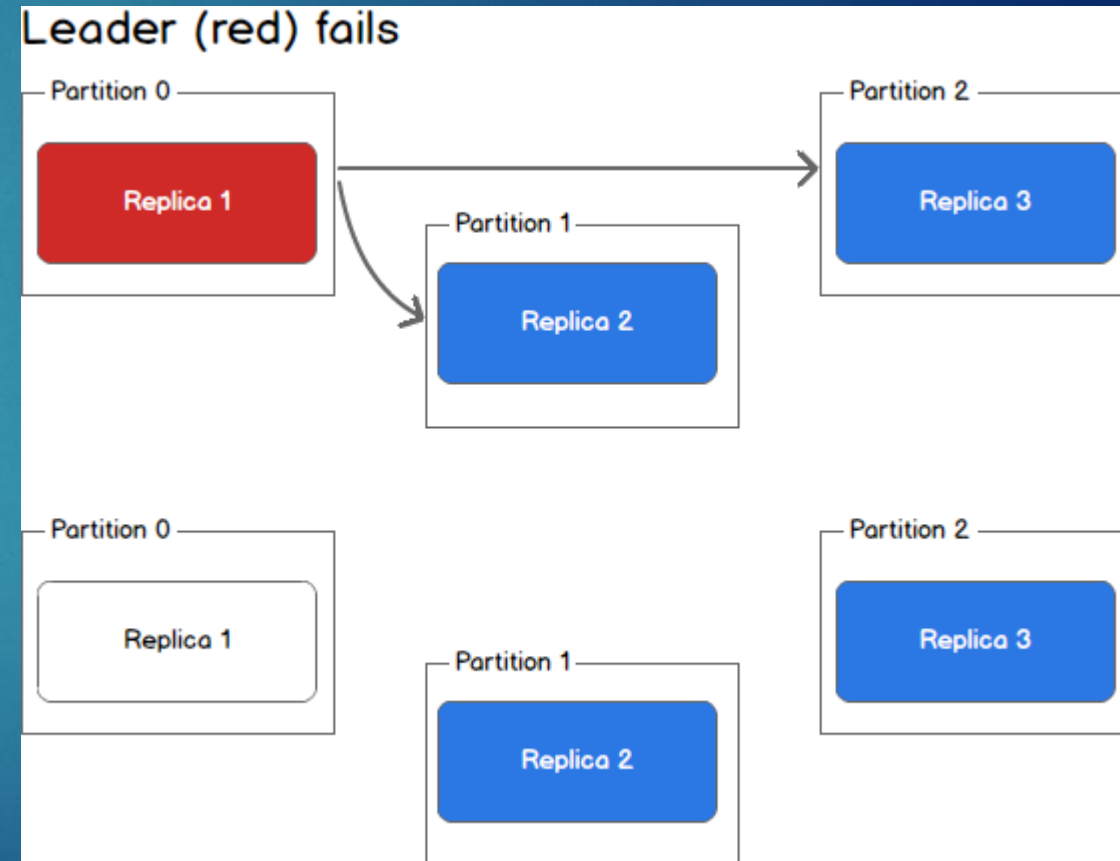


Kafka basics

30

► Handling Failure

- The first, and simplest, scenario is to wait until the leader is back up before continuing. Once the leader is back up it will begin receiving and writing messages and as the replicas are brought back online they will be made in sync with the leader
- The second scenario is to elect the second broker to come back up as the new leader. This broker will be out of sync with the existing leader and all data written between the time where this broker went down and when it was elected the new leader will be lost
 - As additional brokers come back up, they will see that they have committed messages that do not exist on the new leader and drop those messages
 - By electing a new leader as soon as possible messages may be dropped but we will minimize downtime as any new machine can be leader



- ▶ Consistency as a Kafka Client
 - ▶ For a producer we have three choices:
 1. Wait for all in sync replicas to acknowledge the message
 2. Wait for only the leader to acknowledge the message
 3. Do not wait for acknowledgement
 - ▶ On the consumer side, we can only ever read committed messages
 1. Receive each message at most once
 2. Receive each message at least once
 3. Receive each message exactly once

► Consistency as a Kafka Client

- For at most once message delivery, the consumer reads data from a partition, commits the offset that it has read, and then processes the message. If the consumer crashes between committing the offset and processing the message it will restart from the next offset without ever having processed the message. This would lead to potentially undesirable message loss
- A better alternative is at least once message delivery. For at least once delivery, the consumer reads data from a partition, processes the message, and then commits the offset of the message it has processed. In this case, the consumer could crash between processing the message and committing the offset and when the consumer restarts it will process the message again. This leads to duplicate messages in downstream systems but no data loss
- Exactly once delivery is guaranteed by having the consumer process a message and commit the output of the message along with the offset to a transactional system. If the consumer crashes it can re-read the last transaction committed and resume processing from there. This leads to no data loss and no data duplication. In practice however, exactly once delivery implies significantly decreasing the throughput of the system as each message and offset is committed as a transaction
- In practice most Kafka consumer applications choose at least once delivery because it offers the best trade-off between throughput and correctness

