JAVA FULL STACK DEVELOPMENT PROGRAM

Week 9 Day 2: Messaging

OUTLINE

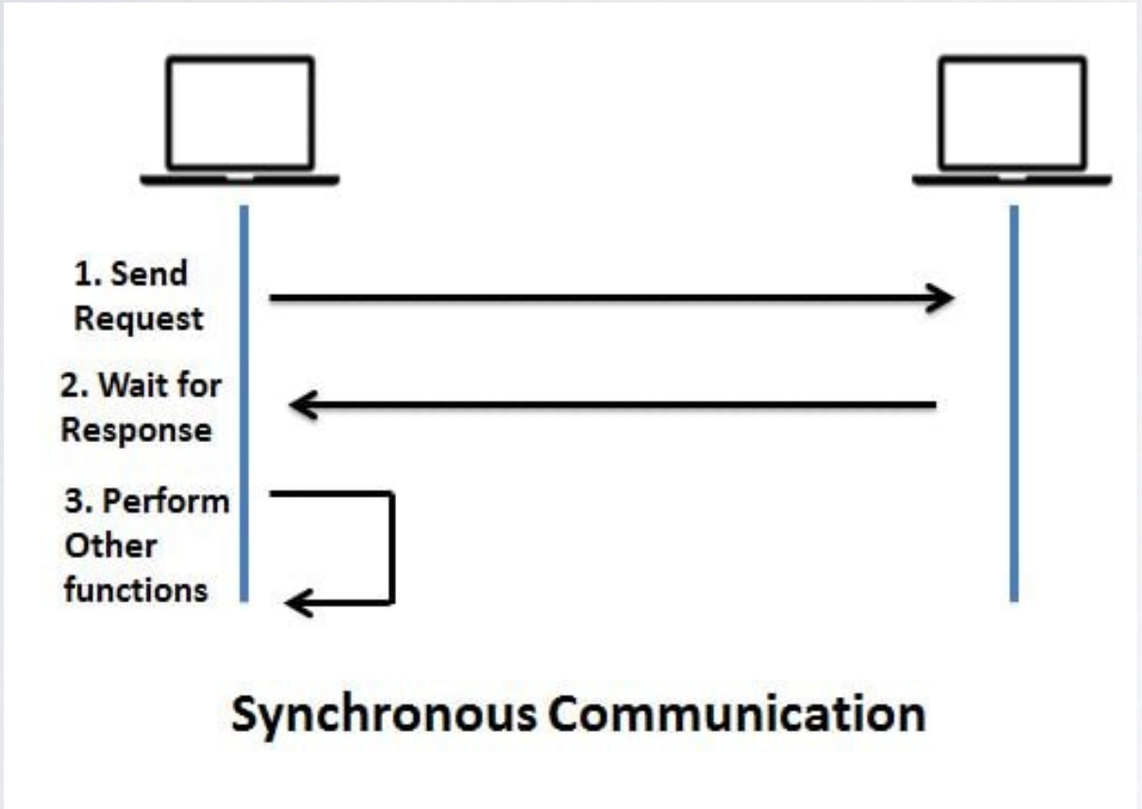
- Asynchronous
    - Messaging
        - Introduction
        - AMQP
        - RabbitMQ
- Scheduling

SCENARIO

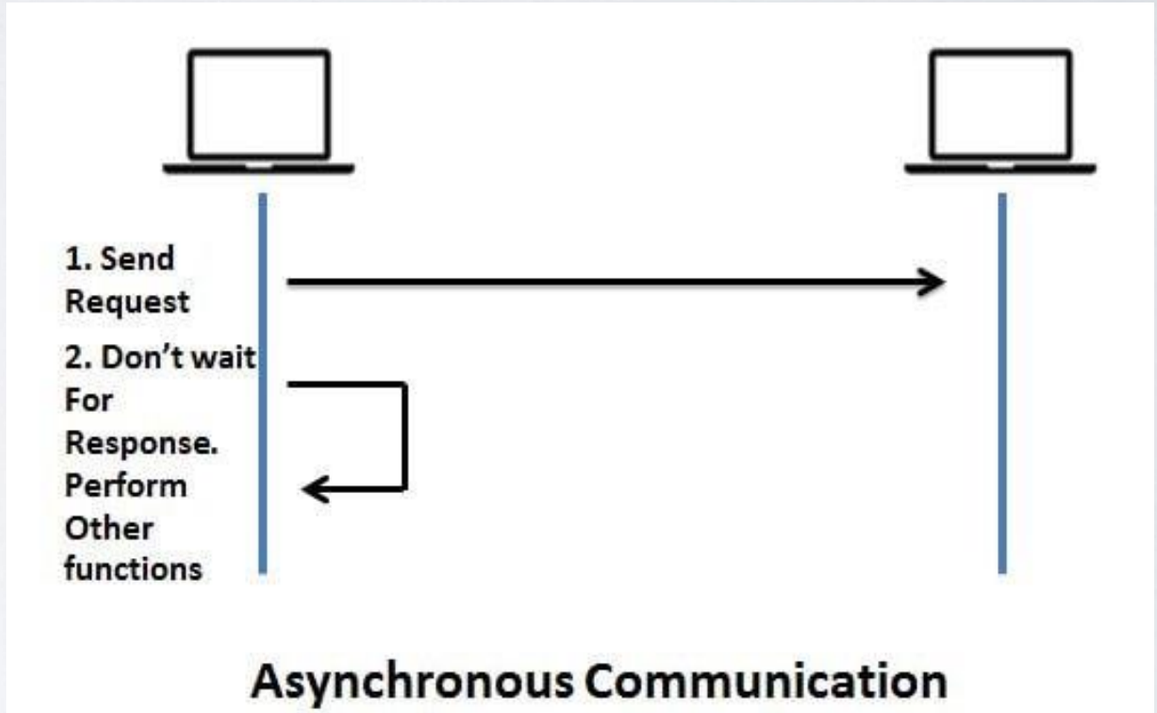
- In enterprise application, it is normal to process transaction offline
  - After user submit an order, instead of asking user to wait online until all process, such as validation and saving to database, completed, we would like to process the transaction offline and after all process have completed we send an email to inform customer.
- When updating user information, it may require us to change multiple database. For example, the user updated phone number, address, billing information and company information.

MESSAGING

- Remote procedure call (RPC) systems are synchronous -- the caller must block and wait until the called method completes execution, and thus offer no potential for developing loosely coupled enterprise applications without the use of multiple threads
  - In other words, RPC systems require the client and the server to be available at the same time
- Message-Oriented Middleware (MOM) systems provide solutions to such problems
  - They are based on the asynchronous interaction model, and provide the abstraction of a message queue that can be accessed across a network
  - The Java Message Service (JMS) (API) was designed to make it easy to develop business applications that asynchronously send and receive business data and events
  - It is an implementation to handle the producer–consumer problem

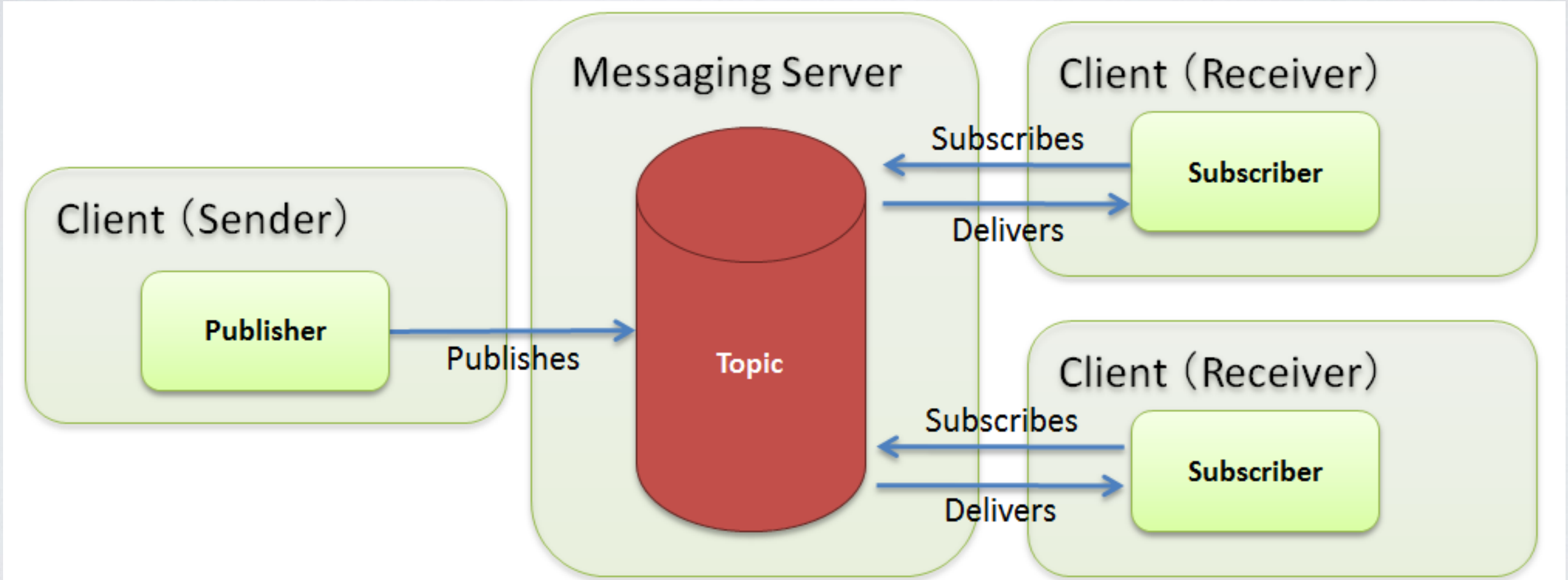


Synchronous Communication



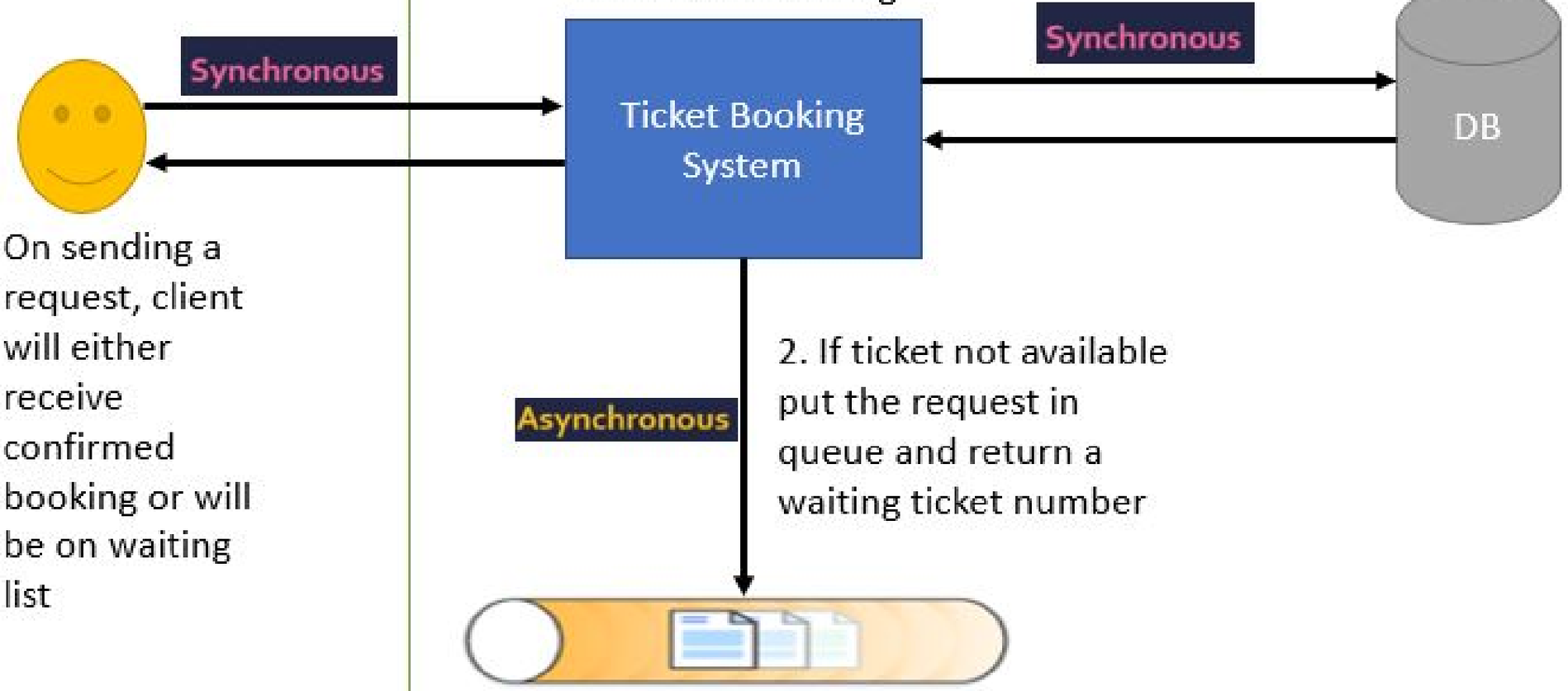
Asynchronous Communication

MESSAGING


Messaging Server
Client（Receiver）
Subscribes
Subscriber
Delivers
Client（Sender）
Publisher
Publishes
Topic
Client（Receiver）
Subscribes
Subscriber
Delivers

TICKET BOOKING SYSTEM

1. Check Ticket availability. If available return ticket number to client with confirmed booking



On sending a request, client will either receive confirmed booking or will be on waiting list

AMQP

- AMQP (Advanced Message Queuing Protocol) is a networking protocol that enables conforming client applications to communicate with conforming messaging middleware brokers.
  - Messaging brokers receive messages from producers (applications that publish them, also known as publishers) and route them to consumers (applications that process them)
  - Since it is a protocol, there is no limit with the programming language
- Note: JMS and AMQP is not comparable
  - JMS is a Java API
  - AMQP is a protocol which can be implemented by different Languages. (That's one of reason why many Java application switch from JMS to AMQP)

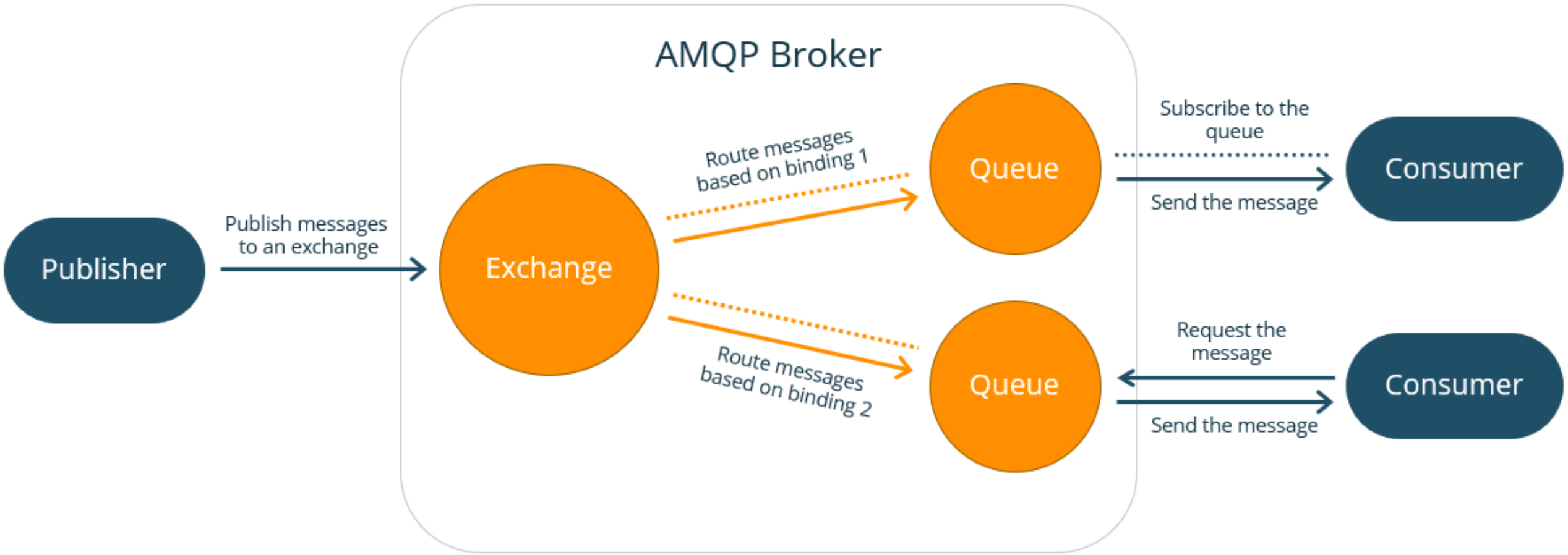
AMQP

- Message Broker
  - In general, a broker is a person who facilitates trades between a buyer and a seller.
    - An example could be a real estate agent or a stockbroker
- Similarly, if we want to trade messages between two distributed software components, we need a mediator.
  - This mediator is known as the message broker.
  - It receives incoming messages from a sender and sends them to a recipient. This way the sender and receiver can be totally isolated
- RabbitMQ is one of such message broker

AMQP

- The conceptual model of AMQP is quite simple and straightforward.
- It has three entities
  - Queue
  - Binding
  - Exchange
- When a publisher pushes a message to RabbitMQ
  - It first arrives at an exchange.
  - The exchange then distributes copies of these messages to variously connected queues.
  - Finally, consumers receive these messages

AMQP


AMQP Broker
Publisher
Publish messages to an exchange
Exchange
Route messages based on binding 1
Route messages based on binding 2
Queue
Queue
Subscribe to the queue
Send the message
Request the message
Send the message
Consumer
Consumer

AMQP

- Queues
  - These queues are somehow similar to the queues from our data structure classes
  - RabbitMQ queues also follow FIFO — First-In-First-Out methodology.
  - A queue is a place where RabbitMQ stores messages/data
- Bindings
  - Bindings are the rules that a queue defines while establishing a connection with an exchange
  - You can have a queue connected to multiple exchanges.
  - Every queue is also connected to a default exchange.
  - An exchange will use these bindings to route messages to queues.

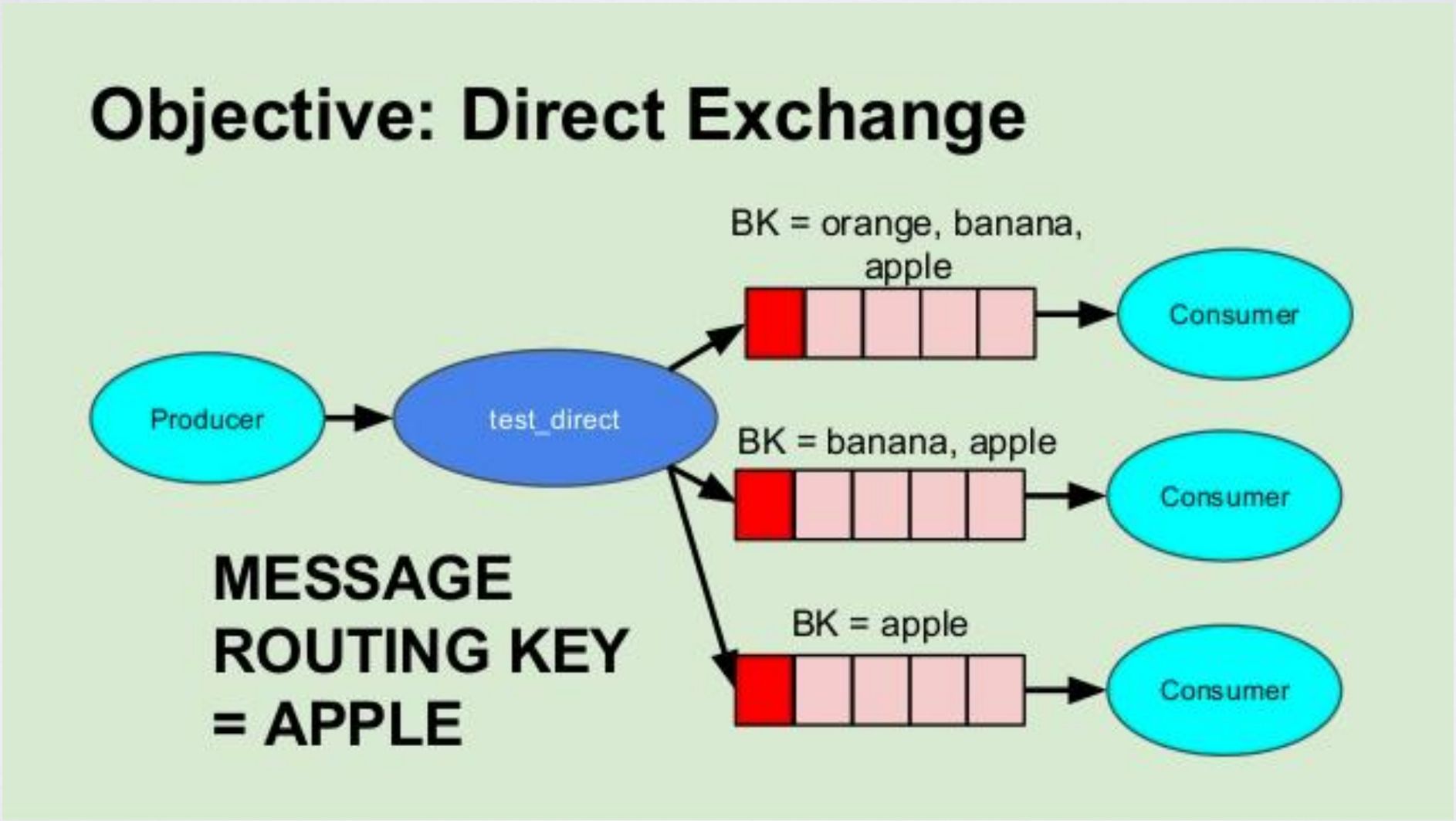
AMQP

- Exchanges
  - An Exchange is a gateway to RabbitMQ for your messages.
  - The distance the message has to travel inside RabbitMQ depends on the type of exchange.
  - The routing key is a message attribute the exchange looks at when deciding how to route the message to queues depending on exchange type listed below
  - Primarily there are four exchange types.
    - Direct
    - Fanout
    - Topic
    - Header

AMQP

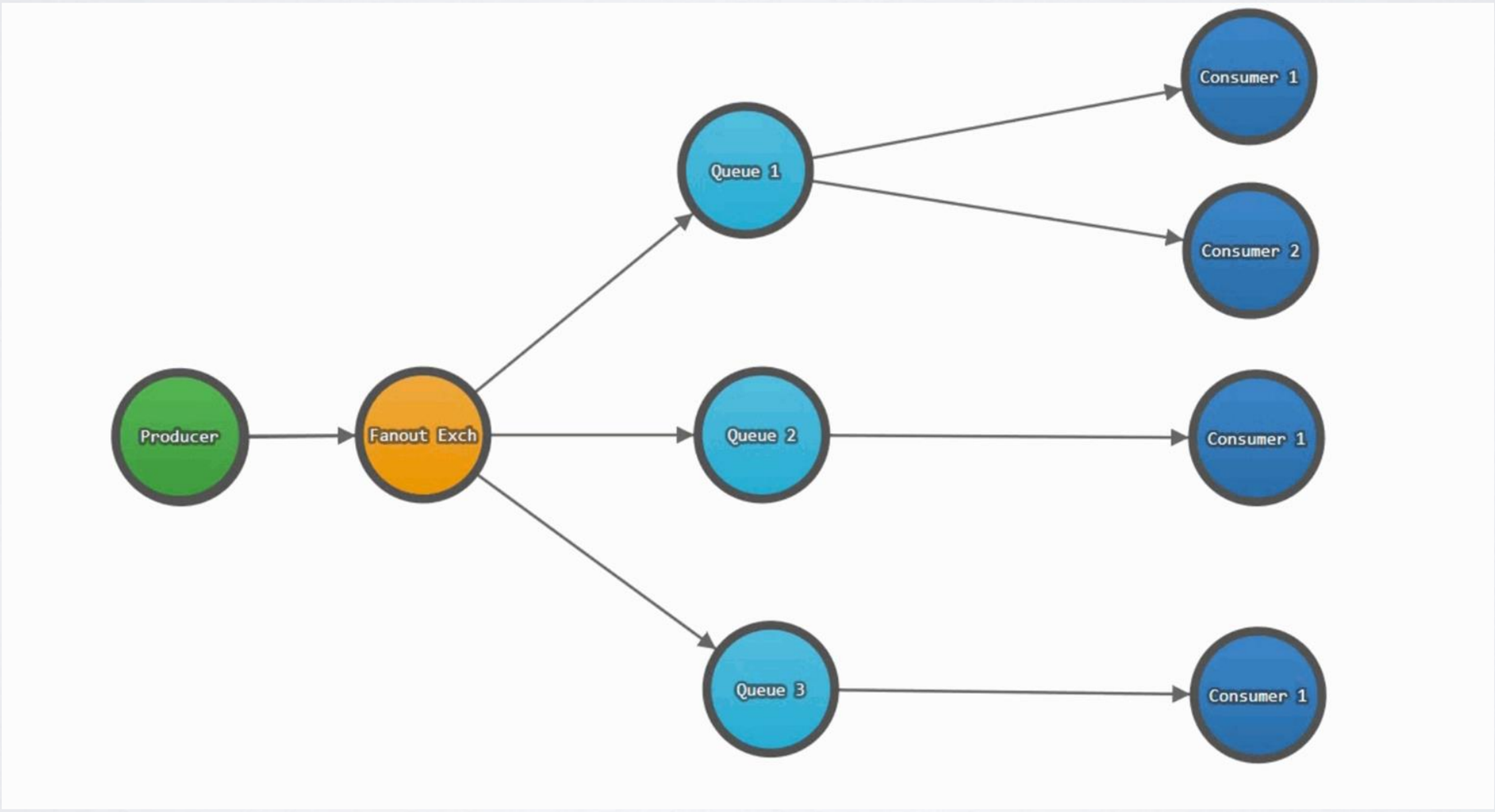
- Direct — A direct exchange delivers a message directly to the queues that satisfy the below condition:
  - A routing key is an attribute of the message.
  - A message goes to the queue(s) with the binding key that exactly matches the routing key of the message.
  - On the other hand, a binding key is something you specify while creating a binding between a queue and an exchange

AMQP


Objective: Direct Exchange
BK = orange, banana, apple
Consumer
Producer
test_direct
BK = banana, apple
Consumer
MESSAGE ROUTING KEY = APPLE
BK = apple
Consumer

AMQP

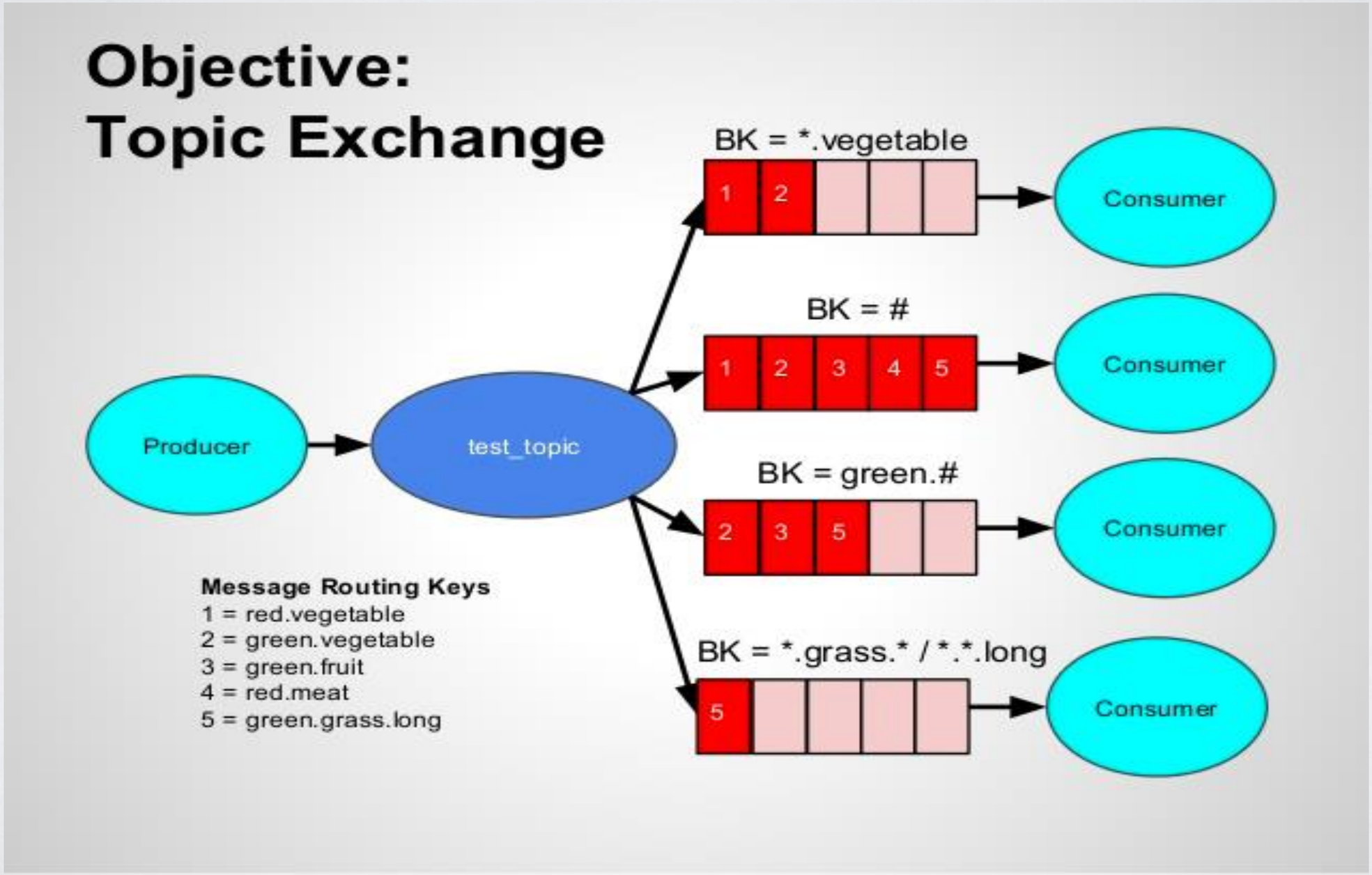
- Fanout — A fanout exchange copies and routes a received message to all queues that are bound to it regardless of routing keys or pattern matching as with direct and topic exchanges.



AMQP

- Topic — Messages are routed to one or many queues based on a matching between a message routing key and this pattern. A topic exchange routes a message by matching routing key with a pattern in the binding key like below:
- RabbitMQ uses two wild card characters for pattern matching * and #.
- Use a * to match 1 word and a # to match 0 or more words
- This type of exchange has a vast range of use cases.
  - It can be used in the publish-subscribe pattern, distributing relevant data to desiring workers processes and many more
  - In most case, we will have the topic exchange in enterprise application.

AMQP


Objective:
Topic Exchange
BK = *.vegetable
1
2
Consumer
BK = #
1
2
3
4
5
Consumer
Producer
test_topic
BK = green.#
2
3
5
Consumer
Message Routing Keys
1 = red.vegetable
2 = green.vegetable
3 = green.fruit
4 = red.meat
5 = green.grass.long
BK = *.grass.* / *.*.long
5
Consumer

AMQP

- Header — Headers exchanges are very similar to topic exchanges, but route messages based on header values instead of routing keys. A message matches if the value of the header equals the value specified upon binding.

- When creating bindings for a header exchange, it is possible to bind a queue to match more than one header.
    - In such a case, RabbitMQ should know from the producer if it should match all or any of these keys.

- It is rarely used exchange types.

RABBITMQ

- Installation:
  https://www.rabbitmq.com/download.html
- Follow the instruction online to install RabbitMQ
- Let's take a look at the RabbitMQ Console

SPRING RABBITMQ

- Spring provides us a very good integration with RabbitMQ — rabbitTemplate

- In order to use the rabbitTemplate, we have to configure the proper queue and exchange.

- Let's take a look at an example

SCENARIO

- Now we are able to process our transaction asynchronously by either using threading or messaging.

- However, what if we send a message to another system, but after one day, we still don’t receive any update from them?

SCHEDULER

- Scheduling is to execute tasks for specific time period.
- Spring Boot provides different scheduling functionalities in spring applications.
  - We can schedule your tasks via @Scheduled annotation or via using a custom thread pool.
- To get started, we just need to add @EnableScheduling to our configuration class.

@SCHEDULED ANNOTATION

- @Scheduled annotation is added to the function in which our task is implemented and it also has some parameters
  - fixedRate
  - fixedDelay
  - initialDelay
  - cron

@SCHEDULED ANNOTATION

- Fixed Rate
  - The method is invoked for every specified time.
- Note that the time measured *between successive start times of each invocation*.
- In other words, the task is invoked again even if the previous invocation of the task is not finished

```
@Scheduled(fixedRate = 1000)
public void scheduledTaskWithFixedRate() {
    Logger.debug("scheduledTaskWithFixedRate: current time: " + new Date());
}
```

@SCHEDULED ANNOTATION

- Fixed Delay
  - The method is invoked for every specified time like for fixedRate but the time is measured from *completion time* of each preceding invocation.

```
@Scheduled(fixedDelay = 3000)
public void scheduledTaskWithFixedDelayAndProcessingTime() throws InterruptedException {
    Logger.debug("scheduledTaskWithFixedDelayAndProcessingTime: started: " + new Date());
    TimeUnit.SECONDS.sleep(2);
    Logger.debug("scheduledTaskWithFixedDelayAndProcessingTime: finished: " + new Date());
}
```

@SCHEDULED ANNOTATION

- Initial Delay
  - We can specify the milliseconds to wait before first execution of task.
- It could be used with both fixedRate and fixedDelay

```
@Scheduled(initialDelay = 5000, fixedRate = 1000)
public void scheduledTaskWithInitialDelay() {
    logger.debug("scheduledTaskWithInitialDelay: current time: " + new Date());
}
```

@SCHEDULED ANNOTATION

```
0 0 12 * * ? 2017
```

```
<second> <minute> <hour> <day-of-month> <month> <day-of-week> <year>
```

- cron expressions could also be used with Scheduled annotations
- it consists following fields

- From these, *<year>* field is optional

  - At 12:00 pm (noon) every day during the year 2017

@SCHEDULED ANNOTATION

- Cron Specials Characters In Expression
  - * (all) – it is used to specify that event should happen for every time unit. For example, "*" in the <minute> field – means "for every minute"
  - ? (any) – it is utilized in the <day-of-month> and <day-of -week> fields to denote the arbitrary value – neglect the field value. For example, if we want to fire a script at "5th of every month" irrespective of what the day of the week falls on that date, then we specify a "?" in the <day-of-week> field
- – (range) – it is used to determine the value range. For example, "10-11" in <hour> field means "10th and 11th hours"
- , (values) – it is used to specify multiple values. For example, "MON, WED, FRI" in <day-of-week> field means on the days "Monday, Wednesday, and Friday"
- / (increments) – it is used to specify the incremental values. For example, a "5/15" in the <minute> field, means at "5, 20, 35 and 50 minutes of an hour"
- # – it is used to specify the "N-th" occurrence of a weekday of the month, for example, "3rd Friday of the month" can be indicated as "6#3"

SCHEDULER

- Let’s take a look at an example