**How to Make Microservices Communicate**

**Microservices are loosely coupled.**

Microservices are pretty awesome, but if they can't communicate they won't do you much good.

**How can we make microservices talk?**

Different modes of communication.

Synchronous (blocking) and

Asynchronous (non-blocking) are used quite often

### MessageQueue vs Message Broker

**MQ** is a solution for application-to-application communication services regardless of where your applications or data reside. Whether on a single server, separate servers of the same type, or separate servers of different architecture types, MQ facilitates communications between applications by sending and receiving message data via messaging queues. Applications then use the information in these messages to interact with Web browsers, business logic, and databases. MQ provides a secure and reliable transport layer for moving data unchanged in the form of messages between applications but it is not aware of the content of the messages. MQ uses a set of small and standard application programming interfaces (APIs) that support a number of programming languages, including Visual Basic, NATURAL, COBOL, Java, and C across all platforms.

**Message Broker** is built to extend MQ, and it is capable of understanding the content of each message that it moves through the Broker. Customers can define the set of operations on each message depending on its content. The message processing nodes supplied with Message Broker are capable of processing messages from various sources, such as Java Message Service (JMS) providers, HyperText Transfer Protocol (HTTP) calls, or data read from files. By connecting these nodes with each other, customers can define linked operations on a message as it flows from one application to its destination.

***Message Broker can do the following:***

    \* Matches and routes communications between services

    \* Converts between different transport protocols

    \* Transforms message formats between requestor and service

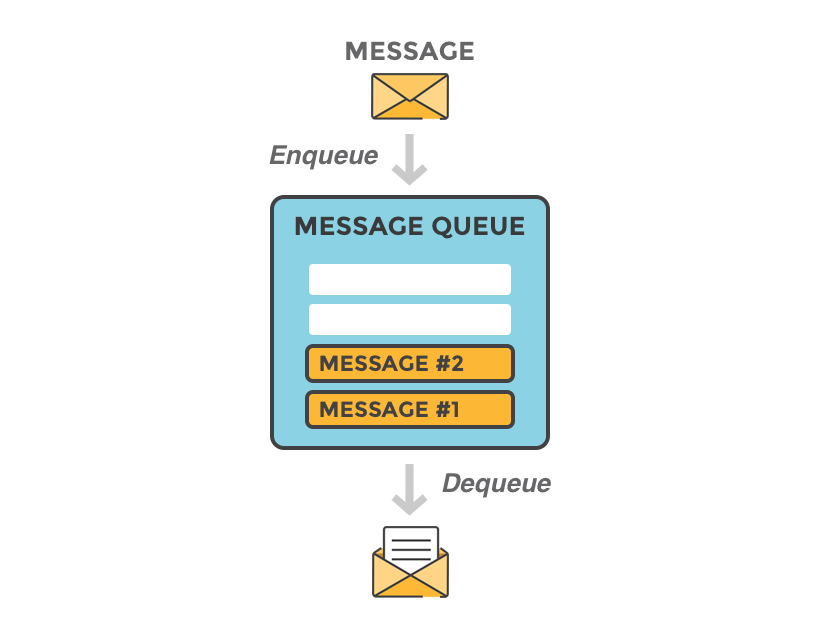
    \* Identifies and distributes business events from disparate sources

Messaging is **asynchronous**, decoupling applications by separating sending and receiving data. ... Or you want to use publish / subscribe, **asynchronous** processing, or work queues. All these are patterns, and they form part of messaging. **RabbitMQ** is a messaging broker - an intermediary for messaging.

A message queue is a data structure, or a container - a way to hold messages for eventual consumption. A message broker is a separate component that manages queues.

A message broker (also know a service bus) is a piece of middleware responsible with persisting and routing of message while allowing you to decouple your system into smaller parts. A message queue is simple a part of a message broker and just the name suggests is based on a queue like data structure (FIFO).

RabbitMQ is a message-queueing software also known as a *message broker* or *queue manager.* Simply said; it is software where queues are defined, to which applications connect in order to transfer a message or messages.



A message can include any kind of information. It could, for example, have information about a process or task that should start on another application (which could even be on another server), or it could be just a simple text message. The queue-manager software stores the messages until a receiving application connects and takes a message off the queue. The receiving application then processes the message.

A message broker acts as a middleman for various services (e.g. a web application, as in this example). They can be used to reduce loads and delivery times of web application servers by delegating tasks that would normally take up a lot of time or resources to a third party that has no other job.

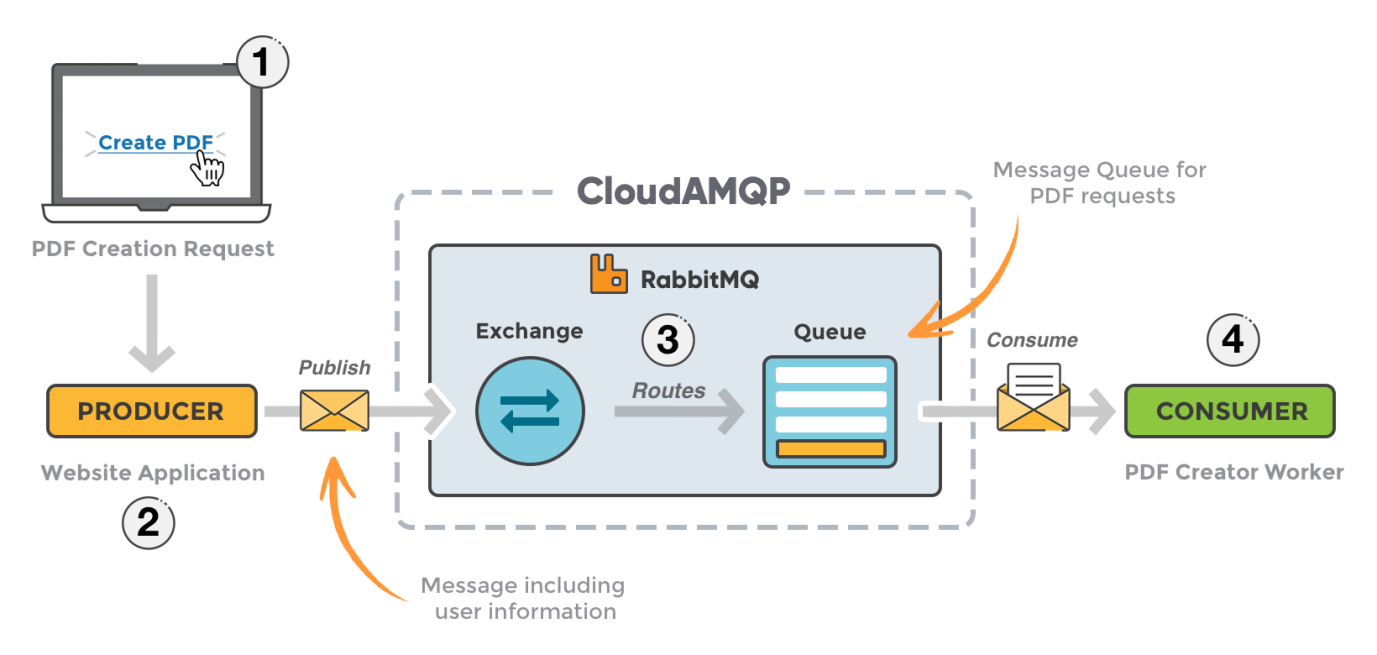


The basic architecture of a message queue is simple - there are client applications called producers that create messages and deliver them to the broker (the message queue). Other applications, called consumers, connect to the queue and subscribe to the messages to be processed. Software may act as a producer, or consumer, or both a consumer and a producer of messages. Messages placed onto the queue are stored until the consumer retrieves them.

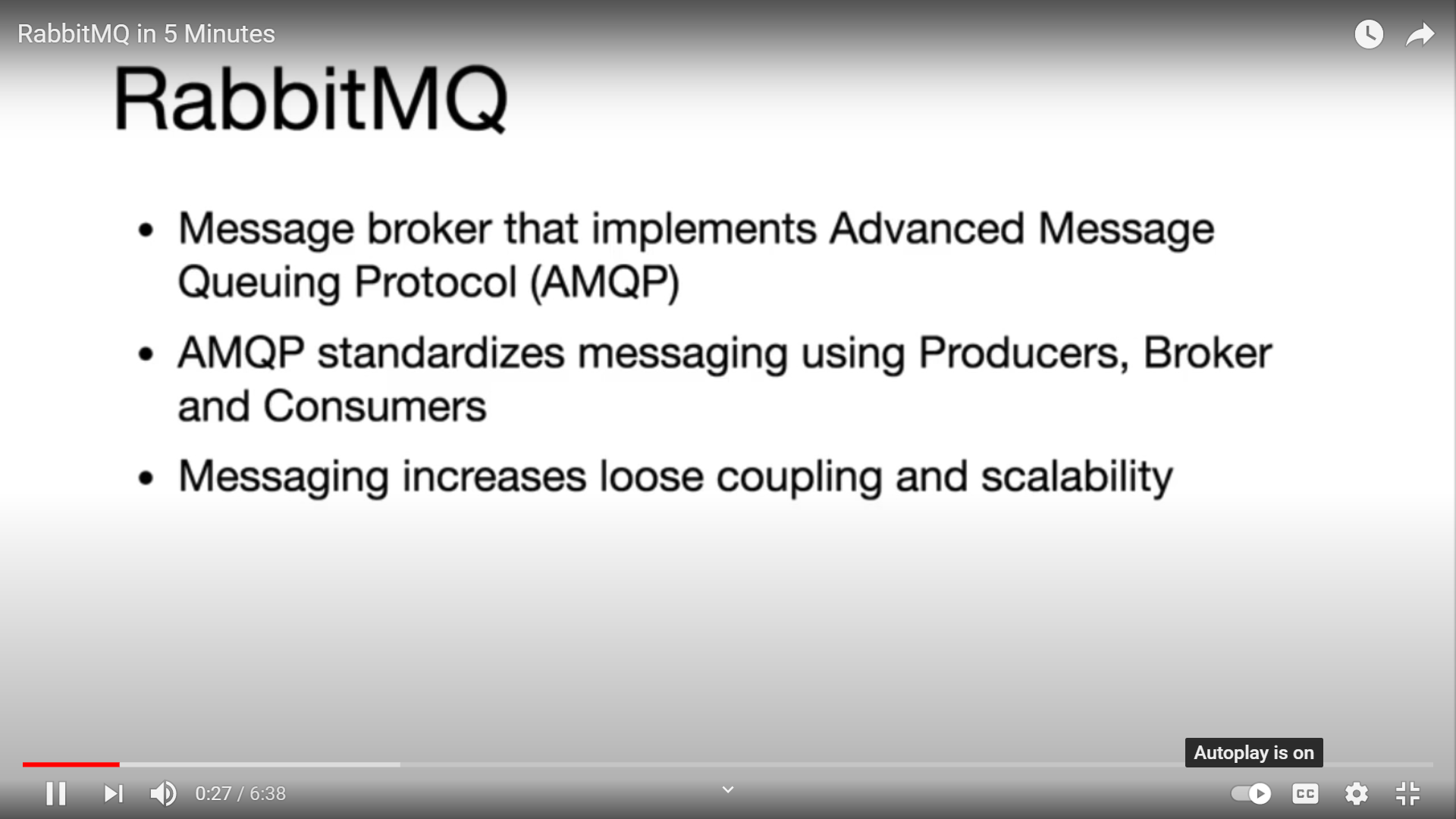
##### When and why should you use RabbitMQ?

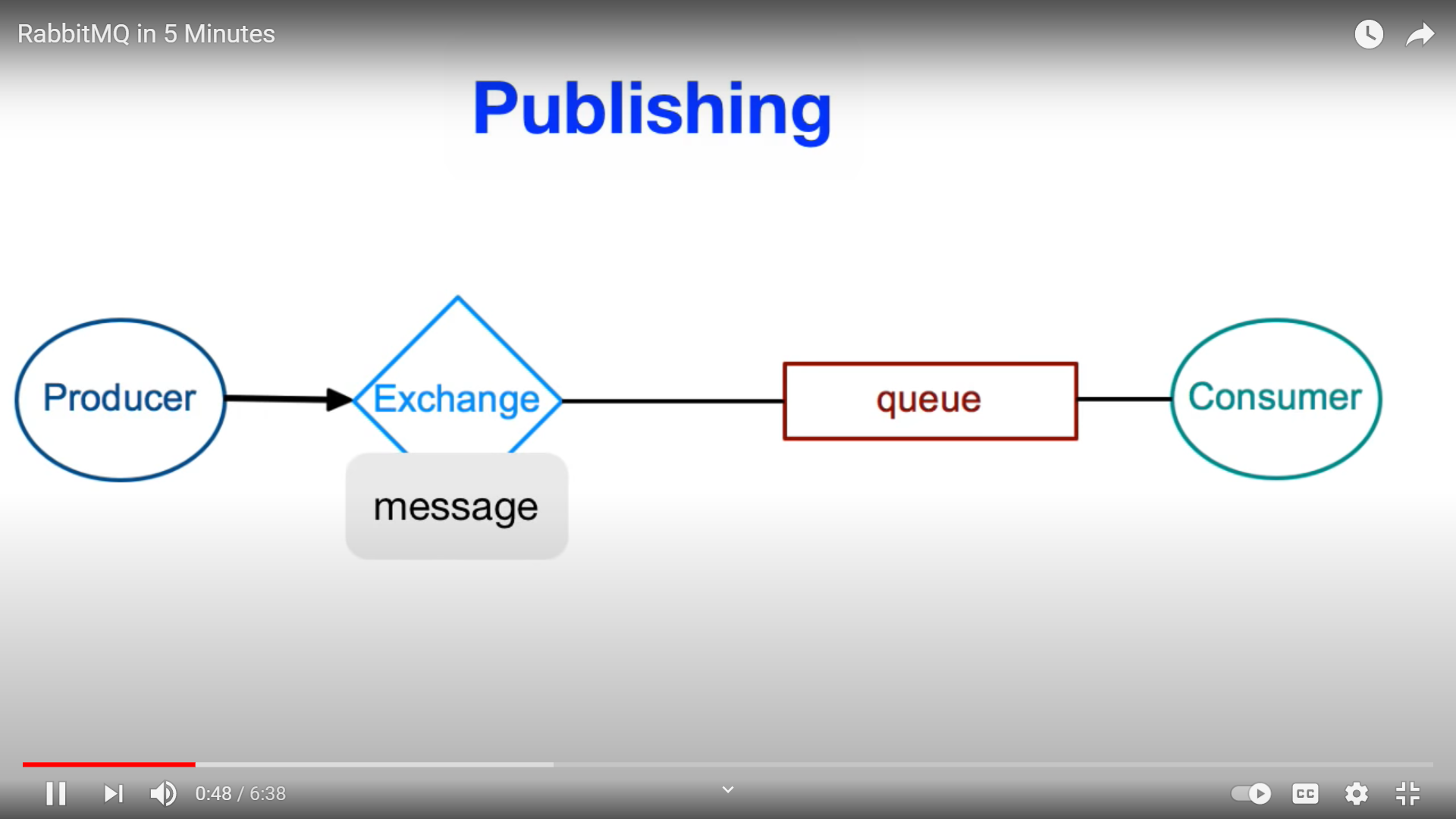
Message queueing allows web servers to respond to requests quickly instead of being forced to perform resource-heavy procedures on the spot that may delay response time. Message queueing is also good when you want to distribute a message to multiple consumers or to balance loads between workers.

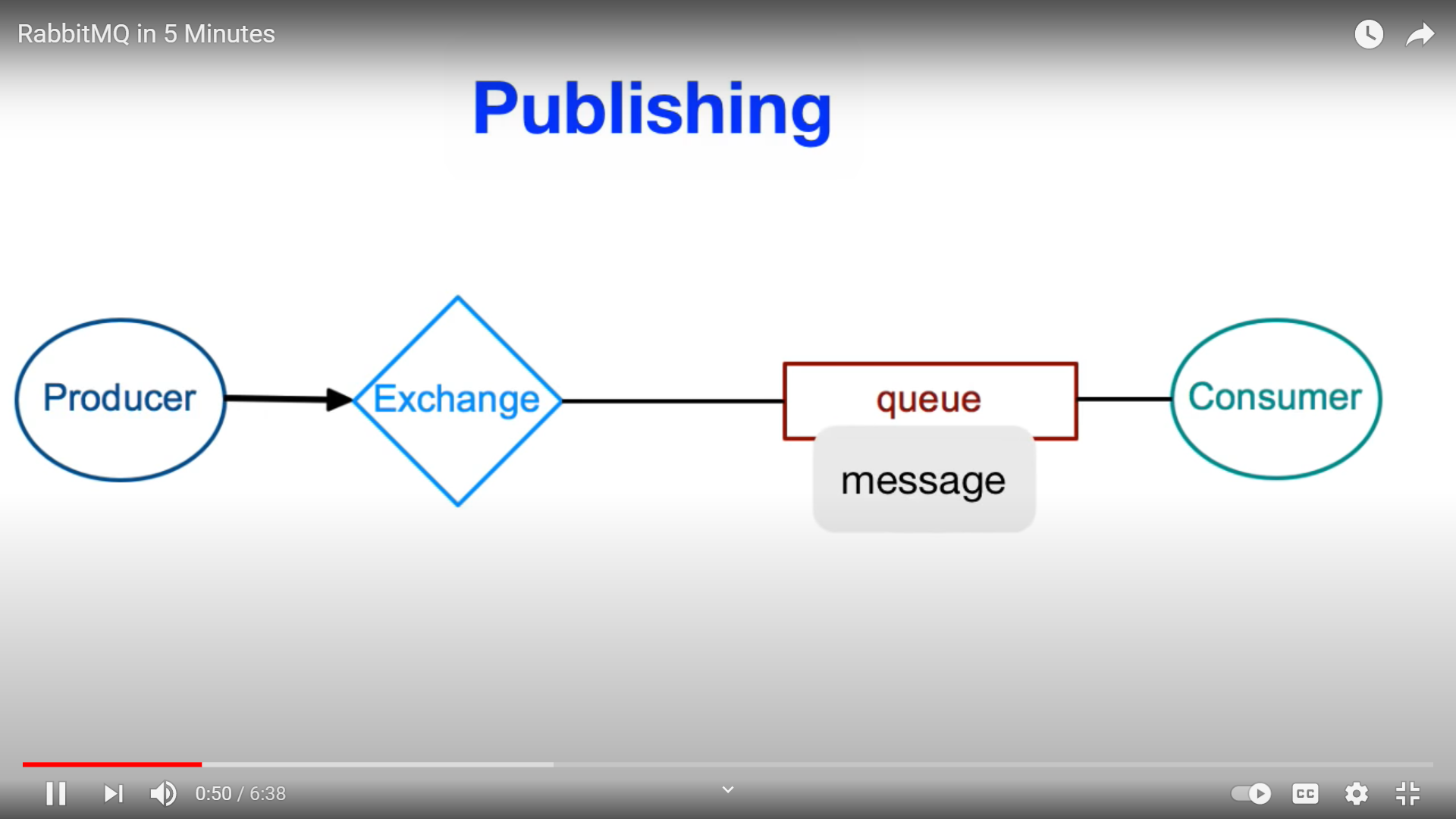
The consumer takes a message off the queue and starts processing the PDF. At the same time, the producer is queueing up new messages. The consumer can be on a totally different server than the producer or they can be located on the same server. The request can be created in one programming language and handled in another programming language. The point is, the two applications will only communicate through the messages they are sending to each other, which means the sender and receiver have low coupling.

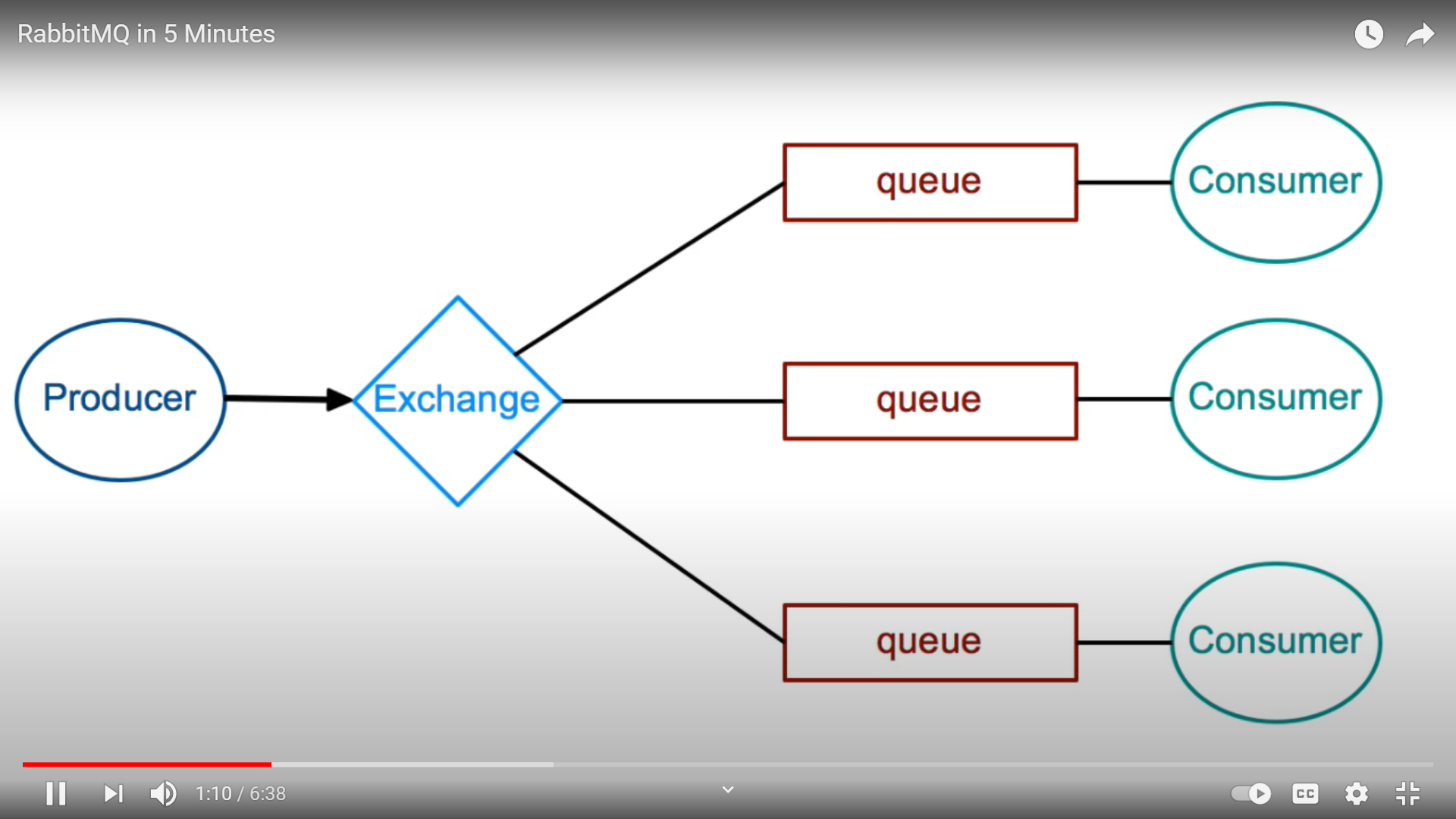


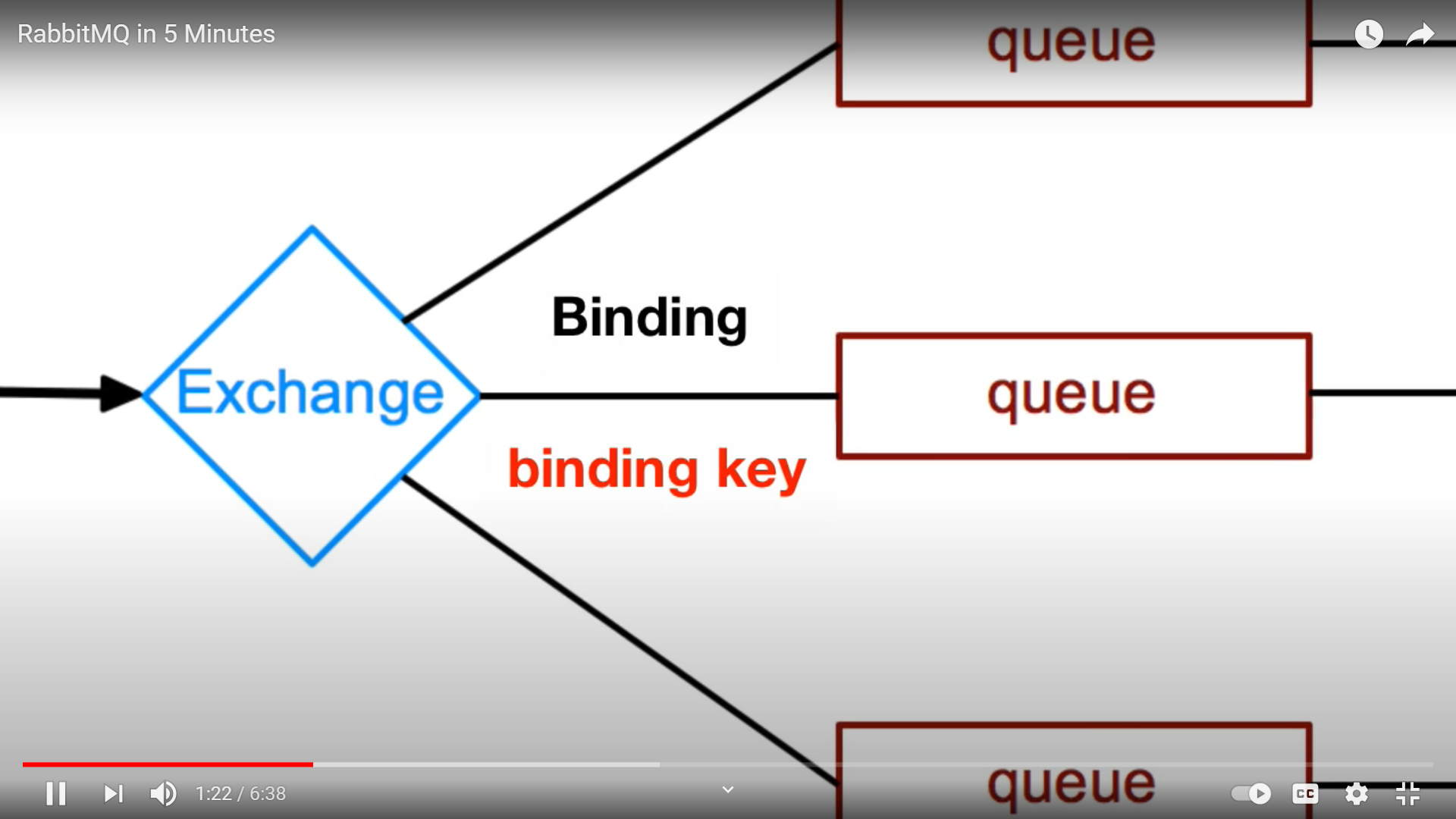
1. The user sends a PDF creation request to the web application.
2. The web application (the producer) sends a message to RabbitMQ that includes data from the request such as name and email.
3. An exchange accepts the messages from the producer and routes them to correct message queues for PDF creation.
4. The PDF processing worker (the consumer) receives the task message and starts processing the PDF.

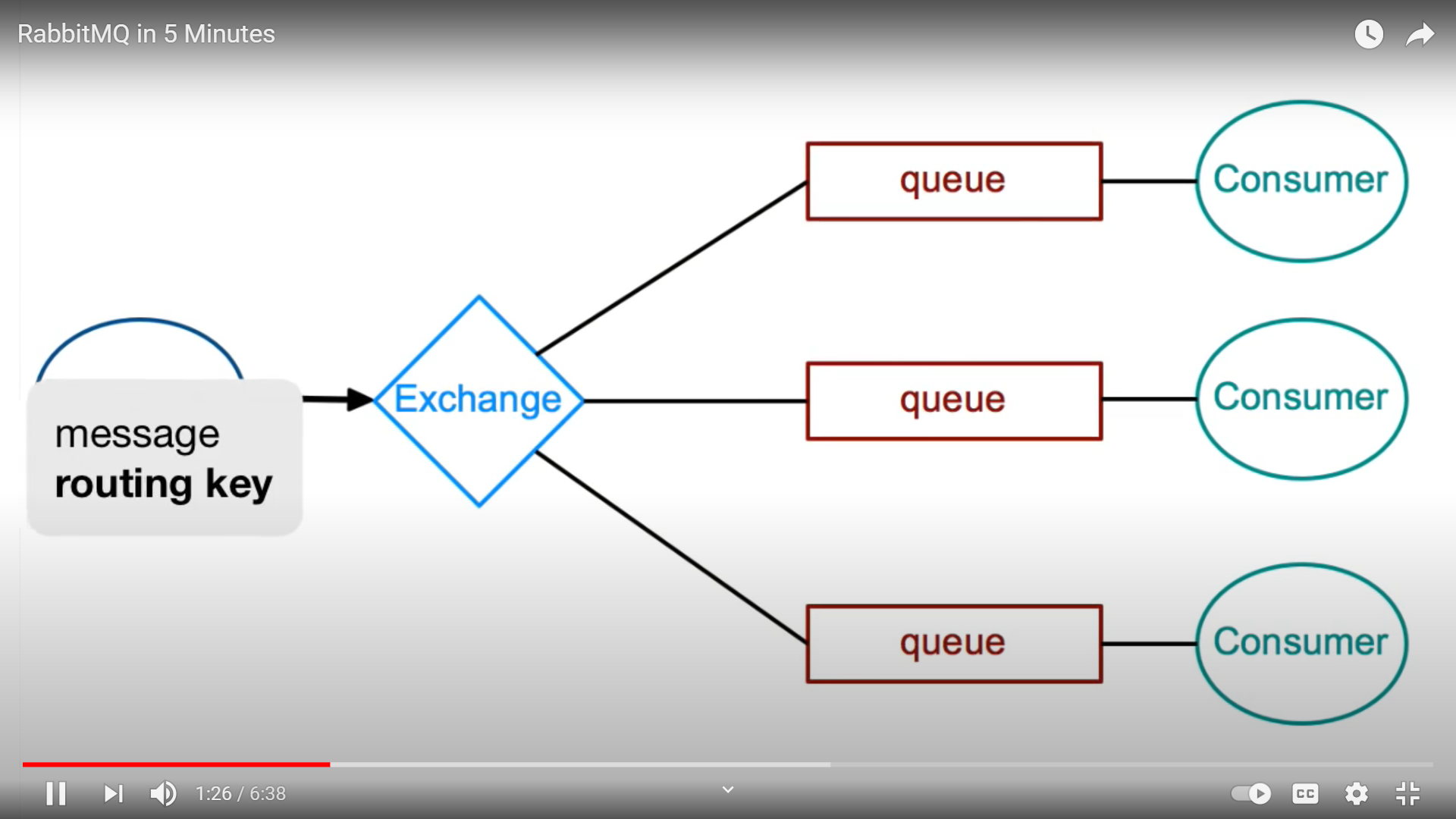


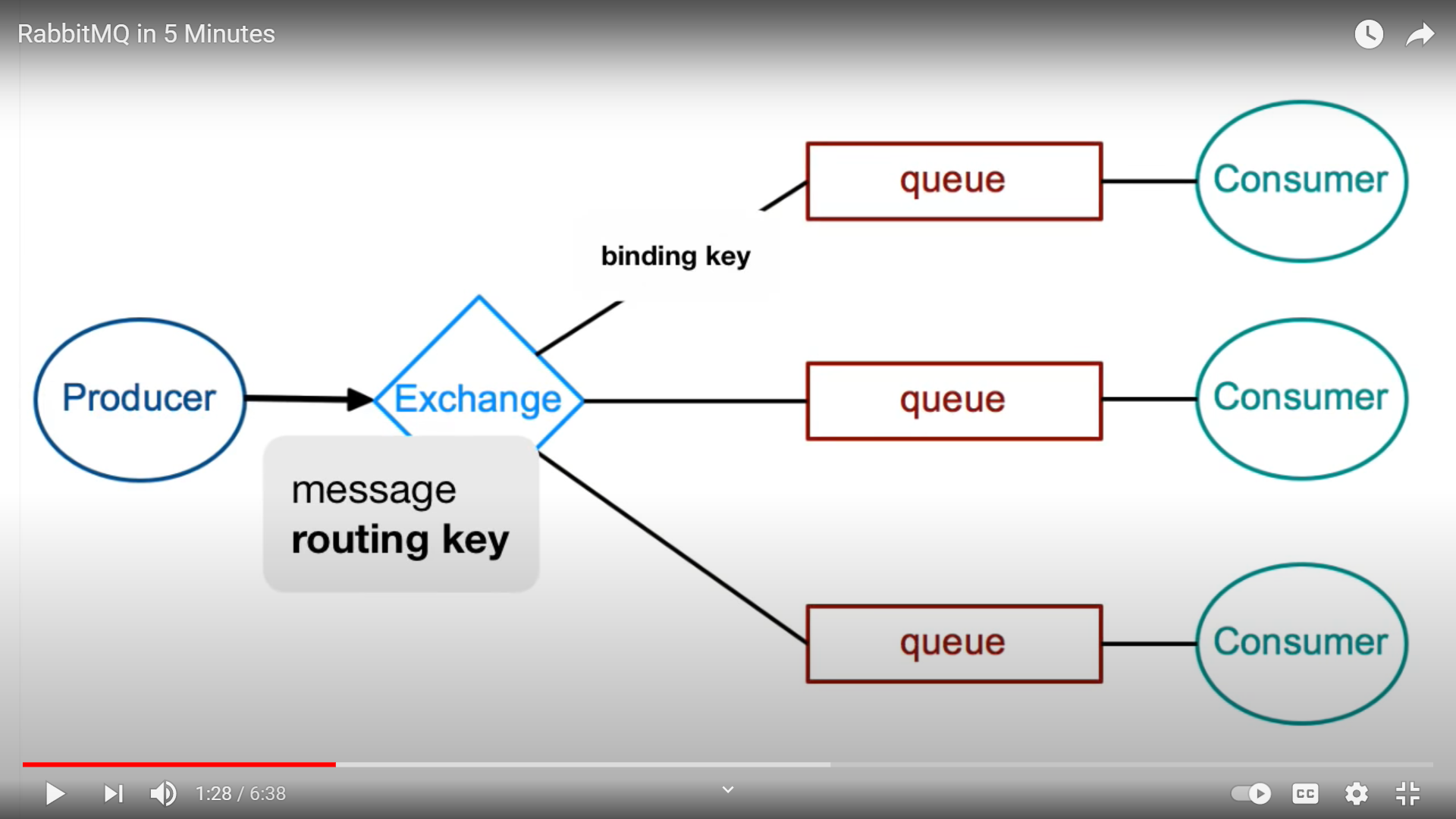


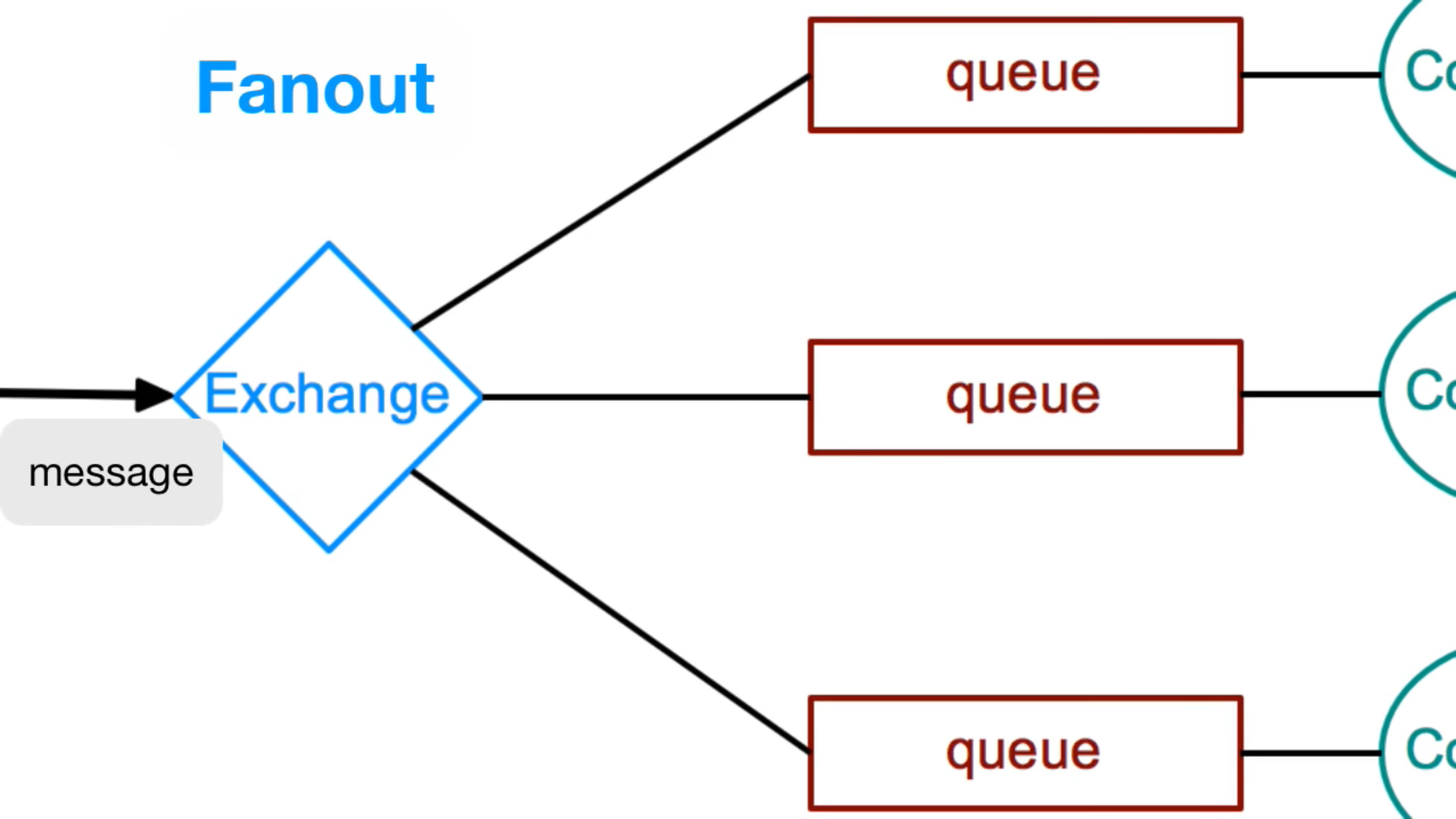


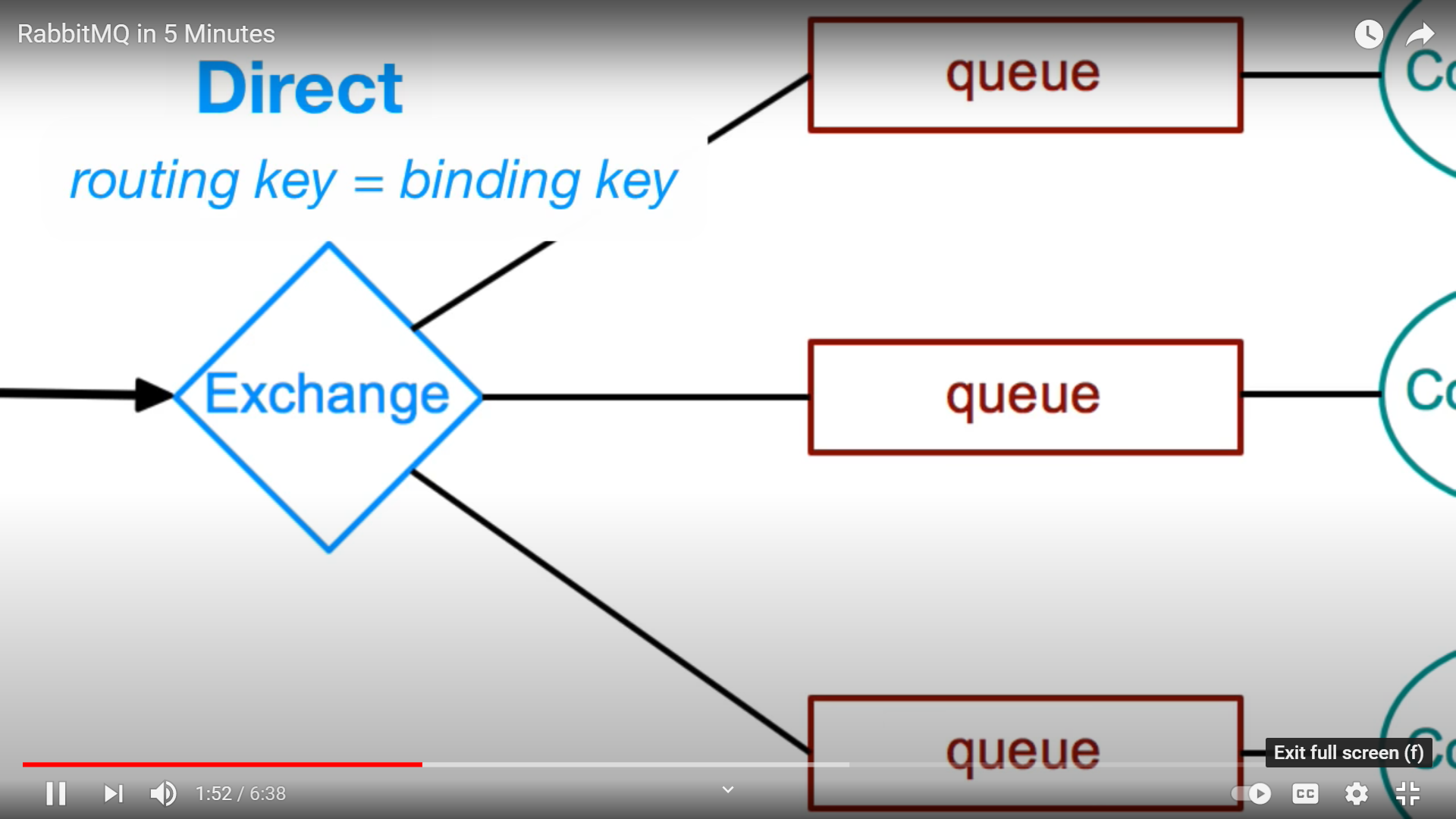


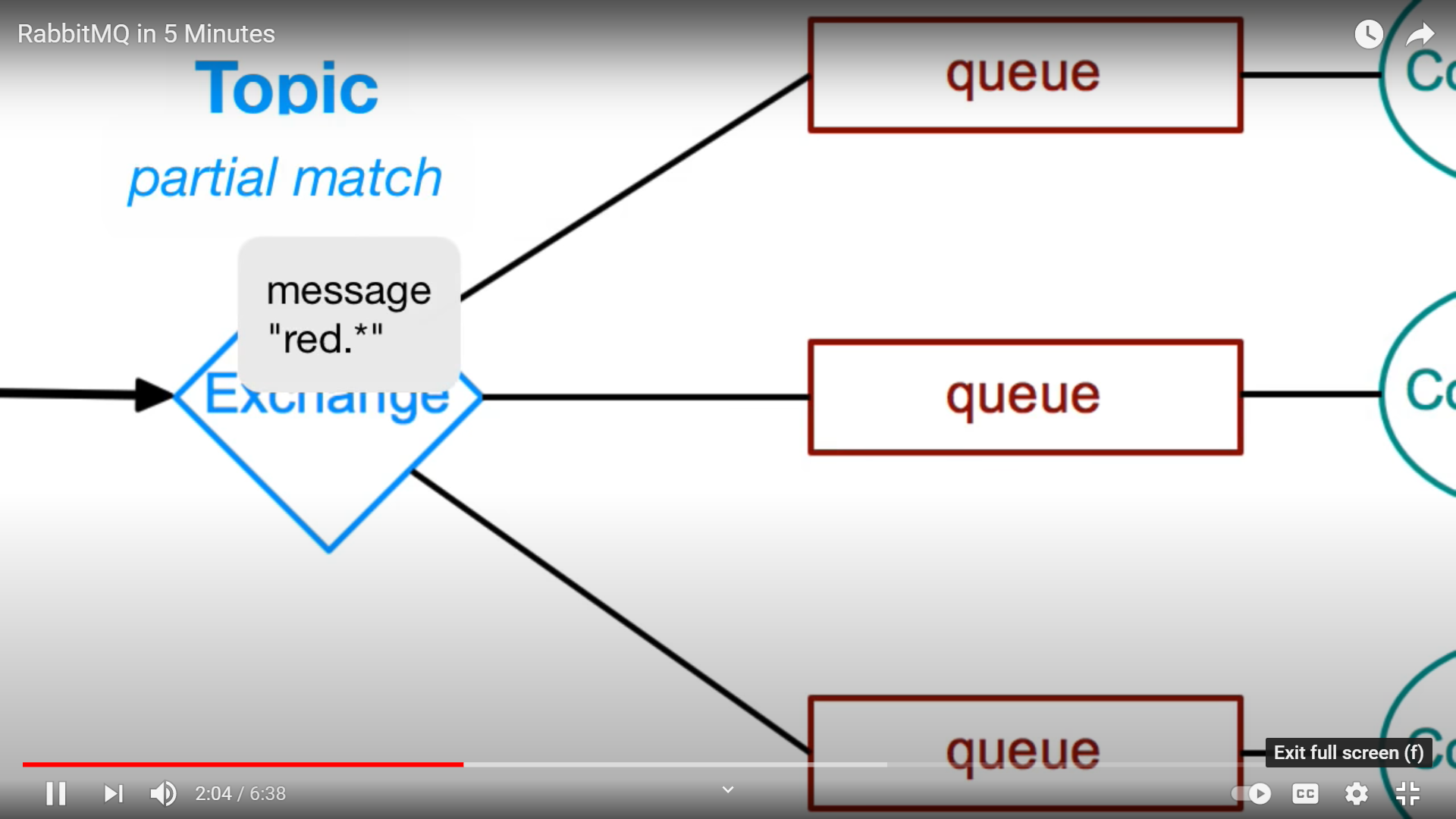


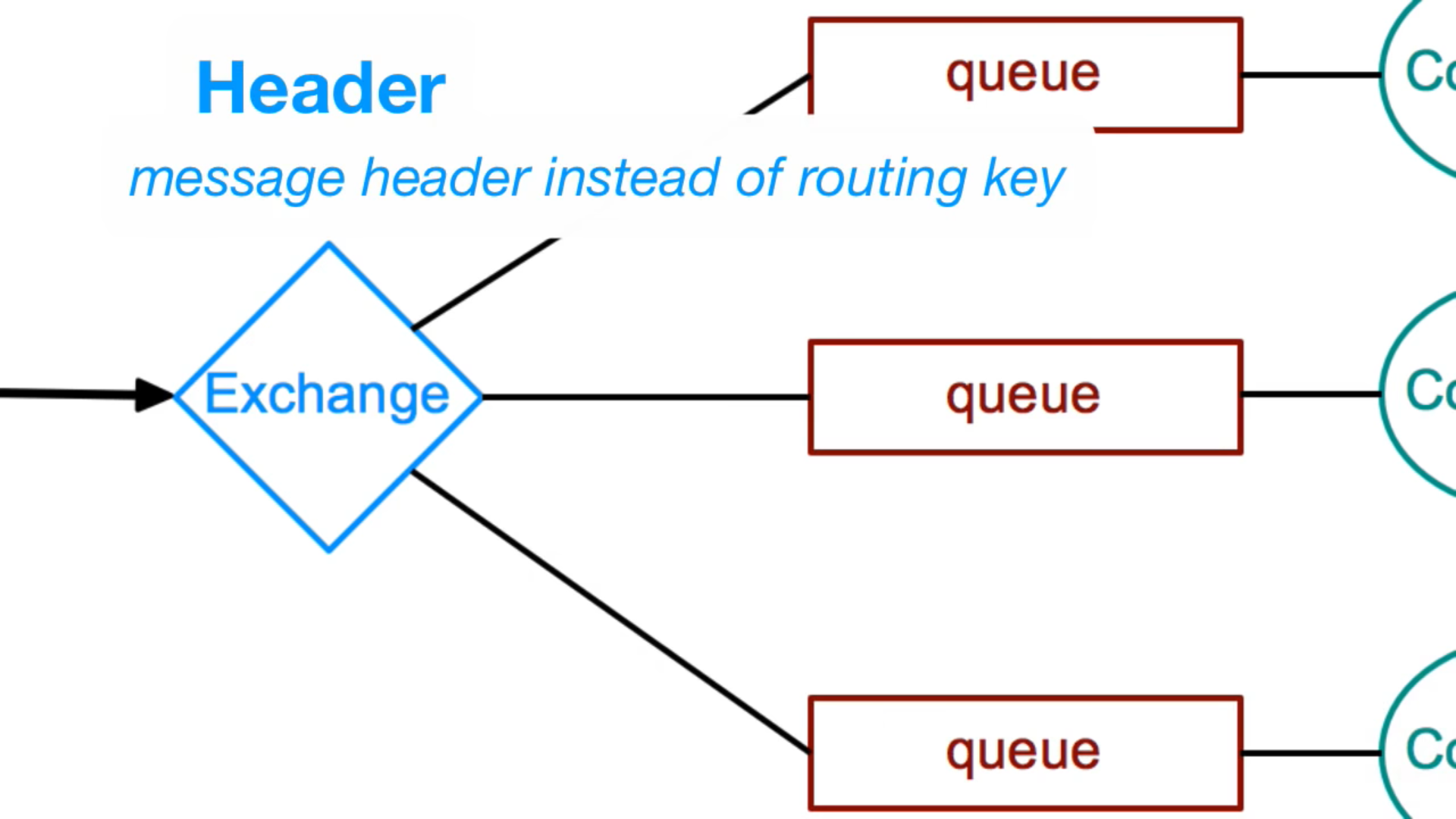


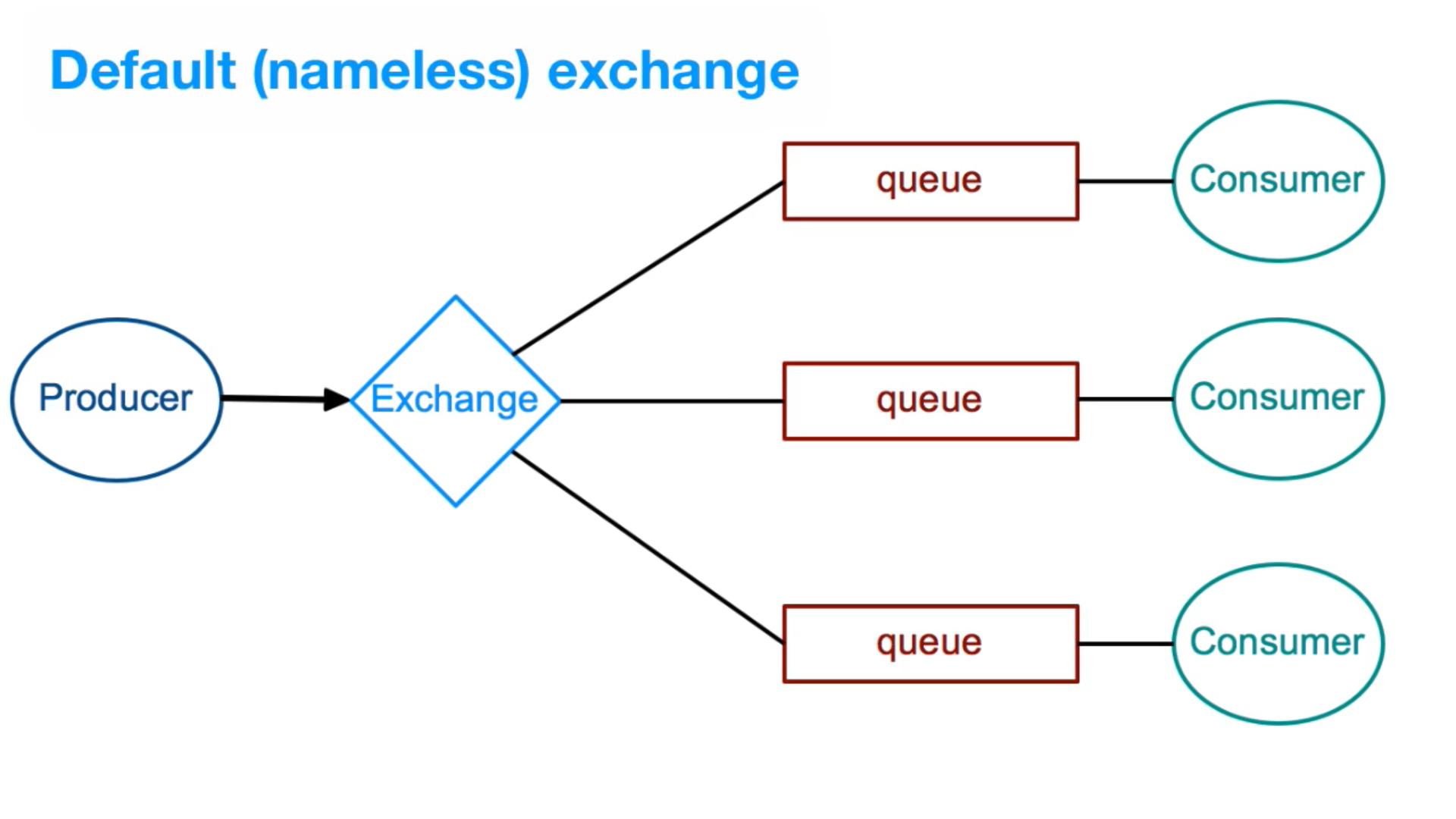


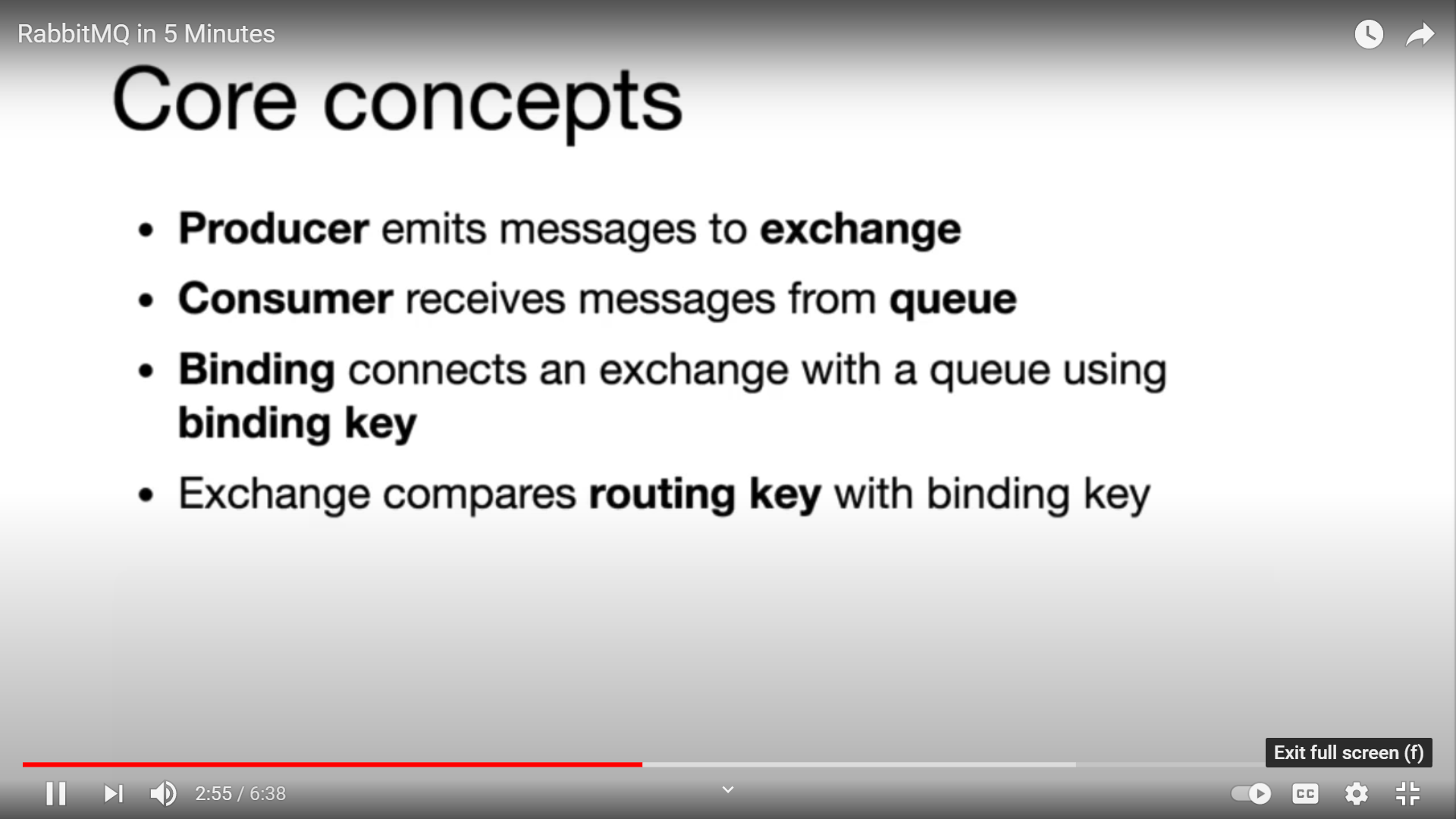












<https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html#when-and-why-should-you-use-rabbitmq>