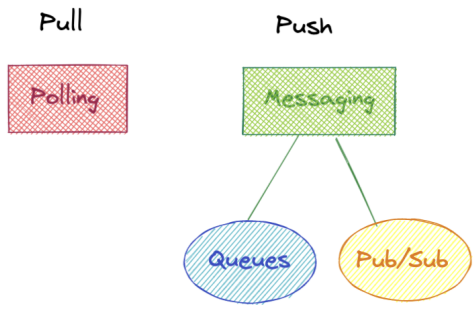
RabbitMQ + Docker-Compose 🥕

🔧 **CONTENTS**

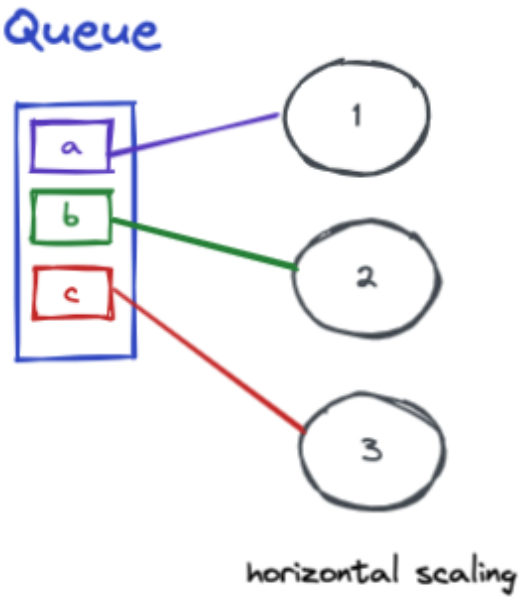
What the what?

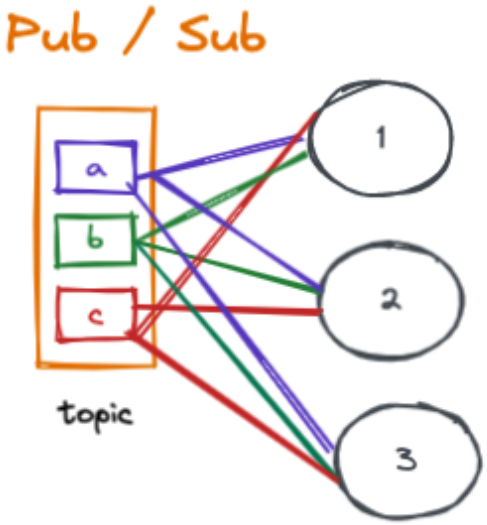
Taxonomy of the Models We Discuss

Polling (pull)	**Messaging (push with middleware:** *RabbitMQ*)



Polling (pull)	**Messaging (push with middleware:** *RabbitMQ*)
Each consumer is pulling from the producer. *Lots of load on the producer.*	**QUEUE** (One message queue, many consumers split up the messages)
No middleware needed.	**PUB/SUB** (Many copies of each message queue, every consumer gets a copy of every message)

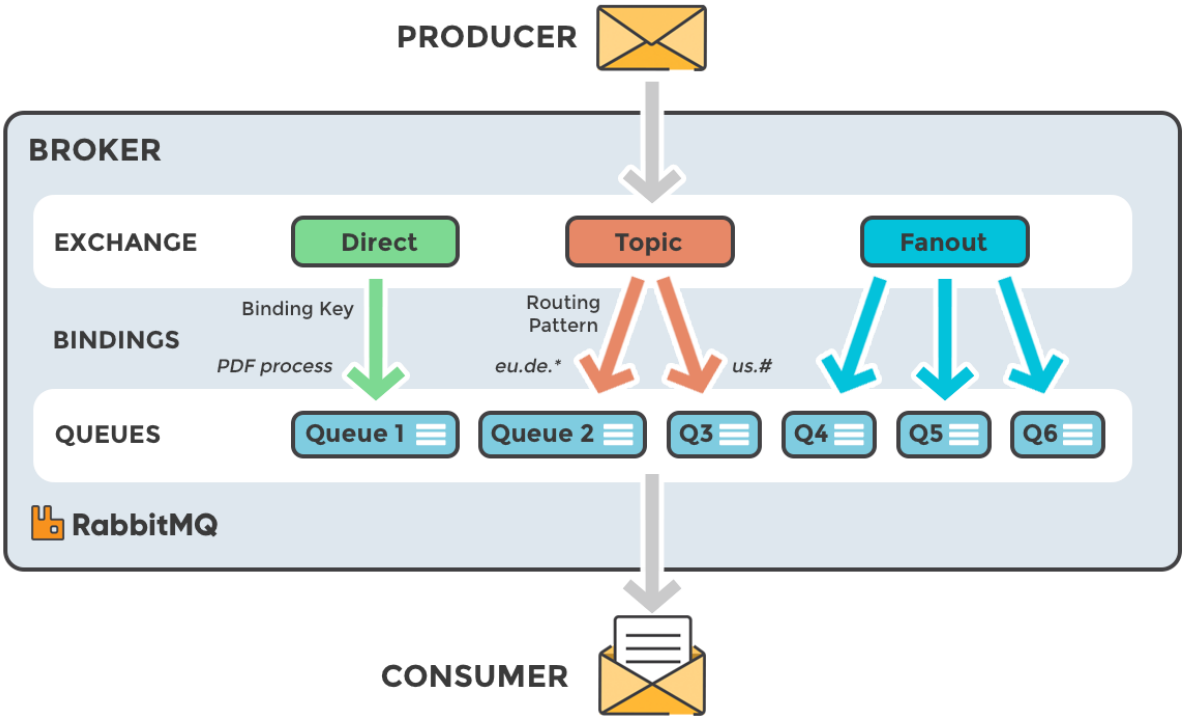




BASIC TERMINOLOGY

Messaging:

1. Producer
2. Consumer
3. Message Broker (RabbitMQ)



- RabbitMQ calls the **Queue** pattern of messaging: "**Direct** Exchange Routing."

- The **Fanout** Exchange pattern in RabbitMQ distributes a copy of every message to every consumer by giving each consumer a copy of the full message queue.

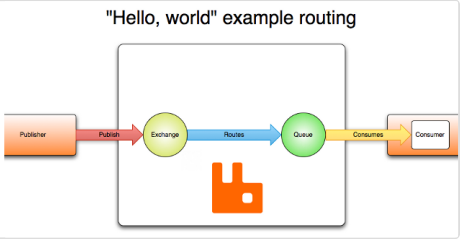
- The **Topic** Exchange pattern in RabbitMQ (what we call **Pub/Sub**) allows consumers to "subscribe" to only specific message queues or "topics". It distributes a copy of every message in a given topic to every consumer that subscribes to that particular topic.

Read more in-depth:

AMQP 0-9-1 Model Explained

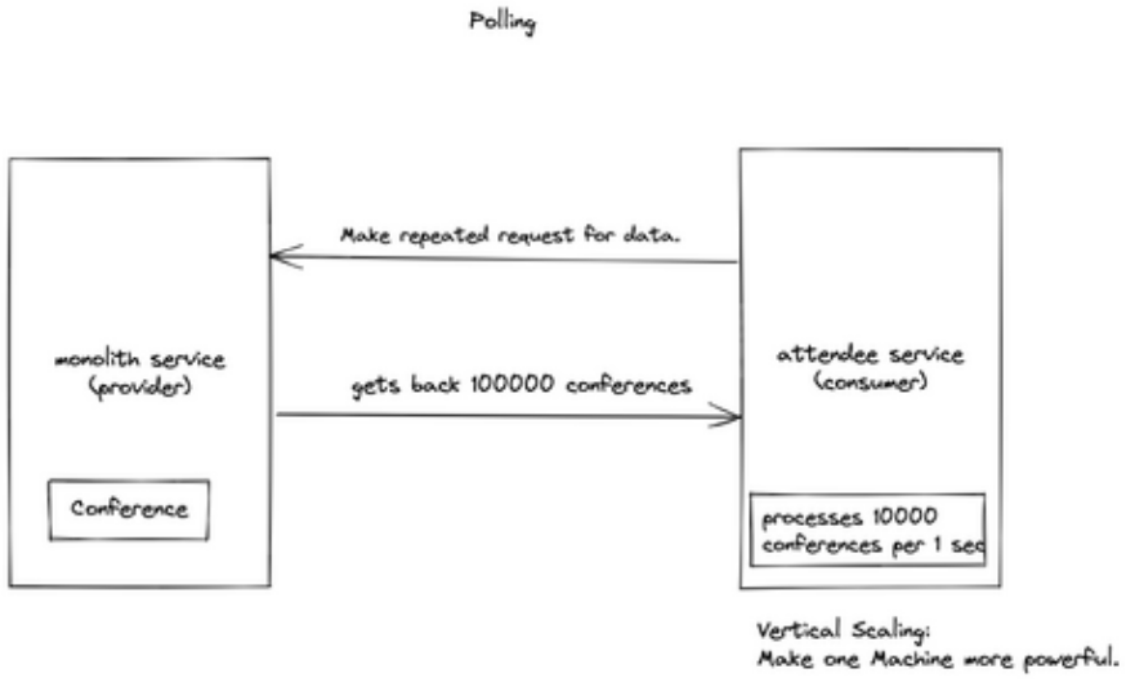
This guide provides an overview of the AMQP 0-9-1 protocol, one of the protocols supported by RabbitMQ. AMQP 0-9-1 (Advanced Message Queuing Protocol) is a messaging protocol that enables conforming client

https://www.rabbitmq.com/tutorials/amqp-concepts.html



A.

Polling

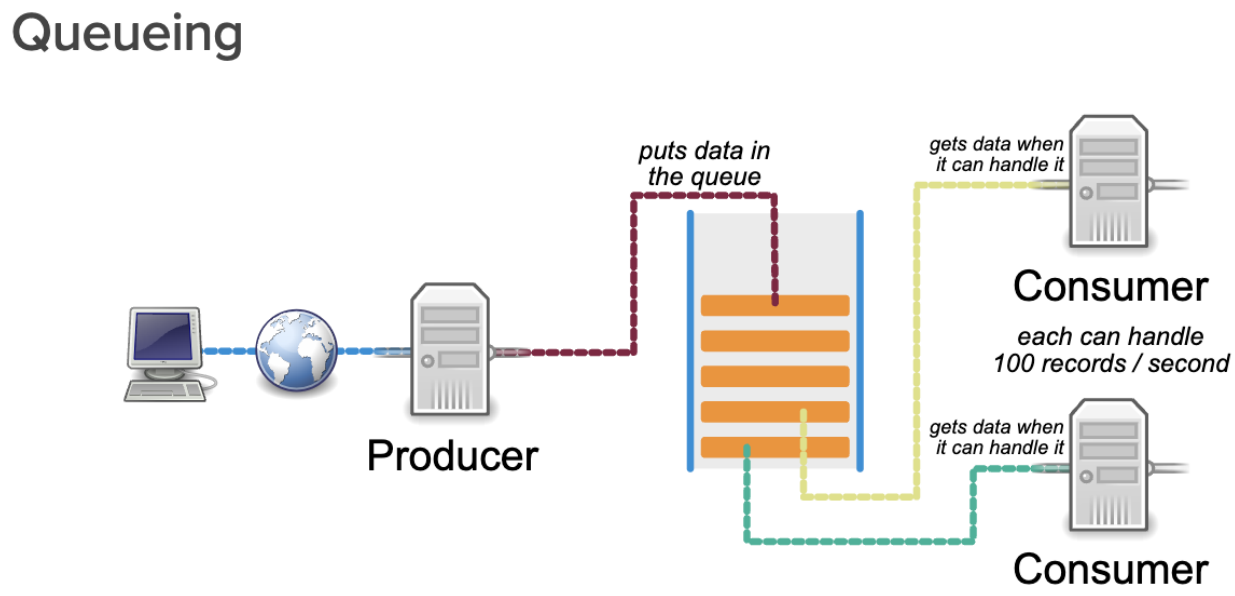


Warren Longmire's Ex-Calidraw from class shows us how polling works in our Week 1 Conference Go app.

Polling requires programming Consumers to repeatedly pull for data from Producers. It is the simplest way to get data in that it allows us to use our already-built APIs to synchronize data between different microservices.

- **Benefits:**
  - There's no extra middleware
- **Drawbacks:**
  - May end up with bottlenecks if the producer has too much new data for the consumers to deal with
  - If two services are trying to do the same thing, they may end up duplicating work or destroying each other's work

B.



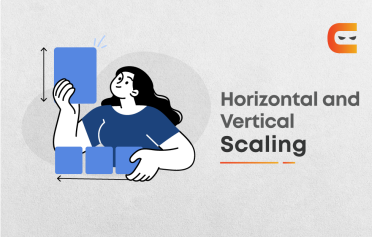
Queueing allows the producer to store messages in a queue. (RabbitMQ!) The consumers will read the messages, one-by-one, from the queue. Each consumer gets a message, works on it, then reads the next message from the queue to work on.

- **Benefit:** Can spread out work to multiple consumers all working on a single string of tasks. *(Horizontal scaling)*
- **Drawback:** To do this, you have another piece of middleware software that you have to become familiar with and support (RabbitMQ or Kafka in Module 3)

System Design: Horizontal and Vertical Scaling | Coding Ninjas Blog

READ NEXT Ninja recently completed the Android App Development Course from Coding Ninjas. To impress his friend Nina, he decided to develop a chatting app. Because of the extensive knowledge, he gained from the

https://www.codingninjas.com/blog/2021/08/25/system-design-horizontal-and-vertical-scaling/#:~:text=Horizontal%20Scaling,-Horizontal%20scaling%20expands&text=It%20does%20not%20change%20the,the%20amount%20of%20provisioned%20resources



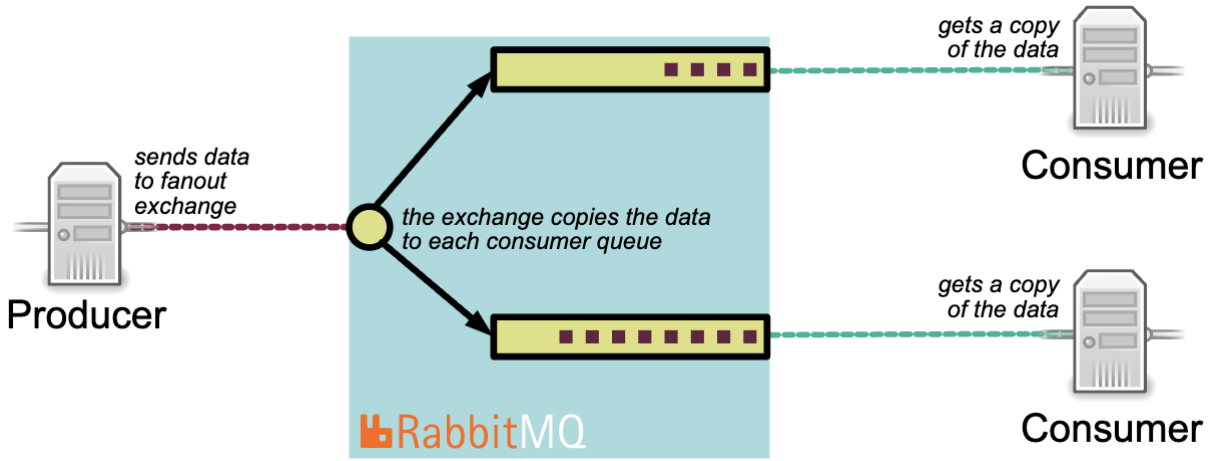
Cracking the System Design Interview [Theory Basics]

System design interviews are in general 45-60 minutes long where you are given a broad topic like "Design Twitter". You'll be expected to generate a high-level design, showing the different system

https://medium.com/geekculture/cracking-the-system-design-interview-theory-basics-c57f5326181b

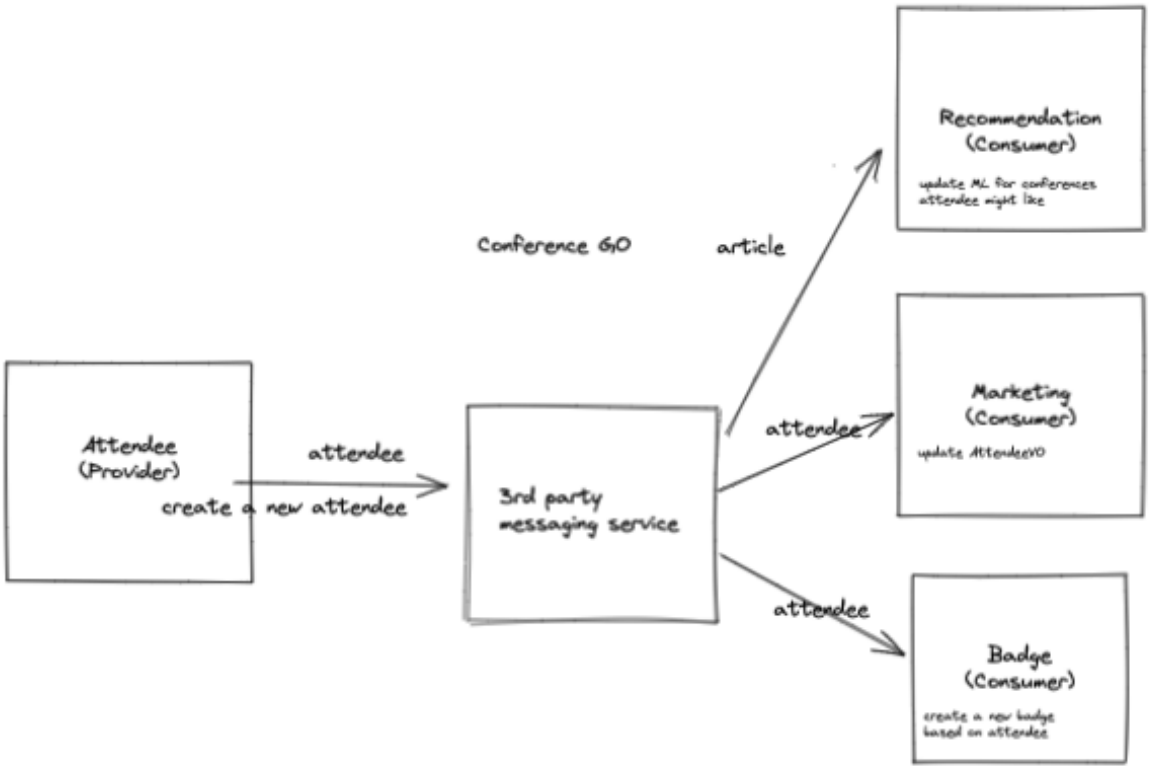
C.

Pub/Sub



Publish/subscribe is a way for a producer to create a message and send it to any consumer that's interested in it. The way that works is through something called a *topic*. Like a queue, a topic can have multiple consumers. But unlike a queue, every single consumer gets a copy of that data.

- **Benefit:** Low demands on the producer
- **Drawback:** If a consumer misses a message, usually it's gone forever. No other consumer will catch it and do the work instead.



Warren Longmire's diagrams for how RabbitMQ handles messaging inside of the ConferenceGO application.

When a consumer subscribes to the exchange, RabbitMQ creates an internal queue for that consumer. When the producer publishes data to the same exchange, RabbitMQ copies the data to the internal queues for each consumer. This way, each consumer gets their own copy of the data at their own speed.

❗ **Module 2 Assessment Focus:**

Message queues are often used where we want to delegate work from a service. In doing so, we want to ensure that the work is only executed one time. This makes it popular with microservices because it allows the team to scale "horizontally" based on the load of messages being queued and waiting to be processed. Once those messages are handled, the services can easily be dialed back down to save on running costs.

Pub sub should be used where we need a guarantee that each subscriber gets a copy of the message. Unlike other forms of messaging, a message will only be deleted if it's consumed by all subscribers to that category of messages. PubSub enables multiple consumers to receive each message in a given topic and also ensure that these messages are received by the consumers in the same order that they were produced. Given its specific order and multiple-recipient design, the pub-sub pattern is ideal for an environment where it's important that multiple consumers get the same message and in the right order.

RabbitMQ Lab

SIMPLIFIED INSTRUCTIONS:

```
# 1. Create a new Docker network:
docker network create --driver bridge rabbit-test1

# 2. Start RabbitMQ in a docker container:
docker run -d --name rabbitmq --hostname rabbitmq --network rabbit-test rabbitmq:3

# 3. From the root of the pubsub directory open two separate terminal windows and run:
docker run -it -v "$(pwd):/app" --network rabbit-test hello-rabbitmq
```

Run this with Producer.py and Consumer.py.

What is Pika?

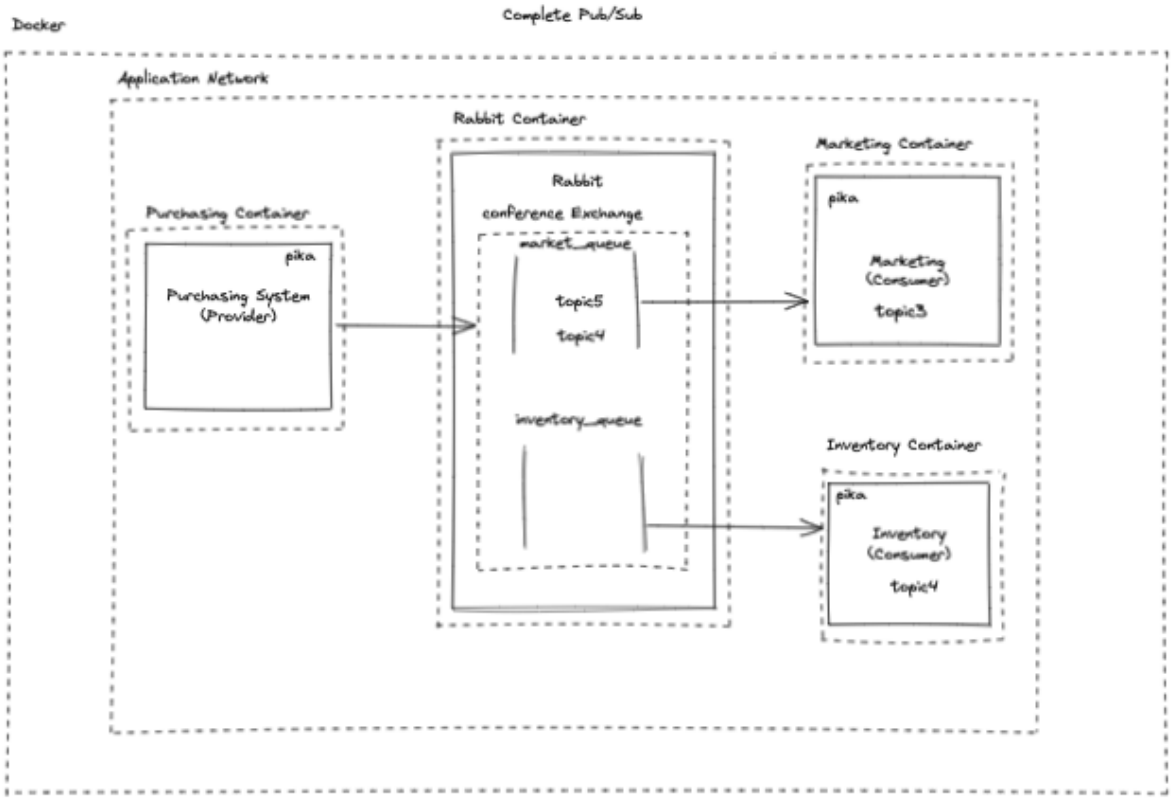
Pika is a Python library for interacting with RabbitMQ. You can use Pika to declare a queue, set up a publisher to send messages to the exchange, and set up a consumer to receive messages from the queue.

A popular alternative to Pika is a library called Celery, which allows for creating distributed task queues. *(Response to Garret's question)* See: StackOverflow response.

Catching Connection Errors in RabbitMQ for slow loading in Docker:

```
from pika.exceptions import AMQPConnectionError

...
while True:
    try:
        # ALL OF YOUR CODE THAT HANDLES READING FROM THE
        # QUEUES AND SENDING EMAILS
    except AMQPConnectionError:
        print("Could not connect to RabbitMQ")
        time.sleep(2.0)
```



Warren Longmire's diagram of how RabbitMQ exchanges messages using the Publisher/Subscribe model, operating as a microservice alongside several other microservices in Docker containers.

Docker Compose

Instead of running each Dockerfile inside of every microservice individually, you can use Docker Compose to run all of them at once.

Docker Compose is an **orchestration** tool that makes spinning up multi-container distributed applications with Docker a task that you can do with a single command, to spin up an entire app.

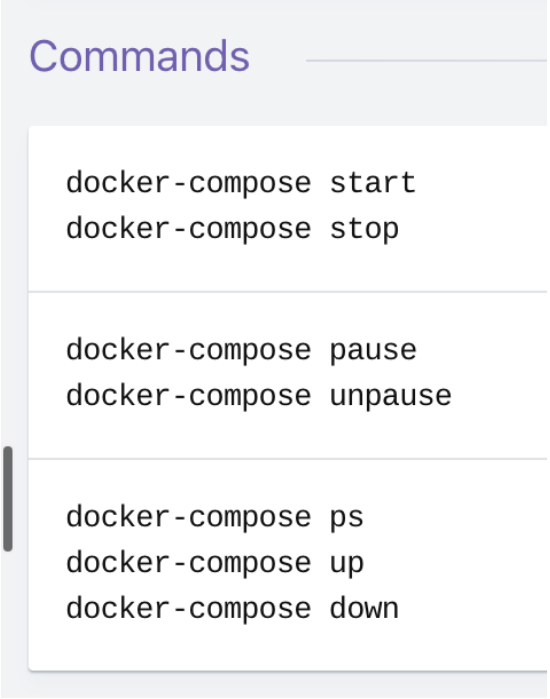
We tell Docker Compose which directory the code is in, the name of the *Dockerfile*, the ports, and the paths to use.

```
version: "3.7"
services:
  monolith:
    build:
      context: ./monolith
      dockerfile: ./Dockerfile.dev
    ports:
      - "8000:8000"
    volumes:
      - ./monolith:/app
    depends_on:
      - rabbitmq
  attendees_microservice:
    build:
      context: ./attendees_microservice
      dockerfile: ./Dockerfile.dev
    ports:
      - "8001:8001"
    volumes:
      - ./attendees_microservice:/app
  presentation_workflow:
    build:
      context: ./presentation_workflow
      dockerfile: ./Dockerfile.dev
    depends_on:
      - rabbitmq
    volumes:
      - ./presentation_workflow:/app
  rabbitmq:
    image: rabbitmq:3
```

The `image` entry tells Docker Compose to use an image. There's no customization that we need to do. We don't have to build our own image. We just want to use the RabbitMQ image.

Docker-Compose commands cheatsheet:

https://devhints.io/docker-compose



Check out the cheatsheet website (right).

Completed Lab:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e804942e-f3d1-418e-9cbd-72f5c5a976f3/conference-go-week-2.zip