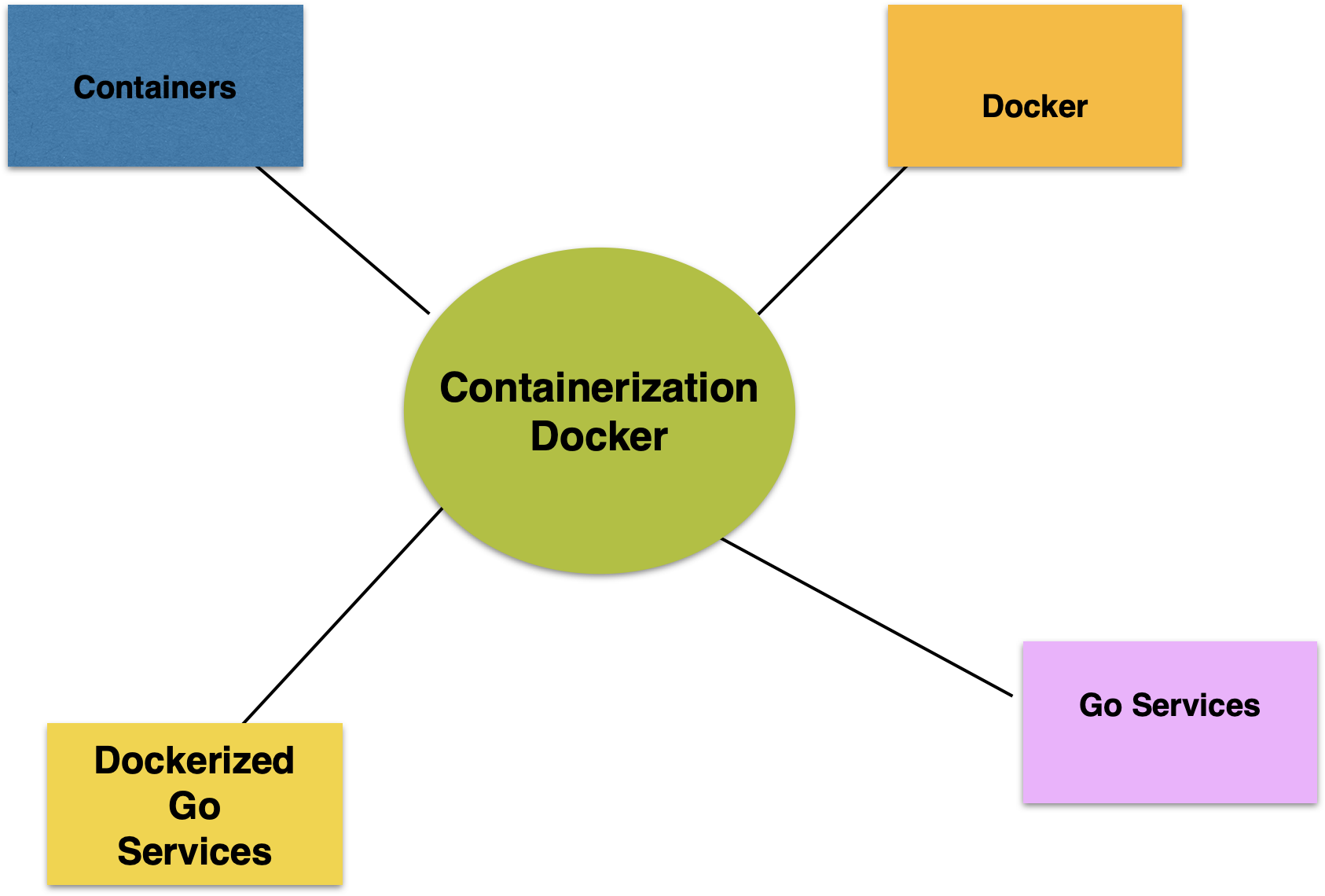
**C H A P T E R 10**

CHAPTER 10

# *Containerization and Docker*

# Introduction

In this chapter, Reader will understand containers and Docker. Reader will be able to develop services which can be containerized using Docker. Docker Images can be created and posted on the docker Hub for different Go services.



## Structure

The chapter covers the following topics:

* Containers
* Docker
* Go Services
* Dockerized Go Services

## Objectives

In this chapter, we are going to look at containers, docker, go services, and Dockerized Go services. Readers will be presented with basic examples using docker as a container and advance examples using docker-compose.

# Containers

There are many containers like Docker and Kubernetes evolving in the market. Enterprises are adopting containerization for easy deployment of microservices on the cloud and on premises. Virtual Machines were first in this market space. Developers had issues as VMs were non performant and overall time for initialization was high. Containers on the other hand were performant as they can be based on the required operating system with all dependent libraries and packages. They were consuming optimally the resources.

Docker is one of the containers which is used for microservices based application packaging and deploying the required dependencies. Docker can be based on Linux, MacOS, Windows, and third-party operating systems. In Linux, you can have different flavors like Ubuntu, CentOs, Redhat, and other vendor operating systems. The next level packaging is at the application or tech stack specific containers like tomcat, jetty, and spring-boot micro-containers. The advantage of docker like containers is to deploy them and test them as a single unit. The image instance can be deployed in the required operating system on the cloud or on-premises.

Containers provide capabilities to scale, perform, agile, control, isolate, and portable across different operating systems. L et us take an example where an enterprise wants to build a web application to enable selling online. This ecommerce application can be built on Go Language framework like Beego. It can expose REST api and React web application can access the REST api. These api are deployed on Beego web framework. We can use Linux ubuntu as the operating system. The image can have Go Language binaries and compiler setup. Beego framework can be installed on the image.

Once the web application with web pages on React and integration done with REST Api through javascrpt. Web application can be deployed on the image and tested by the QA engineers. The image can be uploaded to the docker hub which is the registry. On the cloud, you can access this image for deployment and execute the application. There is a huge cutdown in cost, time, and effort for testing, deployment, and project execution.

We can have performance testing on product environment with higher end RAM like 32 GB and have 4 VMs running the web application. 4 VMs can each have 8 GB as memory. On any cloud, this can be achieved without actual web app deployment. Docker container helps in managing the memory allocation very easily with Docker cluster environment. Docker container virtualizes the operating system and provides capabilities to manage CPU, Memory, Diskspace, Network, and IO.

Now let us first create a image in a container. We will be using Docker as the container. You need to register on hub.docker.com and download Docker software. You can first create a dockerfile.

**Dockerfile**

FROM busybox

CMD echo "Your first image on container is created"

You need to run these commands to build the image.

docker build -t bhagvanarch/docker-container-image .

docker run bhagvanarch/docker-container-image

The output of the above commands when executed is shown below:

**docker-container-image**

(base) apples-MacBook-Air:ContainerImage bhagvan.kommadi$ docker build -t bhagvanarch/docker-container-image .

Sending build context to Docker daemon 2.048kB

Step 1/2 : FROM busybox

latest: Pulling from library/busybox

809d8e20e203: Pull complete

Digest: sha256:2376a0c12759aa1214ba83e771ff252c7b1663216b192fbe5e0fb364e952f85c

Status: Downloaded newer image for busybox:latest

---> 5242710cbd55

Step 2/2 : CMD echo "Your first image on container is created"

---> Running in 3a60d1e48962

Removing intermediate container 3a60d1e48962

---> 265924f1a21b

Successfully built 265924f1a21b

Successfully tagged bhagvanarch/docker-container-image:latest

(base) apples-MacBook-Air:ContainerImage bhagvan.kommadi$ docker run bhagvanarch/docker-container-image

Your first image on container is created

(base) apples-MacBook-Air:ContainerImage bhagvan.kommadi$

As shown in the example, input for the docker process is the Docker file. Docker file has the steps and tasks defined for creating an image.

Containers are used not only for microservices based applications but also Domain driven design pattern-based apps. Container orchestration can be done by having multistage builds and multiple processes initialized by the container. It goes without saying, single monolithic application can be deployed in a container.

# Docker

Docker is a popular container which is used for enterprise app deployment on the cloud and on-premises. Most of the time developers and deployment engineers have arguments about where it was working vs currently it is not working here. Developers try their best in these arguments to show their environment is working fine. Deployment engineers strongly protest saying that it is not working on prod environment though the operating system and data base is same. This is where Docker comes for rescue. All the steps and tasks performed by the developer are captured in the Dockerfile. This Dockerfile is used as the input for docker executable to the deployment engineers for deploying on the production environment. Complex environment which has many steps and tasks to perform can be simplified by having a Dockerfile. Dockerfile typically has environmental variables, different

Go lang compiler versions, build process tasks, mounting of directories, and copying tasks.

Docker was created in a company Dotcloud. Dotcloud Inc., became Docker Inc., Docker was written in Go Language. Docker became popular after creation in 2013 as a Devops tool. Docker was implemented as the deployment tool for deploying enterprise apps on the cloud.

Docker is installed on developer’s desktop as host, daemon, client, and hub repository. Docker client and server communicate through client – server architecture.

Now let us first create an image in a docker container where we can write go lang programs. We will be using golang:1.8 version and the Docker as the container. You can first create a Dockerfile.

**Dockerfile**

FROM golang:1.8

WORKDIR .

COPY \*.go .

RUN go build -o first\_go .

CMD ./first\_go

You need to run these commands to build the image locally.

docker build -t first\_go\_image -f Dockerfile .

docker run -it first\_go\_image bash

The output of the above commands when executed is shown below:

**first\_go\_image**

(base) apples-MacBook-Air:DockerGoImage bhagvan.kommadi$ docker build -t first\_go\_image -f Dockerfile .

Sending build context to Docker daemon 3.072kB

Step 1/5 : FROM golang:1.8

1.8: Pulling from library/golang

4176fe04cefe: Pull complete

851356ecf618: Pull complete

6115379c7b49: Pull complete

69914558965c: Pull complete

b108f9aa98db: Pull complete

df7abcd2981e: Pull complete

3e60cb3f592b: Pull complete

Digest: sha256:f0b5dab7581eddb49dabd1d1b9aa505ca3edcdf79a66395b5bfa4f3c036b49ef

Status: Downloaded newer image for golang:1.8

---> 0d283eb41a92

Step 2/5 : WORKDIR .

---> Running in 0ddfa4086d3d

Removing intermediate container 0ddfa4086d3d

---> dce5d65f2716

Step 3/5 : COPY \*.go .

---> 778d859dd86d

Step 4/5 : RUN go build -o first\_go .

---> Running in 7ae0aa734964

Removing intermediate container 7ae0aa734964

---> 73183a49f1f3

Step 5/5 : CMD ./first\_go

---> Running in 8cd876c87cd1

Removing intermediate container 8cd876c87cd1

---> c6a5e2c080fe

Successfully built c6a5e2c080fe

Successfully tagged first\_go\_image:latest

(base) apples-MacBook-Air:DockerGoImage bhagvan.kommadi$ ls

Dockerfile first\_go\_program.go

(base) apples-MacBook-Air:DockerGoImage bhagvan.kommadi$ docker run first\_go\_image

Running my first Go Lang program

(base) apples-MacBook-Air:DockerGoImage bhagvan.kommadi$ docker run -it first\_go\_image bash

root@ea671e472228:/go# ls

bin first\_go first\_go\_program.go src

root@ea671e472228:/go# ./first\_go

Running my first Go Lang program

root@ea671e472228:/go#

Now let us look at Docker Registry.

## Docker Registry

Docker Registry is used for storing the docker images. The images can be stored in repos. These repos can be public or private. Developers can download the images for development. QA engineers can do the same for testing. Docker Hub is a docker registry which is public.

Each cloud provider like AWS, Azure, Google Cloud, and others provide a container registry. Different companies can have private container registry.

Now let us create the image on the docker hub (registry) where we can write go lang programs. We will be using golang:1.8 version and the Docker as the container. You can first create a Dockerfile.

**Dockerfile**

FROM golang:1.8

WORKDIR .

COPY \*.go .

RUN go build -o first\_go .

CMD ./first\_go

You need to run these commands to build the image on the docker hub.

docker build -t bhagvanarch/first\_go\_image -f Dockerfile .

docker run -it bhagvanarch/first\_go\_image bash

The output of the above commands when executed is shown below:

**Bhagvanarch/first\_go\_image**

(base) apples-MacBook-Air:DockerGoImage bhagvan.kommadi$ docker build -t bhagvanarch/first\_go\_image -f Dockerfile .

Sending build context to Docker daemon 3.072kB

Step 1/5 : FROM golang:1.8

---> 0d283eb41a92

Step 2/5 : WORKDIR .

---> Using cache

---> dce5d65f2716

Step 3/5 : COPY \*.go .

---> Using cache

---> 778d859dd86d

Step 4/5 : RUN go build -o first\_go .

---> Using cache

---> 73183a49f1f3

Step 5/5 : CMD ./first\_go

---> Using cache

---> c6a5e2c080fe

Successfully built c6a5e2c080fe

Successfully tagged bhagvanarch/first\_go\_image:latest

(base) apples-MacBook-Air:DockerGoImage bhagvan.kommadi$ docker run -it bhagvanarch/first\_go\_image bash

root@6bd14be8fd6e:/go# ls

bin first\_go first\_go\_program.go src

root@6bd14be8fd6e:/go# ./first\_go

Running my first Go Lang program

root@6bd14be8fd6e:/go#

## Docker Compose

Docker Compose is used for building images with multiple containers. Multi stage builds and multiple processes involving different containers can help in creating and packaging multiple container application. In the example above, where we were creating an enterprise web application based on Beego Web framework and React, we can use Docker Compose for multiple containers having database MySQL, Beego based rest api, and react web application. These containers are configured in YAML file. Docker-compose can run in the background if you use -d flag.

Now let us create the image using docker-compose where we can write go lang programs. We will be using golang:1.8 version and the Docker as the container. You can first create a Dockerfile.

**Dockerfile**

FROM golang:1.8

WORKDIR .

COPY \*.go .

RUN go build -o first\_go .

CMD ./first\_go

Now let us create the docker-compose.yml

**docker-compose.yml**

version: '2'

services:

first-docker-compose-go:

build: .

environment:

- TEST\_ENV=test

You need to run these commands to build the image on the docker hub.

docker-compose up

docker-compose run first-docker-compose-go bash

The output of the above commands when executed is shown below:

**first\_docker\_compose\_go\_image**

(base) apples-MacBook-Air:DockerComposeGoImage bhagvan.kommadi$ docker-compose up

Creating network "dockercomposegoimage\_default" with the default driver

Building first-docker-compose-go

Step 1/5 : FROM golang:1.8

---> 0d283eb41a92

Step 2/5 : WORKDIR .

---> Using cache

---> dce5d65f2716

Step 3/5 : COPY \*.go .

---> c2320d357ae4

Step 4/5 : RUN go build -o first\_go .

---> Running in c16bef07b86e

Removing intermediate container c16bef07b86e

---> e83e41da70d3

Step 5/5 : CMD ./first\_go

---> Running in 0b798073e2c4

Removing intermediate container 0b798073e2c4

---> fab530f81040

Successfully built fab530f81040

Successfully tagged dockercomposegoimage\_first-docker-compose-go:latest

WARNING: Image for service first-docker-compose-go was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.

Creating dockercomposegoimage\_first-docker-compose-go\_1 ... done

Attaching to dockercomposegoimage\_first-docker-compose-go\_1

first-docker-compose-go\_1 | Running my first Go Lang program

dockercomposegoimage\_first-docker-compose-go\_1 exited with code 0

(base) apples-MacBook-Air:DockerComposeGoImage bhagvan.kommadi$ docker-compose run first-docker-compose-go bash

root@3bda1d8e321a:/go# ls

bin first\_go first\_go\_program.go src

root@3bda1d8e321a:/go# ./first\_go

Running my first Go Lang program

root@3bda1d8e321a:/go#

## Docker Networks

You can create a container and build images in any operating system using docker or docker-compose. When you can execute an image, you can see there is an IP address for each image running on your computer. You can have a docker network setup and have docker images get assigned to the network images. Docker network types are default, user-defined, and overlay docker network.

You can specify the network parameter while creating a container. Default network created will be a container on a bridge network. Containers can talk to each other using IP addresses. You can have multiple containers running on one host. Bridge network has one host. If you want to have multiple Docker hosts, you can use the network parameter and assign host value. Host network stack will have multiple host details when you create them. Containers which are part of the host network can talk to the host network interfaces. You can have no network assigned if you pass value none to the network parameter. Container will not have any IP address assigned.

# Go Services

You can build microservices with Go Language. Moving away from monolithic approach of building apps, Microservices architecture is known for building the application ground up with micro-services. Microservices can talk to each other in different protocols like HTTP, AMQP, HTTPS, and web Sockets. Every microservice can be deployed and executed as a separate process. Every microservice will have its own data model, business logic, rules, and data storage mechanism. The backend database can be No SQL or relational database. Microservices architecture is getting popular because of its benefits which are related to manageability, scalability, performance, and time to market delivery.

Let us now discuss about building REST api (Microservice) with MQ.

We built this Go Lang microservice in chapter 4. In this context of build Go Service docker, let us relook at how to develop Rest API which can interact with message queues. We are going to use Gin and Rabbit MQ.

Now let us look at the code for developing Rest api using Gin and RabbitMQ.

**main.go**

package main

import (

"fmt"

"github.com/gin-gonic/gin"

"github.com/rs/zerolog/log"

"rest\_api\_mq/producer/config"

"rest\_api\_mq/producer/utils"

)

func init() {

mode := utils.GetEnvVar("GIN\_MODE")

gin.SetMode(mode)

}

func main() {

appGin := config.CreateApp()

addrGin := utils.GetEnvVar("GIN\_ADDR")

portGin := utils.GetEnvVar("GIN\_PORT")

log.Info().Msgf("App is up at http//:%s:%s", addrGin, portGin)

if error := appGin.Run(fmt.Sprintf("%s:%s", addrGn, portGin)); error != nil {

log.Fatal().Err(error).Msg("Http Server setup failed")

}

}

Now let us look at the createApp to see how the routes are setup through routers.

**Creating\_app.go**

package config

import (

"github.com/gin-gonic/gin"

"github.com/rs/zerolog/log"

"rest\_api\_mq/producer/middlewares"

"rest\_api\_mq/producer/routers"

)

func CreateApp() \*gin.Engine {

log.Info().Msg("service starting")

app := gin.New()

app.Use(gin.Recovery())

app.SetTrustedProxies(nil)

log.Info().Msg(" cors, request id, request logging middleware added")

app.Use(middlewares.CORSMiddleware(), middlewares.RequestID(), middlewares.RequestLogger())

log.Info().Msg("routers setup")

routers.SetupRouters(app)

return app

}

Routers package has the routes mapped to methods which talk to the message queue.

Get /ping controllers.ping

Post /publish/example controllers.Example

**setup.go**

package routers

import (

"github.com/gin-gonic/gin"

"rest\_api\_mq/producer/controllers"

)

func CreateRouters(engine \*gin.Engine) {

version1 := engine.Group("/v1")

{

version1.GET("/ping", controllers.Ping)

version1.POST("/publish/example", controllers.Example)

}

}

Controllers package has the message publisher configured to publish messages to a message queue.

**controllers.go**

package controllers

import (

"net/http"

"github.com/gin-gonic/gin"

"github.com/rs/zerolog/log"

"rest\_api\_mq/producer/environment"

"rest\_api\_mq/producer/models"

"rest\_api\_mq/producer/utils"

)

func Example(context \*gin.Context) {

var msg models.Message

request\_id := context.GetString("x-request-id")

if binderr := context.ShouldBindJSON(&msg); binderr != nil {

log.Error().Err(binderr).Str("request\_id", request\_id).

Msg("Error occurred while binding request data")

context.JSON(http.StatusUnprocessableEntity, gin.H{

"message": binderr.Error(),

})

return

}

connectionString := utils.GetEnvVar("RMQ\_URL")

producer := utils.MessagePublisher{

environment.EXAMPLE\_QUEUE,

connectionString,

}

producer.PublishMessage("text/plain", []byte(msg.Message))

context.JSON(http.StatusOK, gin.H{

"response": "Message received from Rest API",

})

}

MessagePublisher is a struct defined in utils package. It has a method to publish message to a queue defined in the environment.

**Publisher.go**

package utils

import (

"github.com/rs/zerolog/log"

"github.com/streadway/amqp"

)

type MessagePublisher struct {

Queue string

ConnectionString string

}

func (x MessagePublisher) OnError(err error, msg string) {

if err != nil {

log.Err(err).Msgf("Publishing message error '%s' queue. Error message: %s", publisher.Queue, msg)

}

}

func (publisher MessagePublisher) PublishMessage(contentType string, body []byte) {

conn, error := amqp.Dial(publisher.ConnectionString)

publisher.OnError(error, "RabbitMQ not connected")

defer conn.Close()

channel, err := conn.Channel()

publisher.OnError(err, "Channel not opened")

defer channel.Close()

q, error := channel.QueueDeclare(

publisher.Queue,

false,

false,

false,

false,

nil,

)

publisher.OnError(error, "Queue Not declared")

error = channel.Publish(

"",

q.Name,

false,

false,

amqp.Publishing{

ContentType: contentType,

Body: body,

})

publisher.OnError(error, "message not published")

}

Now let us look at the consumer package.

**main.go**

package main

import (

"rest\_api\_mq/consumer/environment"

"rest\_api\_mq/consumer/handlers"

"rest\_api\_mq/consumer/utils"

)

func main() {

connectionString := utils.GetEnvVar("RMQ\_URL")

messageQueue := utils.MessageConsumer{

enviornment.EXAMPLE\_QUEUE,

connectionString,

handlers.HandleMessaage,

}

forever := make(chan bool)

go messageQueue.Consume()

<-forever

}

MessageConsumer is created from utils package.

**Message\_consumer.go**

package utils

import (

"github.com/rs/zerolog/log"

"github.com/streadway/amqp"

)

type MessageConsumer struct {

Queue string

ConnectionString string

MsgHandler func(queue string, msg amqp.Delivery, err error)

}

func (consumer MessageConsumer) OnError(errors error, msg string) {

if errors != nil {

consumer.MsgHandler(consumer.Queue, amqp.Delivery{}, errors)

}

}

func (consumer MessageConsumer) Consume() {

conn, err := amqp.Dial(consumer.ConnectionString)

consumer.OnError(err, "Failed to connect to RabbitMQ")

defer conn.Close()

channel, err := conn.Channel()

consumer.OnError(err, "Failed to open a channel")

defer channel.Close()

q, err := channel.QueueDeclare(

consumer.Queue,

false,

false,

false,

false,

nil,

)

consumer.OnError(err, "Failed to declare a queue")

msgs, err := channel.Consume(

q.Name,

"",

true,

false,

false,

false,

nil,

)

consumer.OnError(err, "Failed to register a consumer")

forever := make(chan bool)

go func() {

for delivery := range msgs {

consumer.MsgHandler(consumer.Queue, delivery, nil)

}

}()

log.Info().Msgf("Started listening- messages from '%s' queue", consumer.Queue)

<-forever

}

Environment package has the constants define for Environment file,directory, and the message queue.

**constants.go**

package environment

const ENV\_FILE = ".env"

const ENV\_FILE\_DIRECTORY = "."

const EXAMPLE\_QUEUE = "message\_queue"

.env will Rabbit message queue configuration URL and Log level.

**.env**

LOG\_LEVEL = debug

RMQ\_URL = amqp://guest:guest@localhost:5000/

Each microservice is shown above is an independent deployable unit which can be tested and loosely coupled. Microservice can consist of small services built and used in the framework for different features. A small team can build microservices architecture based application very easily and quickly. Agile process and microservices go together well in making the development of features meet the time to market. Polyglot microservices architecture style embraces different tech stacks like java, python, go lang, php, and .net.

Microservice architecture provides benefits like good readability of the code, components decentralization, continuous integration, test automation, continuous deployment, and domain driven design.

# Dockerized Go Services

Now let us look how docker can help the deployment of the go lang microservices more effective. In the example below we will look at how to develop Rest API (dockerized microservice) which can interact with message queues. We are going to use Gin and Rabbit MQ.

Environment package has the constants define for Environment file,directory, and the message queue.

**constants.go**

package environment

const ENV\_FILE = ".env"

const ENV\_FILE\_DIRECTORY = "."

const EXAMPLE\_QUEUE = "message\_queue"

.env will Rabbit message queue configuration URL and Log level.

**.env**

LOG\_LEVEL = debug

RMQ\_URL = amqp://guest:guest@localhost:5000/

**docker-compose.yaml**

networks:

rabbitmq-example:

driver: bridge

services:

rabbitmq:

image: 'rabbitmq:3-management'

networks:

- rabbitmq-example

volumes:

- ./rabbit-mq/rabbitmq.conf:/etc/rabbitmq/rabbitmq.conf:ro

ports:

- "8080:15672"

healthcheck:

test: [ "CMD", "rabbitmqctl", "status"]

interval: 5s

timeout: 15s

retries: 5

producer:

build: ./producer

ports:

- "5050:5050"

networks:

- rabbitmq-example

depends\_on:

- rabbitmq

environment:

GIN\_MODE: "release"

GIN\_HTTPS: "false"

GIN\_ADDR: "0.0.0.0"

GIN\_PORT: "5050"

LOG\_LEVEL: "debug"

RMQ\_URL: "amqp://guest:guest@rabbitmq:5673/"

consumer:

build: ./consumer

networks:

- rabbitmq-example

depends\_on:

- rabbitmq

restart: on-failure

environment:

LOG\_LEVEL: "debug"

RMQ\_URL: "amqp://guest:guest@rabbitmq:5673/"

You can now compile and run the rabbitmq, producer, and consumer services in the docker. The command to run the docker is shown below:

docker-compose up

The output will be as shown below:

(base) apples-Air:rest\_api\_mq bhagvan.kommadi$ docker-compose up

Starting rest\_api\_mq\_rabbitmq\_1 ... done

Starting rest\_api\_mq\_consumer\_1 ... done

Starting rest\_api\_mq\_producer\_1 ... done

Attaching to rest\_api\_mq\_rabbitmq\_1, rest\_api\_mq\_consumer\_1, rest\_api\_mq\_producer\_1

consumer\_1 | {"level":"debug","error":"open .env: no such file or directory","time":"2023-05-01T16:30:42Z","message":"Error occurred while reading env file, might fallback to OS env config"}

consumer\_1 | {"level":"error","host":"d5cf6a8361a5","service":"rabbitmq-consumer","error":"dial tcp 172.19.0.2:5673: connect: connection refused","time":1682958642,"caller":"/go/src/rmq-consumer/handlers/example.go:10","message":"Error occurred in RMQ consumer"}

consumer\_1 | {"level":"info","host":"d5cf6a8361a5","service":"rabbitmq-consumer","time":1682958642,"caller":"/go/src/rmq-consumer/handlers/example.go:12","message":"Message received on 'example' queue: "}

consumer\_1 | panic: runtime error: invalid memory address or nil pointer dereference

consumer\_1 | panic: runtime error: invalid memory address or nil pointer dereference

consumer\_1 | [signal SIGSEGV: segmentation violation code=0x1 addr=0x138 pc=0x5f364f]

consumer\_1 |

consumer\_1 | goroutine 19 [running]:

consumer\_1 | github.com/streadway/amqp.(\*Connection).IsClosed(...)

consumer\_1 | /go/pkg/mod/github.com/streadway/amqp@v1.0.0/connection.go:355

consumer\_1 | github.com/streadway/amqp.(\*Connection).Close(0xc0000a40d8?)

consumer\_1 | /go/pkg/mod/github.com/streadway/amqp@v1.0.0/connection.go:323 +0x2f

consumer\_1 | panic({0x6e5820, 0x967f30})

consumer\_1 | /usr/local/go/src/runtime/panic.go:884 +0x213

consumer\_1 | github.com/streadway/amqp.(\*Connection).allocateChannel(0x0)

consumer\_1 | /go/pkg/mod/github.com/streadway/amqp@v1.0.0/connection.go:596 +0x5b

consumer\_1 | github.com/streadway/amqp.(\*Connection).openChannel(0x0?)

consumer\_1 | /go/pkg/mod/github.com/streadway/amqp@v1.0.0/connection.go:626 +0x25

consumer\_1 | github.com/streadway/amqp.(\*Connection).Channel(...)

consumer\_1 | /go/pkg/mod/github.com/streadway/amqp@v1.0.0/connection.go:653

consumer\_1 | github.com/lakhinsu/rabbitmq-go-example/consumer/utils.RMQConsumer.Consume({{0x730ea2, 0x7}, {0xc00001e008, 0x21}, 0x761908})

consumer\_1 | /go/src/rmq-consumer/utils/rmq\_consumer.go:25 +0x168

consumer\_1 | created by main.main

consumer\_1 | /go/src/rmq-consumer/main.go:20 +0xf9

producer\_1 | {"level":"info","host":"6b7870107c71","service":"rabbitmq-producer","time":1682958643,"caller":"/go/src/rmq-producer/app/setup\_app.go:11","message":"Initializing service"}

producer\_1 | {"level":"info","host":"6b7870107c71","service":"rabbitmq-producer","time":1682958643,"caller":"/go/src/rmq-producer/app/setup\_app.go:19","message":"Adding cors, request id and request logging middleware"}

producer\_1 | {"level":"info","host":"6b7870107c71","service":"rabbitmq-producer","time":1682958643,"caller":"/go/src/rmq-producer/app/setup\_app.go:22","message":"Setting up routers"}

producer\_1 | {"level":"info","host":"6b7870107c71","service":"rabbitmq-producer","time":1682958643,"caller":"/go/src/rmq-producer/main.go:38","message":"Starting service on http//:0.0.0.0:5050"}

rest\_api\_mq\_consumer\_1 exited with code 2

rest\_api\_mq\_consumer\_1 exited with code 2

rest\_api\_mq\_consumer\_1 exited with code 2

rabbitmq\_1 | 2023-05-01 16:30:56.024246+00:00 [notice] <0.44.0> Application syslog exited with reason: stopped

rabbitmq\_1 | 2023-05-01 16:30:56.039213+00:00 [notice] <0.234.0> Logging: switching to configured handler(s); following messages may not be visible in this log output

rabbitmq\_1 | 2023-05-01 16:30:56.076366+00:00 [notice] <0.234.0> Logging: configured log handlers are now ACTIVE

rest\_api\_mq\_consumer\_1 exited with code 2

**rest\_api\_mq\_consumer\_1 exited with code 2**

rabbitmq\_1 | 2023-05-01 16:31:08.476556+00:00 [info] <0.234.0> ra: starting system quorum\_queues

rabbitmq\_1 | 2023-05-01 16:31:08.476769+00:00 [info] <0.234.0> starting Ra system: quorum\_queues in directory: /var/lib/rabbitmq/mnesia/rabbit@c493d0d25e3d/quorum/rabbit@c493d0d25e3d

rabbitmq\_1 | 2023-05-01 16:31:08.484163+00:00 [info] <0.304.0> ra system 'quorum\_queues' running pre init for 0 registered servers

rabbitmq\_1 | 2023-05-01 16:31:08.489010+00:00 [info] <0.306.0> ra: meta data store initialised for system quorum\_queues. 0 record(s) recovered

rabbitmq\_1 | 2023-05-01 16:31:08.490866+00:00 [notice] <0.314.0> WAL: ra\_log\_wal init, open tbls: ra\_log\_open\_mem\_tables, closed tbls: ra\_log\_closed\_mem\_tables

rabbitmq\_1 | 2023-05-01 16:31:08.498754+00:00 [info] <0.234.0> ra: starting system coordination

rabbitmq\_1 | 2023-05-01 16:31:08.498914+00:00 [info] <0.234.0> starting Ra system: coordination in directory: /var/lib/rabbitmq/mnesia/rabbit@c493d0d25e3d/coordination/rabbit@c493d0d25e3d

rabbitmq\_1 | 2023-05-01 16:31:08.506267+00:00 [info] <0.327.0> ra system 'coordination' running pre init for 0 registered servers

rabbitmq\_1 | 2023-05-01 16:31:08.511189+00:00 [info] <0.329.0> ra: meta data store initialised for system coordination. 0 record(s) recovered

rabbitmq\_1 | 2023-05-01 16:31:08.512859+00:00 [notice] <0.338.0> WAL: ra\_coordination\_log\_wal init, open tbls: ra\_coordination\_log\_open\_mem\_tables, closed tbls: ra\_coordination\_log\_closed\_mem\_tables

rabbitmq\_1 | 2023-05-01 16:31:08.524814+00:00 [info] <0.234.0>

rabbitmq\_1 | 2023-05-01 16:31:08.524814+00:00 [info] <0.234.0> Starting RabbitMQ 3.11.14 on Erlang 25.3.1 [jit]

rabbitmq\_1 | 2023-05-01 16:31:08.524814+00:00 [info] <0.234.0> Copyright (c) 2007-2023 VMware, Inc. or its affiliates.

rabbitmq\_1 | 2023-05-01 16:31:08.524814+00:00 [info] <0.234.0> Licensed under the MPL 2.0. Website: https://rabbitmq.com

rabbitmq\_1 |

rabbitmq\_1 | ## ## RabbitMQ 3.11.14

rabbitmq\_1 | ## ##

rabbitmq\_1 | ########## Copyright (c) 2007-2023 VMware, Inc. or its affiliates.

rabbitmq\_1 | ###### ##

rabbitmq\_1 | ########## Licensed under the MPL 2.0. Website: https://rabbitmq.com

rabbitmq\_1 |

rabbitmq\_1 | Erlang: 25.3.1 [jit]

rabbitmq\_1 | TLS Library: OpenSSL - OpenSSL 3.0.8 7 Feb 2023

rabbitmq\_1 | Release series support status: supported

rabbitmq\_1 |

rabbitmq\_1 | Doc guides: https://rabbitmq.com/documentation.html

rabbitmq\_1 | Support: https://rabbitmq.com/contact.html

rabbitmq\_1 | Tutorials: https://rabbitmq.com/getstarted.html

rabbitmq\_1 | Monitoring: https://rabbitmq.com/monitoring.html

rabbitmq\_1 |

rabbitmq\_1 | Logs: /var/log/rabbitmq/rabbit@c493d0d25e3d\_upgrade.log

rabbitmq\_1 | <stdout>

rabbitmq\_1 |

rabbitmq\_1 | Config file(s): /etc/rabbitmq/rabbitmq.conf

rabbitmq\_1 | /etc/rabbitmq/conf.d/10-defaults.conf

rabbitmq\_1 |

rabbitmq\_1 | Starting broker...2023-05-01 16:31:08.544582+00:00 [info] <0.234.0>

rabbitmq\_1 | 2023-05-01 16:31:08.544582+00:00 [info] <0.234.0> node : rabbit@c493d0d25e3d

rabbitmq\_1 | 2023-05-01 16:31:08.544582+00:00 [info] <0.234.0> home dir : /var/lib/rabbitmq

rabbitmq\_1 | 2023-05-01 16:31:08.544582+00:00 [info] <0.234.0> config file(s) : /etc/rabbitmq/rabbitmq.conf

rabbitmq\_1 | 2023-05-01 16:31:08.544582+00:00 [info] <0.234.0> : /etc/rabbitmq/conf.d/10-defaults.conf

rabbitmq\_1 | 2023-05-01 16:31:08.544582+00:00 [info] <0.234.0> cookie hash : +iEtGtoN/i1Z+tAFT6dYOw==

rabbitmq\_1 | 2023-05-01 16:31:08.544582+00:00 [info] <0.234.0> log(s) : /var/log/rabbitmq/rabbit@c493d0d25e3d\_upgrade.log

rabbitmq\_1 | 2023-05-01 16:31:08.544582+00:00 [info] <0.234.0> : <stdout>

rabbitmq\_1 | 2023-05-01 16:31:11.508103+00:00 [info] <0.234.0> Running boot step pre\_boot defined by app rabbit

rabbitmq\_1 | 2023-05-01 16:31:11.508391+00:00 [info] <0.234.0> Running boot step rabbit\_global\_counters defined by app rabbit

rabbitmq\_1 | 2023-05-01 16:31:11.513176+00:00 [info] <0.234.0> Running boot step rabbit\_osiris\_metrics defined by app rabbit

rabbitmq\_1 | 2023-05-01 16:31:11.513914+00:00 [info] <0.234.0> Running boot step rabbit\_core\_metrics defined by app rabbit

rabbitmq\_1 | 2023-05-01 16:31:11.518613+00:00 [info] <0.234.0> Running boot step rabbit\_alarm defined by app rabbit

rabbitmq\_1 | 2023-05-01 16:31:11.562041+00:00 [info] <0.390.0> Memory high watermark set to 799 MiB (838465945 bytes) of 1999 MiB (2096164864 bytes) total

rabbitmq\_1 | 2023-05-01 16:31:11.585910+00:00 [info] <0.392.0> Enabling free disk space monitoring (disk free space: 49152098304, total memory: 2096164864)

rabbitmq\_1 | 2023-05-01 16:31:11.586065+00:00 [info] <0.392.0> Disk free limit set to 50MB

rabbitmq\_1 | 2023-05-01 16:31:11.603828+00:00 [info] <0.234.0> Running boot step code\_server\_cache defined by app rabbit

rabbitmq\_1 | 2023-05-01 16:31:11.604246+00:00 [info] <0.234.0> Running boot step file\_handle\_cache defined by app rabbit

rabbitmq\_1 | 2023-05-01 16:31:11.954731+00:00 [info] <0.491.0> Ready to start client connection listeners

rabbitmq\_1 | 2023-05-01 16:31:11.968201+00:00 [info] <0.636.0> started TCP listener on [::]:5673

rabbitmq\_1 | completed with 4 plugins.

rabbitmq\_1 | 2023-05-01 16:31:13.401805+00:00 [info] <0.491.0> Server startup complete; 4 plugins started.

rabbitmq\_1 | 2023-05-01 16:31:13.401805+00:00 [info] <0.491.0> \* rabbitmq\_prometheus

rabbitmq\_1 | 2023-05-01 16:31:13.401805+00:00 [info] <0.491.0> \* rabbitmq\_management

rabbitmq\_1 | 2023-05-01 16:31:13.401805+00:00 [info] <0.491.0> \* rabbitmq\_web\_dispatch

rabbitmq\_1 | 2023-05-01 16:31:13.401805+00:00 [info] <0.491.0> \* rabbitmq\_management\_agent

rabbitmq\_1 | 2023-05-01 16:31:24.389343+00:00 [info] <0.649.0> accepting AMQP connection <0.649.0> (172.19.0.4:50164 -> 172.19.0.2:5673)

rabbitmq\_1 | 2023-05-01 16:31:24.407407+00:00 [info] <0.649.0> connection <0.649.0> (172.19.0.4:50164 -> 172.19.0.2:5673): user 'guest' authenticated and granted access to vhost '/'

rabbitmq\_1 | 2023-05-01 16:31:52.446273+00:00 [info] <0.685.0> accepting AMQP connection <0.685.0> (172.19.0.3:57038 -> 172.19.0.2:5673)

rabbitmq\_1 | 2023-05-01 16:31:52.468150+00:00 [info] <0.685.0> connection <0.685.0> (172.19.0.3:57038 -> 172.19.0.2:5673): user 'guest' authenticated and granted access to vhost '/'

producer\_1 | {"level":"info","host":"6b7870107c71","service":"rabbitmq-producer","request\_id":"10a68de1-f83d-4217-b750-af718bb569c2","client\_ip":"172.19.0.1","user\_agent":"PostmanRuntime/7.28.0","method":"POST","path":"/v1/publish/example","latency":0.0663015,"status":200,"time":1682958712,"caller":"/go/src/rmq-producer/middlewares/request\_logger.go:27"}

rabbitmq\_1 | 2023-05-01 16:31:52.487218+00:00 [info] <0.685.0> closing AMQP connection <0.685.0> (172.19.0.3:57038 -> 172.19.0.2:5673, vhost: '/', user: 'guest')

In the above example, we have used docker compose to build a go lang microservice based on MQ.

# Conclusion

In this chapter we have covered topics related to Containers, Docker, Docker registry, docker compose, docker networks, Go Lang Microservices, and Dockerized Go Lang microservices.

* Containers like Docker and Kubernetes are used to build and deploy software packages
* Docker has components like registry and networks to have multiple client-host models.
* Docker Compose helps in building multi-staged and multiple services.
* Go Lang tech stack can be used to build microservices.
* Dockerized Go services can be built using docker compose and having multiple go lang services.