Go

In this lesson, we'll be starting our discussion about Go and its relation to microservices.
Let's begin!

We'll cover the following

Microservices and the increasing popularity for Go

Go (https://golang.org/) is a programming language that is increasingly being used for microservices due to its great speed and support for concurrency. Concurrency enhances the efficiency of using multiple machines and cores.

Go also provides a powerful standard library for the creation of web

services.

For further details on how Go compares with the other 4 languages commonly used for implementing Microservices, visit this site (https://rubygarage.org/blog/top-languages-for-microservices).

Similar to Java, Go is based on the programming language C. However, in many areas Go is fundamentally different (http://dead10ck.github.io/2014/12/15/go-vs-c.html) from C.

Go code

The Go program below responds to HTTP requests with HTML code.

main.go

```
package main

import (
  "fmt"
  "log"
  //"time"
  "net/http"
)

func main() { http.Handle("/common/css/",
   http.StripPrefix("/common/css/",
   http.FileServer(http.Dir("/css"))))

  http.HandleFunc("/common/header", Header)
  http.HandleFunc("/common/footer", Footer)
  http.HandleFunc("/common/navbar", Navbar)
  fmt.Println("Starting up on 8180")
  log.Fatal(http.ListenAndServe(":8180", nil))
}

// Header and Navbar left out

func Footer(w http.ResponseWriter, req *http.Request) {
  fmt.Fprintln(w,
    `<script src="/common/css/bootstrap-3.3.7-dist/js/bootstrap.min.js" />`)
```

```
}
```

Line 3:

The key word `import` imports some libraries, among others for **HTTP**.

Line 10, and 23:

The program's `main` function defines which methods should respond to which **URLs**. For example, the method `Footer` (**line 23**) returns HTML code.

On the other hand, for the URL `/common/css` (**line 10**) the application delivers content from files.

> It is also very easy to implement a **REST** service with **Go**.

In addition, libraries like Go kit (https://github.com/go-kit/kit) offer many more functionalities to implement microservices.

Go build and compilation

Go compilers are particularly well suited for Docker environments because they can create **static binaries**.

> **Static binaries** do not require any further dependencies or a specific Linux distribution.

However, the applications must be compiled to **Linux binaries**. This requires a Go environment that can create Linux binaries.

Docker multi-stage builds

The example uses Docker multi stage builds. Such a build divides the build process of the Docker image into several stages.

First Stage

The **first stage** can compile the program in a Docker container with a Go build environment.

Second Stage

The **second stage** can execute the Go program in a Docker container as a runtime environment that contains only the compiled program.

Consequently, the runtime environment has no build tools and is therefore much smaller.

Docker multi stage builds are not very complicated, as a look at the *Dockerfile* shows:

Dockerfile

```
FROM golang:1.8.3-jessie

COPY /src/github.com/ewolff/common /go/src/github.com/ewolff/common
WORKDIR /go/src/github.com/ewolff/common
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o common .

FROM scratch
COPY bootstrap-3.3.7-dist /css/bootstrap-3.3.7-dist
COPY --from=0 /go/src/github.com/ewolff/common/common /
ENTRYPOINT ["/common"]
CMD []
EXPOSE 8180
```

Stage 0

Line 1

The **base image** `golang` contains the Go installation.

Line 3

The Go source code is `copied` and compiled into this image **(line 4/5)**. With that, stage 0 of the build is finished.

Stage 1

Stage 1 creates a new Docker image.

Line 7

The image, `scratch`, is an empty Docker image.

Line 8 and 9

The Dockerfile copies the `bootstrap` library (line 8) and the compiled Go binary from stage 0 (line 9) into this image.

The option `--from=0` indicates that the file common originates from **stage 0** of the Docker build.

Stage 2

Line 10

Finally, `ENTRYPOINT` defines the binary that is supposed to be started.

Line 11

`CMD` indicates that no options are to be passed to the binary at the start.

Normally, `ENTRYPOINT` would be a shell that starts the process that is configured with `CMD`. However, in the scratch image, there is no shell.

Line 12

According to Docker Documentation (https://docs.docker.com/engine/reference/builder/), “The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP. The default is TCP if the protocol is not specified.” So, port 8180 is specified here.

QUIZ

1 What are the advantages of using multistage builds to compile Go code?

A) They are much faster

B) They result in runtime environments that are smaller and more secure

C) They are not only the industry standard for Go, but Go code cannot be compiled any other way

Submit Answer

Question 1 of 3

0 attempted

Reset Quiz

In the *next lesson*, we'll discuss the potential of Go Lang in the implementation of microservices.

Back

Spring Boot for Microservices: New Mi...

Next

Go for Microservices?

Mark as Completed

Report an Issue