Advent of Go Microservices

Go microservices with DevOps emphasis

Tit Petric

Step by step guide for
Go microservice development

Advent of Go Microservices

Tit Petric

This book is for sale at http://leanpub.com/go-microservices

This version was published on 2020-02-17

Contents

Introduction

About me

I'm passionate about API development, good practices, performance optimizations and educating people to strive for quality. I have about two decades of experience writing and optimizing software, and often solve real problems in various programming languages, Go being my favorite.

You might know some of my work from:

- Author of API Foundations in Go[1],
- Author of 12 Factor Applications with Docker and Go[2],
- Blog author on scene-si.org[3]

Professionally I specialize in writing APIs in the social/media industry and for various content management products. Due to the public exposure of such APIs, it's performance characteristics are of paramount importance. I'm a solid MySQL DBA with experience on other databases as well.

I have strong opinions about PHP. I've spent the better part of two decades writing PHP code. I have love for Go, and thankfully that doesn't require me to have a super strong opinion, because it's harder to find really bad code. Honestly, the worst you can do is not handle errors and I'll be judging you. Disagree? Tweet @TitPetric[4] and let me know.

I've written a professional dedicated API framework which doubles as an software development kit for the Slovenian national TV and Radio station website, RTV Slovenia. The architecture being put in place with this book is meant to replace and implement the most critical microservices with.

Who is this book for?

This book is for everyone who writes applications for a living. I will cover a wide area of subjects that are related to effective microservice development.

You'll super love this book if you're into CI and DevOps. I hope to show you a bit of the world outside of Go, with examples of effective approaches you can use to develop your microservices.

[1]https://leanpub.com/api-foundations
[2]https://leanpub.com/12fa-docker-golang
[3]https://scene-si.org
[4]https://twitter.com/titpetric

It is the intent of the book to provide a daily exercise which will teach you things about Drone CI, bash, Makefiles, Docker, and anything else which might help you put together the foundation for your microservice or app.

In the book, I will cover these subjects:

1. Go: Introduction to Protobuf: Messages
2. Go: Introduction to Protobuf: Services
3. Go: Introduction to Protobuf: gRPC
4. Make: Dynamic Makefile targets
5. Bash: Poor mans code generation
6. Bash: Embedding files into Go
7. Go: Scaffolding database migrations
8. Drone CI: Testing database migrations
9. Go: Database first struct generation
10. Go: Generating database schema documentation
11. Go: Dependency injection with Wire
12. Docker: Building images with security in mind
13. Go: implementing a microservice
14. Docker/Bash: Issuing requests against our microservice
15. Go: Improving our database handling
16. Go: Instrumenting the HTTP service with Elastic APM
17. Go: Instrumenting the Database client with Elastic APM
18. Go: Stress testing our service
19. Go: Background jobs
20. Go: Optimizing requests with a queue

> The idea for the book is to publish content as an advent calendar. Starting December 1st and then every day until christmas, a new chapter will be added. The titles listed above are already done and are in the queue for publishing.

Following the step by step microservice development guide, we'll learn how to use proto files to scaffold your microservice, as well as how to use SQL-first code generation for our required data structures. We'll implement a Twitch RPC API with the minimal possible development effort, which will be focused on the ease of development for the final service.

When you finish with the exercises in this book, you will have a solid framework for microservice development and you can just keep adding new microservices to the mix.

How should I study it?

Just go through the book from the start to finish. If possible, try to do the exercises yourself not by copy pasting but by actually writing the code and snippets in the book, tailored to how you would lay out your project.

The work in individual chapters builds on what was done in the previous chapter. The examples are part of a step by step, chapter by chapter process, where we are developing the layout of the microservice, and implementing technical requirements for all future microservices.

Be sure to follow the Requirements section as you're working with the book.

Requirements

This is a book which gives you hands on instructions and examples of how to build a microservice repository. We will be using a modern stack of software that will be required in order to complete all the exercises.

Linux and Docker

The examples of the book rely on a recent docker-engine installation on a Linux host. Linux, while a soft requirement, is referenced many times in the book with examples on how to download software, or how we are building our services. You could do this on a Mac, but you might have to adjust some things as you move through the chapters.

- Docker (a recent version, I'd suspect anything after 18.09 is fine),
- Docker Compose (a recent version from their GitHub page),
- Drone CI (instructions to install later in the book),
- Various shell utilities and programs (awk, bash, sed, find, ls, make,…)

Please refer to the official docker installation instructions[5] on how to install a recent docker version, or install it from your package manager.

We will use Docker Compose to pull in images for databases and whatever we need. As we have a Docker-first workflow, this means we don't need to deal with pre-installing software.

Own hardware

The recommended configuration if you have your own hardware is:

- 2 CPU core,
- 2GB ram,
- 128GB disk (SSD)

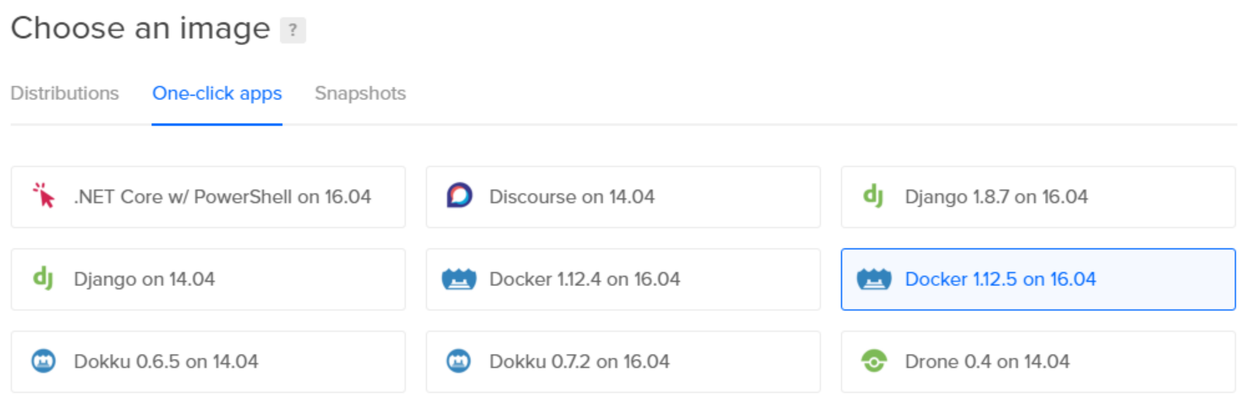
The minimal configuration known to mostly work is about half that, but you might find yourself in a tight place as soon as your usage goes up. If you're just tying out docker, a simple virtual machine might be good enough for you, if you're not running Linux on your laptop already.

[5]https://docs.docker.com/engine/installation/linux/

Cloud quick-start

If having your own hardware is a bit of a buzzkill, welcome to the world of the cloud. You can literally set up your own virtual server on Digital Ocean within minutes. You can use this DigitalOcean referral link[6] to get a $10 credit, while also helping me take some zeros of my hosting bills.

After signing up, creating a Linux instance with a running Docker engine is simple, and only takes a few clicks. There's this nice green button on the top header of the page, where it says "Create Droplet". Click it, and on the page it opens, navigate to "One-click apps" where you choose a "Docker" from the list.



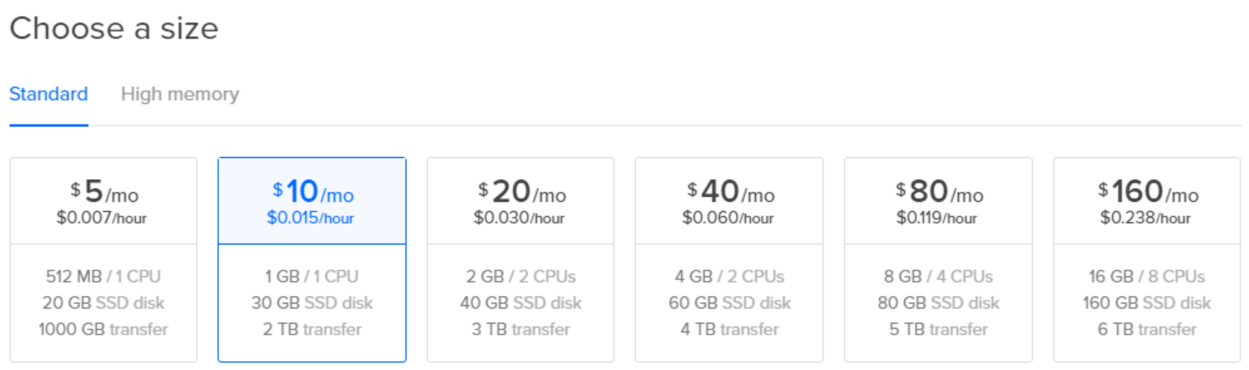
Choose an image ?

Distributions | One-click apps | Snapshots

.NET Core w/ PowerShell on 16.04	Discourse on 14.04	Django 1.8.7 on 16.04
Django on 14.04	Docker 1.12.4 on 16.04	Docker 1.12.5 on 16.04
Dokku 0.6.5 on 14.04	Dokku 0.7.2 on 16.04	Drone 0.4 on 14.04

Choose Docker from "One-click apps"

Running docker can be disk-usage intensive. Some docker images may "weigh" up to or more than 1 GB. I would definitely advise choosing an instance with *at least* 30GB of disk space, which is a bargain for $10 a month, but you will have to keep an eye out for disk usage. It's been known to fill up.



Choose a size

Standard | High memory

$5/mo	$10/mo	$20/mo	$40/mo	$80/mo	$160/mo
$0.007/hour	$0.015/hour	$0.030/hour	$0.060/hour	$0.119/hour	$0.238/hour
512 MB / 1 CPU	1 GB / 1 CPU	2 GB / 2 CPUs	4 GB / 2 CPUs	8 GB / 4 CPUs	16 GB / 8 CPUs
20 GB SSD disk	30 GB SSD disk	40 GB SSD disk	60 GB SSD disk	80 GB SSD disk	160 GB SSD disk
1000 GB transfer	2 TB transfer	3 TB transfer	4 TB transfer	5 TB transfer	6 TB transfer

Choose a reasonable disk size

Aside for some additional options on the page, like chosing a region where your droplet will be running in, there's only a big green "Create" button on the bottom of the page, which will set up everything you need.

[6]https://m.do.co/c/021b61109d56

Go: Introduction to Protobuf: Messages

In it's most basic ways, protocol buffers or protobufs are an approach to serialize structured data with minimal overhead. They require that the data structures be known and compatible between the client and server, unlike JSON where the structure itself is part of the encoding.

Protobufs and Go

The most basic protobuf message definition would look something like this:

```
message ListThreadRequest {
  // session info
  string sessionID = 1;

  // pagination
  uint32 pageNumber = 2;
  uint32 pageSize = 3;
}
```

The above message structure specifies the field names, types, and it's order in the encoded binary structure. Managing the structure has a few requirements that mean different things, if the structures are used as protobufs, or as JSON encoded data.

For example, this is the `protoc` generated code for this message:

```
type ListThreadRequest struct {
  // session info
  SessionID string `protobuf:"bytes,1,opt,name=sessionID,proto3" json:"sessionID,omi\
tempty"`
  // pagination
  PageNumber           uint32   `protobuf:"varint,2,opt,name=pageNumber,proto3" json\
:"pageNumber,omitempty"`
  PageSize             uint32   `protobuf:"varint,3,opt,name=pageSize,proto3" json:"\
pageSize,omitempty"`
  XXX_NoUnkeyedLiteral struct{} `json:"-"`
  XXX_unrecognized     []byte   `json:"-"`
  XXX_sizecache        int32    `json:"-"`
}
```

In the development process, managing the protobuf messages in a forward compatible way, there are some rules to follow:

- Adding new fields is not a breaking change,
- Removing fields is not a breaking change,
- Don't re-use a field number (breaks existing protobuf clients),
- Don't re-number fields (breaks existing protobuf clients),
- Renaming fields is not a breaking change for protobuf,
- Renaming fields is a breaking change for JSON,
- Changing field types is a breaking change for protobuf, mostly JSON as well,
- Numeric type changes e.g. uint16 to uint32 are generally safe for JSON,
- JSON and uint64 isn't fine if you need to use your APIs from javascript

So, basically, if you're relying on changing the protobuf message definitions during your development process, you'll need to keep the client(s) up to date with the server. This is especially important later, if you use the protobuf API with mobile clients (Android and iPhone apps), since making breaking changes to the API is going to hurt you there. Adding new fields or deleting old fields is the safest way to make changes to the API, as the protobufs definitions stay compatible.

Generating Protobuf Go code

In this series of articles, I'm going to build out a real world microservice, that tracks and aggregates view data for a number of services. It doesn't need authentication and is in it's definition a true microservice, as it will only need a single API endpoint.

We will create our `stats` microservice, by creating a `rpc/stats/stats.proto` file to begin with.

```
syntax = "proto3";

package stats;

option go_package = "github.com/titpetric/microservice/rpc/stats";

message PushRequest {
  string property = 1;
  uint32 section = 2;
  uint32 id = 3;
}

message PushResponse {}
```

Here a `proto3` version is declared. The important parts are the `go_package` option: with this an import path is defined for our service, which is useful if another service wants to import and use the message definitions here. Reusability is a protobuf built-in feature.

Since we don't want to do things half-way, we're going to approach our microservice with a CI-first approach. Using Drone CI is a great option for using a CI from the beginning, as it's drone/drone-cli[7] doesn't need a CI service set up, and you can just run the CI steps locally by running `drone exec`.

In order to set up out microservice build framework, we need:

1. Drone CI drone-cli installed
2. A docker environment with `protoc` and `protoc-gen-go` installed,
3. A `Makefile` to help us out for the long run
4. A `.drone.yml` config files with the build steps for generating go code,

Installing Drone CI

Installing drone-cli is very simple. You can run the following if you're on an amd64 linux host, otherwise just visit the drone/drone-cli[8] releases page and pull the version relevant for you and unpack it into `/usr/local/bin` or your common executable path.

```
cd /usr/local/bin
wget https://github.com/drone/drone-cli/releases/download/v1.2.0/drone_linux_amd64.t\
ar.gz
tar -zxvf drone*.tar.gz && rm drone*.tar.gz
```

Creating a build environment

Drone CI works by running CI steps you declare in `.drone.yml` in your provided Docker environment. For our build environment, I've created `docker/build/`, and inside a `Dockerfile` and a `Makefile` to assist with building and publishing the build image required for our case:

[7]https://github.com/drone/drone-cli
[8]https://github.com/drone/drone-cli

```
FROM golang:1.13

# install protobuf
ENV PB_VER 3.10.1
ENV PB_URL https://github.com/google/protobuf/releases/download/v${PB_VER}/protoc-${\
PB_VER}-linux-x86_64.zip

RUN apt-get -qq update && apt-get -qqy install curl git make unzip gettext rsync

RUN mkdir -p /tmp/protoc && \
    curl -L ${PB_URL} > /tmp/protoc/protoc.zip && \
    cd /tmp/protoc && \
    unzip protoc.zip && \
    cp /tmp/protoc/bin/protoc /usr/local/bin && \
    cp -R /tmp/protoc/include/* /usr/local/include && \
    chmod go+rx /usr/local/bin/protoc && \
    cd /tmp && \
    rm -r /tmp/protoc

# Get the source from GitHub
RUN go get -u google.golang.org/grpc

# Install protoc-gen-go
RUN go get -u github.com/golang/protobuf/protoc-gen-go
```

And the `Makefile`, implementing `make && make push` to quickly build and push our image to the docker registry. The image is published under `titpetric/microservice-build`, but I suggest you manage your own image here.

```
.PHONY: all docker push test

IMAGE := titpetric/microservice-build

all: docker

docker:
	docker build --rm -t $(IMAGE) .

push:
	docker push $(IMAGE)

test:
	docker run -it --rm $(IMAGE) sh
```

Creating a Makefile helper

It's very easy to run `drone exec`, but our requirements will grow over time and the Drone CI steps will become more complex and harder to manage. Using a Makefile enables us to add more complex targets which we will run from Drone with time. Currently we can start with a minimal Makefile which just wraps a call to `drone exec`:

```
.PHONY: all

all:
  drone exec
```

This very simple Makefile means that we'll be able to build our project with Drone CI at any time just by running `make`. We will extend it over time to support new requirements, but for now we'll just make sure it's available to us.

Creating a Drone CI config

With this, we can define our initial `.drone.yml` file that will build our Protobuf struct definitions, as well as perform some maintenance on our codebase:

```
workspace:
  base: /microservice

kind: pipeline
name: build

steps:
- name: test
  image: titpetric/microservice-build
  pull: always
  commands:
    - protoc --proto_path=$GOPATH/src:. -Irpc/stats --go_out=paths=source_relative:.\
 rpc/stats/stats.proto
    - go mod tidy > /dev/null 2>&1
    - go mod download > /dev/null 2>&1
    - go fmt ./... > /dev/null 2>&1
```

The housekeeping done is for our go.mod/go.sum files, as well as running `go fmt` on our codebase.

The first step defined under the `commands:` is our `protoc` command that will generate the Go definitions for our declared messages. In the folder where our `stats.proto` file lives, a `stats.pb.go` file will be created, with structures for each declared `message {}`.

Wrapping up

So, what we managed to achieve here:

- we created our CI build image with our `protoc` code generation environment,
- we are using Drone CI as our local build service, enabling us to migrate to a hosted CI in the future,
- we created a protobuf definition for our microservice message structures,
- we generated the appropriate Go code for encoding/decoding the protobuf messages

From here on out, we will move towards implementing a RPC service.

Go: Introduction to Protobuf: Services

The next step (or possibly the first step) about implementing a microservice, is defining it's API endpoints. What people usually do is write http handlers and resort to a routing framework like go-chi/chi[9]. But, protocol buffers can define your service, as RPC calls definitions are built into the protobuf schema.

Protobuf service definitions

Protobuf files may define a `service` object, which has definitions for one or many `rpc` calls. Let's extend our proto definitions for `stats.proto`, by including a service with a defined RPC.

```
service StatsService {
  rpc Push(PushRequest) returns (PushResponse);
}
```

A RPC method is defined with the `rpc` keyword. The service definition here declares a RPC named `Push`, which takes a message `PushRequest` as it's input, and returns the message `PushResponse` as it's output. All RPC definitions should follow the same naming pattern as a best practice. This way you can extend PushRequest and PushResponse, without introducing breaking changes to the RPC definition as well.

Updating the `.proto` file definitions in our project by default doesn't change anything. We need to generate a RPC scaffold using a RPC framework. For Go RPCs, we can consider gRPC from the beginning, or we can look towards a more simple Twitch RPC aka. Twirp[10].

Common reasons for choosing Twirp over gRPC are as follows:

- Twirp comes with HTTP 1.1 support (vendors need to catch up to HTTP/2 still),
- Twirp supports JSON transport out of the gate,
- gRPC re-implements HTTP/2 outside of net/http,

And reasons for gRPC over Twirp are:

- gRPC supports streaming (Twirp has an open proposal for streaming APIs[11])
- gRPC makes wire compatibility promises (this is built in to protobufs)

[9] https://github.com/go-chi/chi
[10] https://github.com/twitchtv/twirp
[11] https://github.com/twitchtv/twirp/issues/70

- More functionality on the networking level (retries, rpc discovery,…)

JSON would be the preferable format to demonstrate payloads, especially in terms of inspecting/sharing the payloads in documentation and similar. While gRPC is written by Google, there are many tools that you'd have to add in to make it a bit more developer friendly - gRPC-gateway[12] to add HTTP/1.1, and gRPCurl[13] to issue json-protobuf bridged requests.

gRPC is a much more rich RPC framework, supporting a wider array of use cases. If you feel that you need RPC streaming or API discovery, or your use cases lay beyond a simple request/response model, gRPC might be your only option.

Our microservice

Let's first start with Twitch RPC, so we can have a real comparison with gRPC in the next chapter.

About 10 years back I wrote a relatively simple microservice that is basically just tracking news item views. That solution is proving to be unmaintainable 10 years later at best, but still pretty good so it manages 0-1ms/request. It's also a bit smarter than that, since it tracks a number of assets that aren't news in the same way. So, effectively the service is tracking views in a multi-tennant way, for a predefined set of applications.

Let's refresh what our current service definition is:

```
service StatsService {
  rpc Push(PushRequest) returns (PushResponse);
}

message PushRequest {
  string property = 1;
  uint32 section = 2;
  uint32 id = 3;
}

message PushResponse {}
```

Our `StatsService` defines a RPC called `Push`, which takes a message with three parameters:

- property: the key name for a tracked property, e.g. "news"
- section: a related section ID for this property (numeric)
- id: the ID which defines the content being viewed (numeric)

The goal of the service is to log the data in PushRequest, and aggregate it over several time periods. The aggregation itself is needed to provide data sets like "Most read news over the last 6 months".

[12]https://github.com/gRPC-ecosystem/gRPC-gateway
[13]https://github.com/fullstorydev/gRPCurl

Twitch RPC scaffolding

The main client and server code generators for Twitch RPC are listed in the README for twitchtv/twirp[14]. The code generator we will use is available from `github.com/twitchtv/twirp/protoc-gen-twirp`. We will add this to our dockerfile:

```
--- a/docker/build/Dockerfile
+++ b/docker/build/Dockerfile
@@ -21,3 +21,6 @@ RUN go get -u google.golang.org/gRPC

 # Install protoc-gen-go
 RUN go get -u github.com/golang/protobuf/protoc-gen-go
+
+# Install protoc-gen-twirp
+RUN go get github.com/twitchtv/twirp/protoc-gen-twirp
```

And now we can extend our code generator in the `.drone.yml` file, by generating the twirp RPC output as well:

```
--- a/.drone.yml
+++ b/.drone.yml
@@ -10,6 +10,7 @@ steps:
   pull: always
   commands:
     - protoc --proto_path=$GOPATH/src:. -Irpc/stats --go_out=paths=source_relative:\
. rpc/stats/stats.proto
+    - protoc --proto_path=$GOPATH/src:. -Irpc/stats --twirp_out=paths=source_relati\
ve:. rpc/stats/stats.proto
```

We run the `protoc` command twice, but the `--twirp_out` option could actually be added to the existing command. We will keep this seperate just to help with readability, so we know which command is responsible to generate what. When it comes to the code generator plugins for protoc, there's a long list of plugins that can generate anything from JavaScript clients to Swagger documentation. As we will add these, we don't want the specific for generating one type of output to bleed into other generator rules.

The above command will generate a `stats.twirp.go` file in the same folder as `stats.proto` file. The important part for our implementation is the following interface:

[14] https://github.com/twitchtv/twirp

```go
type StatsService interface {
	Push(context.Context, *PushRequest) (*PushResponse, error)
}
```

In order to implement our Twitch RPC service, we need an implementation for this interface. For that, we will look at our own code generation that will help us with this. Particularly, we want to scaffold both the server and the client code that could get updated if our service definitions change.

Up next: we'll generate a gRPC server with the same proto definition, and look at the implementation changes this brings for us. We will try to attempt to answer which RPC framework will serve us better at the long run, with more concrete examples of the differences between the two.