

---

# 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



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2020 Tit Petric

# Contents

<b>Introduction</b>	<b>1</b>
About me	1
Who is this book for?	1
How should I study it?	3
<b>Requirements</b>	<b>4</b>
Linux and Docker	4
<b>Go: Introduction to Protobuf: Messages</b>	<b>6</b>
Protobufs and Go	6
Generating Protobuf Go code	7
Wrapping up	11
<b>Go: Introduction to Protobuf: Services</b>	<b>12</b>
Protobuf service definitions	12
Our microservice	13
Twitch RPC scaffolding	14
<b>Go: Introduction to Protobuf: gRPC</b>	<b>16</b>
Generating the gRPC scaffolding	16
Comparing the server side	16
Comparing the client side	17
<b>Make: Dynamic Makefile targets</b>	<b>19</b>
Building multiple targets with Go	19
Dynamic Makefile target for building RPC/Protobuf files	19
Building our services from cmd/	21
<b>Bash: Poor mans code generation</b>	<b>23</b>
Generating our service cmd/ folders	23
Generating the client for our service	25
Server implementation	26
<b>Bash: Embedding files into Go</b>	<b>29</b>
Planning database migrations	29
Poor mans file embedding	30

## CONTENTS

<b>Go: Scaffolding database migrations</b> . . . . .	<b>33</b>
The migration API . . . . .	35
<b>Drone CI: Testing database migrations</b> . . . . .	<b>40</b>
Configuring Drone CI for database integration tests . . . . .	42
<b>Go: Database first struct generation</b> . . . . .	<b>46</b>
Database schema data structures and helpers . . . . .	46
Querying information schema . . . . .	47
Type conversions . . . . .	49
Moving forward . . . . .	55
<b>Go: Generating database schema documentation</b> . . . . .	<b>56</b>
Updating the Go generators . . . . .	56
Implementing a markdown output renderer . . . . .	58
A caveat emptor about database migrations . . . . .	63
<b>Go: Dependency injection with Wire</b> . . . . .	<b>65</b>
The Go way . . . . .	65
Runtime vs. compile time . . . . .	65
Wire and dependency providers . . . . .	66
Wire and dependency Injector . . . . .	69
<b>Docker: Building images with security in mind</b> . . . . .	<b>73</b>
Configuring Makefile targets . . . . .	73
The basic Dockerfile image . . . . .	74
Security implications of our docker image . . . . .	75
Possible improvements . . . . .	77
<b>Go: implementing a microservice</b> . . . . .	<b>79</b>
The database schema . . . . .	79
Improving the microservice environment . . . . .	80
Implementing Push . . . . .	82
<b>Docker/Bash: Issuing requests against our microservice</b> . . . . .	<b>87</b>
The docker-compose file . . . . .	87
Our first request . . . . .	88
<b>Go: Improving our database handling</b> . . . . .	<b>93</b>
Data source name - DSN . . . . .	93
Multiple database connections . . . . .	94
Updating database connections . . . . .	95
Database connection retry . . . . .	97
Migration improvements . . . . .	99
Re-testing the bundled migrations . . . . .	104

## CONTENTS

<b>Go: Instrumenting the HTTP service with Elastic APM</b>	<b>105</b>
Wrapping our existing handler	105
Logging errors too	106
Setting up ELK	107
Setting up APM	108
Configuring our sender	109
Reviewing ELK data	110
<b>Go: Instrumenting the Database client with Elastic APM</b>	<b>114</b>
Extending DB connection	114
Verifying it's working	115
<b>Go: Stress testing our service</b>	<b>118</b>
Setting up our stress test	118
Resolving detected errors	120
Removing context cancellation	123
Wrapping up	127
<b>Go: Background jobs</b>	<b>128</b>
Adding background jobs to our service	128
Do we really need Init?	130
The background job	131
Housekeeping	133
Verifying everything works	134
<b>Go: Optimizing requests with a queue</b>	<b>136</b>
The Queue	136
The Producer	138
The Consumer	140
Benchmarking	141
Notes and pitfalls	142

# 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](https://leanpub.com/api-foundations)<sup>1</sup>,
- Author of [12 Factor Applications with Docker and Go](https://leanpub.com/12fa-docker-golang)<sup>2</sup>,
- Blog author on [scene-si.org](https://scene-si.org)<sup>3</sup>

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](https://twitter.com/titpetric)<sup>4</sup> 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.

---

<sup>1</sup><https://leanpub.com/api-foundations>

<sup>2</sup><https://leanpub.com/12fa-docker-golang>

<sup>3</sup><https://scene-si.org>

<sup>4</sup><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](https://docs.docker.com/engine/installation/linux/)<sup>5</sup> 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 trying out docker, a simple virtual machine might be good enough for you, if you're not running Linux on your laptop already.

---

<sup>5</sup><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](#)<sup>6</sup> 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	 <b>Docker 1.12.5 on 16.04</b>
 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

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

### Choose a reasonable disk size

Aside for some additional options on the page, like choosing 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.

<sup>6</sup><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:

```
1 message ListThreadRequest {
2     // session info
3     string sessionID = 1;
4
5     // pagination
6     uint32 pageNumber = 2;
7     uint32 pageSize = 3;
8 }
```

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:

```
1 type ListThreadRequest struct {
2     // session info
3     SessionID string `protobuf:"bytes,1,opt,name=sessionID,proto3" json:"sessionID,omitempty"`
4
5     // pagination
6     PageNumber      uint32 `protobuf:"varint,2,opt,name=pageNumber,proto3" json:"\
7 :\"pageNumber,omitempty"`
8     PageSize        uint32 `protobuf:"varint,3,opt,name=pageSize,proto3" json:"\
9 pageSize,omitempty"`
10    XXX_NoUnkeyedLiteral struct{} `json:"- "`
11    XXX_unrecognized    []byte `json:"- "`
12    XXX_sizecache        int32  `json:"- "`
13 }
```

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 its 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.

```
1  syntax = "proto3";
2
3  package stats;
4
5  option go_package = "github.com/titpetric/microservice/rpc/stats";
6
7  message PushRequest {
8      string property = 1;
9      uint32 section = 2;
10     uint32 id = 3;
11 }
12
13 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](https://github.com/drone/drone-cli)<sup>7</sup> 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 our 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](https://github.com/drone/drone-cli/releases)<sup>8</sup> releases page and pull the version relevant for you and unpack it into `/usr/local/bin` or your common executable path.

```
1 cd /usr/local/bin
2 wget https://github.com/drone/drone-cli/releases/download/v1.2.0/drone_linux_amd64.t\
3 ar.gz
4 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:

---

<sup>7</sup><https://github.com/drone/drone-cli>

<sup>8</sup><https://github.com/drone/drone-cli>

```

1  FROM golang:1.13
2
3  # install protobuf
4  ENV PB_VER 3.10.1
5  ENV PB_URL https://github.com/google/protobuf/releases/download/v${PB_VER}/protoc-${\
6  PB_VER}-linux-x86_64.zip
7
8  RUN apt-get -qq update && apt-get -qqy install curl git make unzip gettext rsync
9
10 RUN mkdir -p /tmp/protoc && \
11     curl -L ${PB_URL} > /tmp/protoc/protoc.zip && \
12     cd /tmp/protoc && \
13     unzip protoc.zip && \
14     cp /tmp/protoc/bin/protoc /usr/local/bin && \
15     cp -R /tmp/protoc/include/* /usr/local/include && \
16     chmod go+rx /usr/local/bin/protoc && \
17     cd /tmp && \
18     rm -r /tmp/protoc
19
20 # Get the source from GitHub
21 RUN go get -u google.golang.org/grpc
22
23 # Install protoc-gen-go
24 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.

```

1  .PHONY: all docker push test
2
3  IMAGE := titpetric/microservice-build
4
5  all: docker
6
7  docker:
8      docker build --rm -t $(IMAGE) .
9
10 push:
11     docker push $(IMAGE)
12
13 test:
14     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`:

```
1 .PHONY: all
2
3 all:
4     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:

```
1 workspace:
2     base: /microservice
3
4 kind: pipeline
5 name: build
6
7 steps:
8 - name: test
9     image: titpetric/microservice-build
10    pull: always
11    commands:
12        - protoc --proto_path=$GOPATH/src:. -Irpc/stats --go_out=paths=source_relative:. \
13          rpc/stats/stats.proto
14        - go mod tidy > /dev/null 2>&1
15        - go mod download > /dev/null 2>&1
16        - 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](https://github.com/go-chi/chi)<sup>9</sup>. 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.

```
1 service StatsService {  
2     rpc Push(PushRequest) returns (PushResponse);  
3 }
```

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](#)<sup>10</sup>.

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](#)<sup>11</sup>)
- gRPC makes wire compatibility promises (this is built in to protobufs)

---

<sup>9</sup><https://github.com/go-chi/chi>

<sup>10</sup><https://github.com/twitchtv/twirp>

<sup>11</sup><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](https://github.com/gRPC-ecosystem/gRPC-gateway)<sup>12</sup> to add HTTP/1.1, and [gRPCurl](https://github.com/fullstorydev/gRPCurl)<sup>13</sup> 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-tenant way, for a predefined set of applications.

Let's refresh what our current service definition is:

```

1  service StatsService {
2      rpc Push(PushRequest) returns (PushResponse);
3  }
4
5  message PushRequest {
6      string property = 1;
7      uint32 section = 2;
8      uint32 id = 3;
9  }
10
11 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".

<sup>12</sup><https://github.com/gRPC-ecosystem/gRPC-gateway>

<sup>13</sup><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](https://github.com/twitchtv/twirp)<sup>14</sup>. The code generator we will use is available from [github.com/twitchtv/twirp/protoc-gen-twirp](https://github.com/twitchtv/twirp/protoc-gen-twirp). We will add this to our dockerfile:

```

1  --- a/docker/build/Dockerfile
2  +++ b/docker/build/Dockerfile
3  @@ -21,3 +21,6 @@ RUN go get -u google.golang.org/gRPC
4
5  # Install protoc-gen-go
6  RUN go get -u github.com/golang/protobuf/protoc-gen-go
7  +
8  +# Install protoc-gen-twirp
9  +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:

```

1  --- a/.drone.yml
2  +++ b/.drone.yml
3  @@ -10,6 +10,7 @@ steps:
4      pull: always
5      commands:
6          - protoc --proto_path=$GOPATH/src:. -Irpc/stats --go_out=paths=source_relative:\
7      . rpc/stats/stats.proto
8  +   - protoc --proto_path=$GOPATH/src:. -Irpc/stats --twirp_out=paths=source_relati\
9  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 separate 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:

---

<sup>14</sup><https://github.com/twitchtv/twirp>

```
1  type StatsService interface {  
2      Push(context.Context, *PushRequest) (*PushResponse, error)  
3  }
```

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.

# Go: Introduction to Protobuf: gRPC

We implemented a Twitch RPC generator in the previous chapter, but we do have some hard core gRPC fans that requested that I should check out what gRPC produces. I am interested into comparing these two as well, so let's start by scaffolding our gRPC handlers. Our main goal is to compare the implementation details of gRPC over Twitch RPC in more concrete terms.

## Generating the gRPC scaffolding

We need to modify the `Makefile` slightly, in order to include the `grpc` codegen plugin, by changing a `protoc` option:

Under the `rpc.%` target, change the `go_out` option, from:

```
1 --go_out=paths=source_relative:.
```

to the following:

```
1 --go_out=plugins=grpc,paths=source_relative:.
```

This will include the `grpc` plugin that generates the code for the gRPC Client and Server. The first thing we can notice is that there is a significant change in our `go.sum` file, namely that 42 different package versions have been listed as dependencies. The only package that really stands out however is `prometheus/client_model`. This might mean that the gRPC server implementation internals support some custom prometheus metrics out of the box. We definitely don't get that from Twitch RPC, but we are planning to add on instrumentation.

## Comparing the server side

Inspecting the changed `*.pb.go` files, we can compare the interface produced by Twitch RPC, and what gRPC produces. The gRPC `protoc` generator produces two distinct interfaces, `StatsServiceClient` and `StatsServiceServer`. As we are interested in the first one, let's compare it now:

```

1 // StatsServiceServer is the server API for StatsService service.
2 type StatsServiceServer interface {
3     Push(context.Context, *PushRequest) (*PushResponse, error)
4 }

```

Compared to Twitch RPC:

```

1 type StatsService interface {
2     Push(context.Context, *PushRequest) (*PushResponse, error)
3 }

```

So, first, we see that the implementation for our Twitch RPC service is compatible with our gRPC server. This means that our workflow will not change a single bit, if we decide to migrate from Twitch to gRPC.

We can run a single service, which exposes both gRPC and Twirp endpoints. Maintaining them both seems like a bad idea, but as we don't have any Twirp specific implementation in our service itself, it seems like we can manage to run both endpoints without difficulty.

## Comparing the client side

The difference seems to be in the client itself:

```

1 type StatsServiceClient interface {
2     Push(ctx context.Context, in *PushRequest, opts ...grpc.CallOption) (*PushResponse\
3     , error)
4 }

```

While the Twirp Client and Server conform to a single interface, gRPC clients have additional options available. A cursory reading of [grpc.CallOption](https://godoc.org/google.golang.org/grpc#CallOption)<sup>15</sup> gives us a list of possible constructors. As is supposedly common with Google APIs, of the listed options currently:

- CallContentSubtype (string) can be set to use JSON encoding on the wire,
- CallCustomCodec - DEPRECATED (use ForceCodec)
- FailFast - DEPRECATED (use WaitForReady)
- ForceCodec - EXPERIMENTAL API (wait, we just came here from CallCustomCodec)
- Header - retrieves header metadata
- MaxCall(Recv/Send)MsgSize - client message size limits
- MaxRetryRPCBufferSize - EXPERIMENTAL
- Peer (p \*peer.Peer) - Populate \*p after RPC completes

<sup>15</sup><https://godoc.org/google.golang.org/grpc#CallOption>

- PerRPCCredentials - Sets credentials for a RPC call
- Trailer (md \*metadata.MD) - returns trailer metadata (no idea what is a Trailer)
- UseCompressor - EXPERIMENTAL
- WaitForReady (waitForReady bool) - if false, fail immediately, if true block/retry (default=false)

So, to summarize - out of all those options, 3 are EXPERIMENTAL, 2 are DEPRECATED and one of them is pointing to an EXPERIMENTAL option, and the biggest question raised is how the gRPC client behaves in relation to WaitForReady, and which option is recommended if any.

In comparison, Twitch RPC produces, at least in my opinion, a nicer interface for specific clients:

```

1 // NewStatsServiceProtobufClient creates a Protobuf client that implements the Stats\
2 Service interface.
3 func NewStatsServiceProtobufClient(addr string, client HTTPClient) StatsService {
4     ...
5
6 // NewStatsServiceJSONClient creates a JSON client that implements the StatsService \
7 interface.
8 func NewStatsServiceJSONClient(addr string, client HTTPClient) StatsService {
9     ...

```

gRPC could benefit from some interface deduplication here. The gRPC clients constructor could take those particular call options so both the client and server could conform to the same interface. Also, much as the server side implementation, it's also obvious that gRPC implements the client side as well, as the transport isn't based on net/http:

```

1 // NewStatsServiceClient creates a gRPC client that implements the StatsServiceClien\
2 t interface.
3 func NewStatsServiceClient(cc *grpc.ClientConn) StatsServiceClient {
4     ...

```

What we can also see is that the gRPC client can authenticate to the gRPC server via the PerRPCCredentials option. This is also something that Twitch RPC doesn't provide for us. We don't need it for our service, but it's something to consider if you want to increase the level of security inside your service mesh.

We won't create the gRPC server just yet, but we'll keep this codegen option enabled for the future. As we already discussed, gRPC is a great framework to have when we have clients that can speak it natively. It's not great for the browser or the javascript console (am I wrong?), and it's definitely not great for debugging/sniffing traffic, but using the proto files to generate the clients for Android/iPhone apps is a valid use case. gRPC has wider code generation support than Twirp and is a better option if you want to cover a larger ecosystem.

# Make: Dynamic Makefile targets

A common pattern which we want to support when building our services is to build everything under `rpc/{service}/{service}.proto` as well as `cmd/*` and other locations which may be manually written or generated with various tools.

## Building multiple targets with Go

Go does actually support building the whole `cmd/` folder in one command:

```
1 go build -o ./build ./cmd/...
```

This functionality was added in Go [just recently, in the 1.13.4 tag<sup>16</sup>](#).

It will not create the build folder itself, but will build all your applications under based on the `./cmd/...` argument. Why do we need dynamic makefile targets? We can live without them for building multiple Go programs, but we don't only need them for building the `cmd/` folder, but for other code generation folders as well.

We could resort to `//go:generate` tags to invoke our files, but it would again mean some duplication/replication of responsibility. Defining a single dynamic Makefile target keeps all the logic in one place. It also allows you to customize the build steps to target multiple architectures, operating systems, and other use cases.

Let's learn how dynamic Makefile targets work and try to cover our use cases.

## Dynamic Makefile target for building RPC/Protobuf files

The first target that we want to support is running all code generation required for our RPC services. We define a `rpc` makefile target, and use dynamic execution to build a list of dynamic targets:

```
1 rpc: $(shell ls -d rpc/* | sed -e 's/\/\\.\\.\\.')
```

Here we use the make built-in feature to run a shell, and with the output of that shell command define our dynamic targets. For each folder found under `rpc/`, a target like `rpc.{folder}` is created. For example, if we wanted to build the code for a target, we could do:

---

<sup>16</sup><https://github.com/golang/go/commit/b48bda9c6f57ca9a940eac95700485b9640a62e9>



```

1  rpc.stats:
2      protoc --proto_path=$GOPATH/src:. -Irpc/stats --go_out=paths=source_relative:. rpc\
3  /stats/stats.proto
4      protoc --proto_path=$GOPATH/src:. -Irpc/stats --twirp_out=paths=source_relative:. \
5  rpc/stats/stats.proto

```

If we used `rpc/stats` as the target name, `make` wouldn't do anything as that file/folder already exists. This is why we are rewriting the target name to include a `.` (`rpc.stats`). The obvious second part of our requirement is to make this target dynamic. We want to build any number of services.

```

1  --- a/Makefile
2  +++ b/Makefile
3  @@ -1,4 +1,10 @@
4  -.PHONY: all
5  +.PHONY: all rpc
6
7  all:
8      drone exec
9  +
10 +rpc: $(shell ls -d rpc/* | sed -e 's/\///.g')
11 +rpc.%: SERVICE=$*
12 +rpc.%:
13 +  @echo '> protoc gen for $(SERVICE)'
14 +  @protoc --proto_path=$(GOPATH)/src:. -Irpc/$(SERVICE) --go_out=paths=source_\
15 relative:. rpc/$(SERVICE)/$(SERVICE).proto
16 +  @protoc --proto_path=$(GOPATH)/src:. -Irpc/$(SERVICE) --twirp_out=paths=sour\
17 ce_relative:. rpc/$(SERVICE)/$(SERVICE).proto

```

The `%` target takes any option (which doesn't include `/` in the pattern). We can also declare variables like `SERVICE` for each target. The variable can also be exported into the environment by prefixing it with `export` (example: `rpc.%: export SERVICE=$*`). Currently we don't need this, but we will need it later on to pass build flags for our `cmd/` programs.

The `$*` placeholder is the matched target parameter. As the target is `rpc.stats`, the variable here will only contain `stats` but not `rpc.` since it's part of the target definition.

With the `@` prefix on individual commands in the target, we suppress the output of the command. All we need to do is update the step in `.drone.yml` into `make rpc`, and we have support for a dynamic number of services. Running `make` verifies this:

```

1 # make
2 [test:0] + make rpc
3 [test:1] > protoc gen for stats
4 [test:2] + go mod tidy > /dev/null 2>&1
5 [test:3] + go mod download > /dev/null 2>&1
6 [test:4] + go fmt ./... > /dev/null 2>&1

```

## Building our services from cmd/

For each service, we will create a `cmd/{service}/*.go` structure, containing at least `main.go`. Let's start with adding a simple `cmd/stats/main.go` with a hello world to greet us. We will come back and scaffold the actual service later.

```

1 package main
2
3 void main() {
4     println("Hello world")
5 }

```

The function `println` is a Go built-in function that works without importing any package. It shouldn't really be used, but as far as providing some test output, it's the shortest way to do that. We will throw this program away, so don't pay it much attention.

Using what we learned for dynamic targets with the `rpc` target, let's create a `build` target which will build all the applications we will put under `cmd/` by running `make build`.

```

1 build: export GOOS = linux
2 build: export GOARCH = amd64
3 build: export CGO_ENABLED = 0
4 build: $(shell ls -d cmd/* | sed -e 's/cmd\\//build./')
5     echo OK.
6
7 build.%: SERVICE=$*
8 build.%:
9     go build -o build/$(SERVICE)-$(GOOS)-$(GOARCH) ./cmd/$(SERVICE)/*.go

```

For the main `build` target, we define our build environment variables - we want to build our services for linux, for amd64 architecture, and we want to disable CGO so we have static binaries. We list

all cmd locations as dynamic targets and remap them to `build.%`, similarly to what we do with the `rpc` target. All that is left to do is to add `make build` at the end of `.drone.yml`, and add our `/build` folder into the `.gitignore` file for our project.

# Bash: Poor mans code generation

When it comes to generating code, I often find that the simplest code generation only needs variable replacement. There's a cli utility that already facilitates this with environment variables, named [envsubst](https://www.gnu.org/software/gettext/manual/html_node/envsubst-Invocation.html)<sup>17</sup>, part of the gettext suite of tools.

In it's most simple form, it takes a template file and replaces any variables contained within, and outputs the resulting text. It's not super great if you have any dangling \$ characters in your template, but for most purposes it's going to work well.

## Generating our service cmd/ folders

The first variable that we will need in our generated files, is the full import name that our project lives at. We can get this name from the go.mod file in the root of the project. Let's start with a dynamic rule for make templates and go from there.

```
1 templates: $(shell ls -d rpc/* | sed -e 's/rpc\\//templates./g')
2   @echo OK.
3
4 templates.%. SERVICE=$*
5 templates.%. export MODULE=$(shell grep ^module go.mod | sed -e 's/module //g')
6 templates.%.
7   @echo templates: $(SERVICE) $(MODULE)
```

For each of our RPC services, a templates.{service} target is created, and for those, the SERVICE and MODULE environment variable are exported. When we're going to be using envsubst, the environment variables will be available for replacement.

Let's create our basic template for main.go files (save as templates/cmd\_main.go.tpl):

---

<sup>17</sup>[https://www.gnu.org/software/gettext/manual/html\\_node/envsubst-Invocation.html](https://www.gnu.org/software/gettext/manual/html_node/envsubst-Invocation.html)

```

1  package main
2
3  // file is autogenerated, do not modify here, see
4  // generator and template: templates/cmd_main.go.tpl
5
6  import (
7      "log"
8      "context"
9
10     "net/http"
11
12     _ "github.com/go-sql-driver/mysql"
13
14     "${MODULE}/rpc/${SERVICE}"
15     server "${MODULE}/server/${SERVICE}"
16 )
17
18 func main() {
19     ctx := context.TODO()
20
21     srv, err := server.New(ctx)
22     if err != nil {
23         log.Fatalf("Error in service.New(): %v", err)
24     }
25
26     twirpHandler := ${SERVICE}.New${SERVICE_CAMEL}ServiceServer(srv, nil)
27
28     http.ListenAndServe(":3000", twirpHandler)
29 }

```

As you can see, there is really only one issue with the replacements here, and that is the need to have a camel-cased name of the service we are writing. It is quite easy to provide it:

```

1  templates.%.%: export SERVICE_CAMEL=$(shell echo ${SERVICE} | sed -r 's/^(^|_)([a-z])/\\
2  U\2/g')

```

The regular expression replacement is looking for two matches:

- either the beginning of the string (^) or an underscore (\_)
- the character directly following the first match

And then replaces it with the uppercase version of this character. So stats becomes Stats, and hello\_world would become HelloWorld, and so on. Let's use this template to generate cmd/stats/main.go, and verify the output:

```

1 templates.%:
2   mkdir -p cmd/${SERVICE}
3   envsubst < templates/cmd_main.go.tpl > cmd/${SERVICE}/main.go

```

Looking at the main.go output produced, we can verify that everything works as expected:

```

1   twirpHandler := stats.NewStatsServiceServer(srv, nil)

```

## Generating the client for our service

Another simple template which we can generate is the Twirp client for our microservice.

```

1 package ${SERVICE}
2
3 // file is autogenerated, do not modify here, see
4 // generator and template: templates/client_client.go.tpl
5
6 import (
7     "net/http"
8
9     "${MODULE}/rpc/${SERVICE}"
10 )
11
12 func New() ${SERVICE}.${SERVICE_CAMEL}Service {
13     return NewCustom("http://${SERVICE}.service:3000", http.Client{})
14 }
15
16 func NewCustom(addr string, client ${SERVICE}.HTTPClient) ${SERVICE}.${SERVICE_CAMEL}
17 }Service {
18     return ${SERVICE}.New${SERVICE_CAMEL}ServiceJSONClient(addr, client)
19 }

```

No additional variables are required, so we can just add an additional envsubst line to the makefile:

```

1 templates.%:
2   mkdir -p cmd/${SERVICE} client/${SERVICE}
3   envsubst < templates/cmd_main.go.tpl > cmd/${SERVICE}/main.go
4   envsubst < templates/client_client.go.tpl > client/${SERVICE}/client.go

```

## Server implementation

From here on out we need to see what we can do to build the implementation of the server side of our service. The templating here will serve only as an initial stubbed version of a service implementation, as we will need to extend it with complexity of our own implementation.

Let's start by adding an interface scaffolding utility to our build environment Dockerfile:

```
1 # Install interface scaffolder
2 RUN go get -u github.com/josharian/impl
```

Using [josharian/impl](https://github.com/josharian/impl)<sup>18</sup>, we can generate a scaffold of the interface which we need to satisfy. We can now create the server side template like this:

```
1 package ${SERVICE}
2
3 import (
4     "context"
5
6     "${MODULE}/rpc/${SERVICE}"
7 )
8
9 type Server struct {
10 }
11
12 func New(ctx context.Context) (*Server, error) {
13     return &Server{}, nil
14 }
15
16 var _ ${SERVICE}.${SERVICE_CAMEL}Service = &Server{}
```

And then we can just append the output of `impl` after it, so the generated code immediately satisfies the service interface. We need to update the makefile target now, to invoke this template, as well as generate the function stubs for the `StatsService` interface.

---

<sup>18</sup><https://github.com/josharian/impl>

```

1 templates.%:
2   mkdir -p cmd/$(SERVICE) client/$(SERVICE) server/$(SERVICE)
3   envsubst < templates/cmd_main.go.tpl > cmd/$(SERVICE)/main.go
4   envsubst < templates/client_client.go.tpl > client/$(SERVICE)/client.go
5   envsubst < templates/server_server.go.tpl > server/$(SERVICE)/server.go
6   impl -dir rpc/$(SERVICE) 'svc *Server' $(SERVICE).$(SERVICE_CAMEL)Service >> serve\
7   r/$(SERVICE)/server.go

```

After checking that the stubs are generated, running `make` will successfully build our project. Looking at our service, we are left to implement the function defined in the server interface:

```

1 func (svc *Server) Push(_ context.Context, _ *stats.PushRequest) (*stats.PushRespon\
2 e, error) {
3   panic("not implemented") // TODO: Implement
4 }

```

We also want to generate our `server.go` file only if it doesn't exist - we only want to scaffold it for the first time, and checking file existence in Makefile is difficult, so let's move this into a bash script, and adjust the verbosity/output of the complete templates target:

```

1 templates.%:
2   @mkdir -p cmd/$(SERVICE) client/$(SERVICE) server/$(SERVICE)
3   @envsubst < templates/cmd_main.go.tpl > cmd/$(SERVICE)/main.go
4   @echo "~ cmd/$(SERVICE)/main.go"
5   @envsubst < templates/client_client.go.tpl > client/$(SERVICE)/client.go
6   @echo "~ client/$(SERVICE)/client.go"
7   @./templates/server_server.go.sh

```

The `@` command suppresses the output of the command being run, and we output `~ file.go` for each file being written. Our `server_server.go.sh` has checks for the existence of our `server.go` file:

```

1 #!/bin/bash
2 cd $(dirname $(dirname $(readlink -f $0)))
3
4 if [ -z "${SERVICE}" ]; then
5   echo "Usage: SERVICE=[name] SERVICE_CAMEL=[Name] $0"
6   exit 255
7 fi
8
9 OUTPUT="server/${SERVICE}/server.go"
10
11 # only generate server.go if it doesn't exist

```



```
12 if [ ! -f "$OUTPUT" ]; then
13     envsubst < templates/server_server.go.tpl > $OUTPUT
14     impl -dir rpc/${SERVICE} 'svc *Server' ${SERVICE}.${SERVICE_CAMEL}Service >> $OUTP\
15 UT
16     echo "~ $OUTPUT"
17 fi
```

This should give us a good start for our scaffolding. The correct and most optimal way to plan your services would be to take your time planning the proto schema, and then let the scaffolding take you to a place where you only need to implement the defined RPCs.

With more fluid and changing requirements, you will not really take advantage of the code generation when you'll be updating the proto file during development. Adding new RPC calls will mean you'll have to define the new functions by hand, which isn't really that bad with ~3 SLOC, but still isn't ideal for anything other than a microservice with only a few function calls. Planning saves time.

# Bash: Embedding files into Go

There's always that chance that your Go app/service will need to bundle some files in order to fulfill some requirement. A common case of that is bundling SQL migrations into an app. SQL migrations are a set of SQL queries that need to run to set up a clean database, or to change database schema over the course of the applications lifetime.

## Planning database migrations

Database migrations can be a tricky thing. With common systems, people tend to write database migrations that go both ways - up, or down (undo). Since there's always a possibility of data loss, we will only plan migrations that go one way.

Let's implement the following filesystem schema:

```
1 /db - the main database migration package (generated .go files too),
2 /db/schema/ - a collection of service migrations
3 /db/schema/stats/ - migrations for `stats` schema, *.up.sql files
4 /db/schema/.../ - other service migrations...
```

The individual migrations should be consistently named, for example, a database migration might be stored in the file 2019-11-18-141536-create-message.up.sql. Particularly, we only consider \*.up.sql to be a migration, and the prefix with the full date and time serves as a sorting key, so we know in which order the migrations should execute in the database.

In addition to the actual migrations, we need to track the migrations as they have been applied. A migration that was already applied shouldn't run again as it will produce errors, or worse. For that we need a migration table, which should be part of every service schema.

```
1 CREATE TABLE IF NOT EXISTS `migrations` (
2   `project` varchar(16) NOT NULL COMMENT 'Microservice or project name',
3   `filename` varchar(255) NOT NULL COMMENT 'yyyy-mm-dd-HHMMSS.sql',
4   `statement_index` int(11) NOT NULL COMMENT 'Statement number from SQL file',
5   `status` text NOT NULL COMMENT 'ok or full error message',
6   PRIMARY KEY (`project`, `filename`)
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

We want to embed these files into our app, and at the same time, keep them grouped by service, so only individual migrations for a service can be executed. For this purpose, we create a filesystem wrapper in db/fs.go:

```
1 type FS map[string]string
```

Well, that was anti-climactic. If you expected some over-engineering here, it really isn't the point - we're going for simplicity here. The filesystem takes the filename as the key, and the file contents as the value. It's very easy to generate this filesystem, as effectively each filesystem is only a few lines of code:

```
1 package db;
2
3 var Stats FS = FS{
4     "2019-11-18-141536-create-message.up.sql": "...",
5     "...": "...",
6 }
```

In fact, the filesystems are so simple, that we can again resort to simple code generation to implement them. We don't need any kind of compression, and the only issue we need to solve, is properly quoting or escaping the file contents, so we can know that the generated code won't produce errors.

## Poor mans file embedding

Ah, as we analyze how we're going to generate the filesystem assets, we also realize that the poor mans templating from the previous section isn't really appropriate anymore. We need to have a way to loop over the list of migration files, in order to generate individual filesystems.

We still might not want the complexity of a Go application to embed these files, so let's resort to using bash to generate the files. The requirements are simple - for each migration for a schema generate a filesystem, and generate an index of `map[string]FS`, with a key/value for each service.

Since the files contain special characters like newlines and back-ticks and aren't nicely embeddable in Go as is, we will resort to base64 encoding for the file contents. For that we can use the shell `base64` command, which supports encoding and decoding.

```
1 #!/bin/bash
2 cd $(dirname $(dirname $(readlink -f $0)))
3
4 ## encode file contents in base64
5 function base64_encode {
6     cat $1 | base64 -w 0
7 }
8
9 ## generate a service FS
10 function render_service_schema {
```

```

11  local schema=$(basename $1)
12  echo "package db;"
13  echo
14  echo "var $schema FS = FS{"
15  local files=$(find $1 -name '*.sql' | sort)
16  for file in $files; do
17      echo "\"$(basename $file)\": \"$(base64_encode $file)\", "
18  done
19  echo "}"
20 }
21
22 ## list all service FS into `migrations` global
23 function render_schema {
24     echo "package db;"
25     echo
26     echo "var migrations map[string]FS = map[string]FS{"
27     for schema in $schemas; do
28         local package=$(basename $schema)
29         echo "\"${package}\": ${package},"
30     done
31     echo "}"
32 }
33
34 ## list all service migrations (db/schema/stats, ...)
35 schemas=$(ls db/schema/*/migrations.sql | xargs -n1 dirname)
36 for schema in $schemas; do
37     # db/schema/stats -> schema/stats
38     schema_relative=${schema/db//}
39     # schema/stats -> db/schema_stats.go
40     output="db/${schema_relative}/../.go"
41
42     render_service_schema $schema > $output
43 done
44
45 render_schema > db/schema.go

```

All that's left to do here is just to run `go fmt` on the resulting go files. As that is already handled in our Drone CI steps, we have now successfully prepared the required SQL migrations into the `db` package so we can use it from here.

As we have now embedded all the database migrations into go code, we can move on towards running these migrations on a real database as part of our CI testing suite.

It's worth noting that just days before publishing this chapter, a proposal for embedding landed on golang/go, by [@bradfitz](https://twitter.com/bradfitz)<sup>a</sup>. It seems if all goes well with the planning here, and the proposal isn't rejected outright for usability concerns, that some time in the future the Go toolchain might handle embedding files in a portable and secure way. Take a read here: [proposal: cmd/go: support embedding static assets \(files\) in binaries #35950](https://github.com/golang/go/issues/35950)<sup>b</sup>

---

<sup>a</sup><https://twitter.com/bradfitz>

<sup>b</sup><https://github.com/golang/go/issues/35950>

# Go: Scaffolding database migrations

As we prepared the database migration files and embedded them into the db package, we are now left to implement the details required to process these migrations. Let's start with extending the FS type, to actually provide functionality for reading them.

```
1 // ReadFile returns decoded file contents from FS
2 func (fs FS) ReadFile(filename string) ([]byte, error) {
3     if val, ok := fs[filename]; ok {
4         return base64.StdEncoding.DecodeString(val)
5     }
6     return nil, os.ErrNotExist
7 }
```

Reading a file from our FS type is super simple. All we need to do is check that the file we are trying to read actually exists, and base64 decode it. In order to be somewhat compliant to the standard practice for reading files, we return an `os.ErrNotExist` error in case the file we're trying to read isn't there.

Let's also add a migrations helper, that will list the migrations inside the filesystem, so that we can just loop over them and run them one by one. It only lists actual migration files (\*.up.sql), and doesn't return any other files embedded (migrations.sql for one). The files are sorted into their expected execution order.

```
1 // Migrations returns list of SQL files to execute
2 func (fs FS) Migrations() []string {
3     result := []string{}
4     for filename, contents := range fs {
5         // skip empty files
6         if contents == "" {
7             continue
8         }
9         if matched, _ := filepath.Match("*.up.sql", filename); matched {
10             result = append(result, filename)
11         }
12     }
13     sort.Strings(result)
14     return result
15 }
```

We also need a structure for our migration schema (migration.sql), and this is it:

```
1 package db
2
3 type (
4     migration struct {
5         Project      string `db:"project"`
6         Filename      string `db:"filename"`
7         StatementIndex int    `db:"statement_index"`
8         Status        string `db:"status"`
9     }
10 )
11
12 func (migration) Fields() []string {
13     return []string{"project", "filename", "statement_index", "status"}
14 }
```

The fields are as follows:

- project denotes the migration group / service (e.g. stats),
- filename is a migration from that group (2019-11-26-092610-description-here.up.sql),
- statement\_index - the sequential statement index from a migration,
- status - this will either be “ok”, or the error produced from a failing migration

To explain statement\_index: each migration can be several SQL queries. A common case is to import the complete database schema inside one migration, where a migration will be several CREATE TABLE statements in a single file. We need to split this file with a delimiter ;, which should be at the end of the line. For this we can use the ;\$ regular expression to split the migration into statements, remove empty statements, and return a list of them. The list key is the statement index.

The Fields() function there is to help us issue queries against the database. In order to issue a full INSERT (or REPLACE,...), we need to list the database table fields. The other way to do it would be to use reflection, but that’s overkill. This function helps us to remove some of the field stutter which applies to data insertion/update.

We can produce a helper function, that takes the result of FS.ReadFile:

```

1 func statements(contents []byte, err error) ([]string, error) {
2     result := []string{}
3     if err != nil {
4         return result, err
5     }
6     stmts := regexp.MustCompilePOSIX(";").Split(string(contents), -1)
7     for _, stmt := range stmts {
8         stmt = strings.TrimSpace(stmt)
9         if stmt != "" {
10             result = append(result, stmt)
11         }
12     }
13     return result, nil
14 }

```

This way, we can produce the statements from a migration file in the following way:

```

1 stmts, err := statements(migrations["stats"].ReadFile("migrations.sql"))

```

The result from `ReadFile` is expanded into the function parameters here. The error checking here is a bit reduced, on account that the error is just propagated forward.

## The migration API

So now that we have all the required structures and functions to read migrations and split them into statements, we need to consider the migration API. I would like to provide the following functionality from this package:

- `List() []string` - should list the service names for the embedded migrations
- `Print(project string) error` - output the migration queries with `log.Println`,
- `Run(project string, db *sqlx.DB) error` - pass a database handle and project name to run migrations

This gives us the ability to write a cli tool, that can print or run specific migrations. Let's start with the low hanging fruit: `List()`:



```
1 func List() []string {
2     result := []string{}
3     for k, _ := range migrations {
4         result = append(result, k)
5     }
6     return result
7 }
```

The function will run only once, and isn't performance intensive, but for a small exercise, let's rewrite that into an efficiently allocated function. I do this so often that it becomes second nature, just to avoid inefficient allocations with `append()`.

```
1 func List() []string {
2     result := make([]string, len(migrations))
3     i := 0
4     for k, _ := range migrations {
5         result[i] = k
6     }
7     return result
8 }
```

Which leaves us with `Print` and `Run` functions. Let's start with `Print`, since it doesn't require us to set up a database just yet. The goal is to get the project filesystem and list the statements for the contained migrations. Let's implement `cmd/db-migrate-cli/main.go`:

```
1 package main
2
3 import (
4     "log"
5
6     "github.com/titpetric/microservice/db"
7 )
8
9 func main() {
10     log.Printf("Migration projects: %+v", db.List())
11     log.Println("Migration statements for stats")
12     if err := db.Print("stats"); err != nil {
13         log.Printf("An error occurred: %+v", err)
14     }
15 }
```

We will extend this file over our development, but for now we are only interested in verifying that the API which we will create is functional. We could make unit tests, but this is just as good as we

need this tool in any case. We cannot verify validity of the data just yet, the only thing we can do is to verify that the data is there, and can be read.

```
1 package db
2
3 import (
4     "fmt"
5     "log"
6
7     "github.com/pkg/errors"
8 )
9
10 func Print(project string) error {
11     fs, ok := migrations[project]
12     if !ok {
13         return errors.Errorf("Migrations for '%s' don't exist", project)
14     }
15
16     printQuery := func(idx int, query string) error {
17         log.Println()
18         log.Println("-- Statement index:", idx)
19         log.Println(query)
20         log.Println()
21         return nil
22     }
23
24     migrate := func(filename string) error {
25         log.Println("Printing migrations from", filename)
26         if stmts, err := statements(fs.ReadFile(filename)); err != nil {
27             return errors.Wrap(err, fmt.Sprintf("Error reading migration: %s", filename))
28         } else {
29             for idx, stmt := range stmts {
30                 if err := printQuery(idx, stmt); err != nil {
31                     return err
32                 }
33             }
34         }
35         return nil
36     }
37
38     // print main migration
39     if err := migrate("migrations.sql"); err != nil {
40         return err
41     }
42 }
```

```

41     }
42
43     // print service migrations
44     for _, filename := range fs.Migrations() {
45         if err := migrate(filename); err != nil {
46             return err
47         }
48     }
49     return nil
50 }

```

The structure here is a bit shorter than what we expect from Run, as we don't need to log the statement indices in the migration struct. We print the initial migrations.sql, as well as every other migration contained in the migration index for the project.

After running make and building all our binaries, we can run the db-migrate-cli cli to verify that the migration statements are being printed.

```

1  # ./build/db-migrate-cli-linux-amd64
2  2019/11/26 11:20:00 Migration projects: [stats]
3  2019/11/26 11:20:00 Migration statements for stats
4  2019/11/26 11:20:00 Printing migrations from migrations.sql
5  2019/11/26 11:20:00
6  2019/11/26 11:20:00 -- Statement index: 0
7  2019/11/26 11:20:00 CREATE TABLE IF NOT EXISTS `migrations` (
8    `project` varchar(16) NOT NULL COMMENT 'Microservice or project name',
9    `filename` varchar(255) NOT NULL COMMENT 'yyyy-mm-dd-HHMMSS.sql',
10   `statement_index` int(11) NOT NULL COMMENT 'Statement number from SQL file',
11   `status` text NOT NULL COMMENT 'ok or full error message',
12   PRIMARY KEY (`project`,`filename`)
13 ) ENGINE=InnoDB DEFAULT CHARSET=utf8
14 2019/11/26 11:20:00
15 2019/11/26 11:20:00 Printing migrations from 2019-11-26-092610-description-here.up.s\
16  ql
17 2019/11/26 11:20:00
18 2019/11/26 11:20:00 -- Statement index: 0
19 2019/11/26 11:20:00 -- Hello world
20 2019/11/26 11:20:00

```

We now verified that:

- we can list our db schema migration projects (stats),
- we have our migration.sql file embedded,

- we have some migrations contained, notably -- Hello world :)

We can now move onto implementing execution of the migration against a real database.

# Drone CI: Testing database migrations

Since we can now list our database migrations, the next logical step is testing them out on a real database. For that we still need to implement our Run function. Based on what we have in Print(), we only need to update the migrate() function.

```
1  execQuery := func(idx int, query string, useLog bool) error {
2      if useLog {
3          log.Println()
4          log.Println("-- Statement index:", idx)
5          log.Println(query)
6          log.Println()
7      }
8      if _, err := db.Exec(query); err != nil && err != sql.ErrNoRows {
9          return err
10     }
11     return nil
12 }
```

Our printQuery function adds database execution over the same output that we already know. We skip the printing of the query if useLog = false, namely for the migrations.sql file. We still need to track the statement index in the migration{} struct, so we need to update what we have in migrate():

```
1  migrate := func(filename string) error {
2      log.Println("Running migrations from", filename)
3
4      status := migration{
5          Project: project,
6          Filename: filename,
7      }
8
9      // we can't log the main migrations table
10     useLog := (filename != "migrations.sql")
11     if useLog {
12         if err := db.Get(&status, "select * from migrations where project=? and filename\
13 =?", status.Project, status.Filename); err != nil && err != sql.ErrNoRows {
```

```

14     return err
15 }
16 if status.Status == "ok" {
17     log.Println("Migrations already applied, skipping")
18     return nil
19 }
20 }

```

In this section we set the initial migration structure, and fetch the status of it from the database. If the migration status is “ok”, this means the migration was already fully applied, and we can skip it.

```

1 up := func() error {
2     stmts, err := statements(fs.ReadFile(filename))
3     if err != nil {
4         return errors.Wrap(err, fmt.Sprintf("Error reading migration: %s", filename))
5     }
6
7     for idx, stmt := range stmts {
8         // skip stmt if it has already been applied
9         if idx >= status.StatementIndex {
10             status.StatementIndex = idx
11             if err := execQuery(idx, stmt, useLog); err != nil {
12                 status.Status = err.Error()
13                 return err
14             }
15         }
16     }
17     status.Status = "ok"
18     return nil
19 }

```

Just like before, we loop over the statements, and apply them if they haven’t been applied yet. In case a migration failed from some external cause (like the database going offline), we also support resuming the migrations by checking the statement index. If the statement index is equal or higher than the logged statement index, then the migration statement will be executed.

We probably don’t need this code, but, it also depends if you will keep a persistent database for testing the migrations, where it’s more likely that you might end up with a broken state, especially while testing migrations for development purposes.

And finally:

```

1  err := up()
2  if useLog {
3      // log the migration status into the database
4      set := func(fields []string) string {
5          sql := make([]string, len(fields))
6          for k, v := range fields {
7              sql[k] = v + "=: " + v
8          }
9          return strings.Join(sql, ", ")
10     }
11     if _, err := db.NamedExec("replace into migrations set "+set(status.Fields()), s\
12 tatus); err != nil {
13         log.Println("Updating migration status failed:", err)
14     }
15 }
16 return err
17 }

```

Here we update the migration status into the database, regardless if it produced an error or not. The error or success is written in the `status.Status` field, and we update the complete record.

## Configuring Drone CI for database integration tests

It should be really simple to run the database integration tests from Drone. Let's configure a database service, and a special makefile target to run the migrations.

```

1  - name: migrations
2    image: percona/percona-server:8.0.17
3    user: root
4    pull: always
5    commands:
6      - yum -q -y install make
7      - "bash -c 'while : ; do sleep 1 ; $(cat < /dev/null > /dev/tcp/mysql-test/3306\
8 ) && break ; done' 2>/dev/null"
9      - make migrate
10
11 services:
12 - name: mysql-test
13   pull: always
14   image: percona/percona-server:8.0.17
15   ports:
16     - 3306

```

```

17 environment:
18     MYSQL_ROOT_PASSWORD: default

```

Here we slightly hacked together support for running the migrations by:

1. installing make into the migration step, using the provided database container (all you need is a mysql client)
2. a bash-fu check using `/dev/tcp` to wait until our database service came online before running migrations
3. the database service itself, with a configured “default” mysql root password

And the Makefile target should look like this:

```

1 migrate: $(shell ls -d db/schema/*/migrations.sql | xargs -n1 dirname | sed -e 's/db\'
2 .schema./migrate./')
3     @echo OK.
4
5 migrate.%: export SERVICE = $*
6 migrate.%: export MYSQL_ROOT_PASSWORD = default
7 migrate.%:
8     mysql -h mysql-test -u root -p$(MYSQL_ROOT_PASSWORD) -e "CREATE DATABASE $(SERVICE\'
9 );"
10    ./build/db-migrate-cli-linux-amd64 -service $(SERVICE) -db-dsn "root:$(MYSQL_ROOT_\'
11 PASSWORD)@tcp(mysql-test:3306)/$(SERVICE)" -real=true
12    ./build/db-migrate-cli-linux-amd64 -service $(SERVICE) -db-dsn "root:$(MYSQL_ROOT_\'
13 PASSWORD)@tcp(mysql-test:3306)/$(SERVICE)" -real=true

```

The makefile target produces the database for our service, and then runs our db-migration program. We still need to configure some flags for the program, but as you see, we pass the service name, the database connection DSN, and the flag `-real=true` to actually execute the migrations on the database.

We run the migrations twice, so we make sure that our migration status is logged correctly as well. All the migrations in the second run must be skipped.

Let's add support for this into `cmd/db-migrate-cli/main.go`:



```
1  package main
2
3  import (
4      "flag"
5      "log"
6
7      _ "github.com/go-sql-driver/mysql"
8
9      "github.com/jmoiron/sqlx"
10     "github.com/titpetric/microservice/db"
11 )
12
13 func main() {
14     var config struct {
15         db struct {
16             DSN      string
17             Driver string
18         }
19         Real    bool
20         Service string
21     }
22     flag.StringVar(&config.db.Driver, "db-driver", "mysql", "Database driver")
23     flag.StringVar(&config.db.DSN, "db-dsn", "", "DSN for database connection")
24     flag.StringVar(&config.Service, "service", "", "Service name for migrations")
25     flag.BoolVar(&config.Real, "real", false, "false = print migrations, true = run mi\
26 grations")
27     flag.Parse()
28
29     if config.Service == "" {
30         log.Printf("Available migration services: %v", db.List())
31         log.Fatal()
32     }
33
34     switch config.Real {
35     case true:
36         if handle, err := sqlx.Connect(config.db.Driver, config.db.DSN); err != nil {
37             log.Fatalf("Error connecting to database: %v", err)
38         } else {
39             if err := db.Run(config.Service, handle); err != nil {
40                 log.Fatalf("An error occurred: %v", err)
41             }
42         }
43     default:
```

```

44     if err := db.Print(config.Service); err != nil {
45         log.Fatalf("An error occurred: %v", err)
46     }
47 }
48 }

```

Here we do a couple of things:

- import the mysql database driver
- add configuration options with the standard library flag package,
- print services if no service is passed
- print migrations if real=false,
- connect to database and run migrations if real=true

All that's left to see is if our migrations execute. Let's take a look at the output of our migration target:

```

1  [migrations:0] + yum -q -y install make
2  [migrations:1] + bash -c 'while : ; do sleep 1 ; $(cat < /dev/null > /dev/tcp/mysql\
3  -test/3306) && break ; done' 2>/dev/null
4  [migrations:2] + make migrate
5  [migrations:3] mysql -h mysql-test -u root -pdefault -e "CREATE DATABASE stats;"
6  [migrations:4] mysql: [Warning] Using a password on the command line interface can b\
7  e insecure.
8  [migrations:5] ./build/db-migrate-cli-linux-amd64 -service stats -db-dsn "root:defau\
9  lt@tcp(mysql-test:3306)/stats" -real=true
10 [migrations:6] 2019/11/26 11:22:06 Running migrations from migrations.sql
11 [migrations:7] 2019/11/26 11:22:06 Running migrations from 2019-11-26-092610-descrip\
12 tion-here.up.sql
13 [migrations:8] 2019/11/26 11:22:06
14 [migrations:9] 2019/11/26 11:22:06 -- Statement index: 0
15 [migrations:10] 2019/11/26 11:22:06 -- Hello world
16 [migrations:11] 2019/11/26 11:22:06
17 [migrations:12] ./build/db-migrate-cli-linux-amd64 -service stats -db-dsn "root:defa\
18 ult@tcp(mysql-test:3306)/stats" -real=true
19 [migrations:13] 2019/11/26 11:22:06 Running migrations from migrations.sql
20 [migrations:14] 2019/11/26 11:22:06 Running migrations from 2019-11-26-092610-descri\
21 ption-here.up.sql
22 [migrations:15] 2019/11/26 11:22:06 Migrations already applied, skipping
23 [migrations:16] OK.

```

Great success - the migrations are executed, and the second migration run doesn't end up with an error. We can see that the migration is skipped, as it has already been applied to the database.

# Go: Database first struct generation

As we set up our protobuf structures and data migrations, we will need to interface between them. Unfortunately, protobuf structs don't have good support for [adding go tags](#)<sup>19</sup>, so we can't easily pin db tags to it, nor should we really. There's always going to be some mismatching with what a RPC request/response is, and what's actually stored in the database.

SQL databases have pretty good support over their `information_schema` interface. We can list the database tables in a schema, and individually get their fields, field types and even comments so we have all the information about structures available. Since we're using Drone CI to test migrations, we can use the same database state to source our Go structures.

## Database schema data structures and helpers

We care both about the tables themselves, as well as the column names, types, and any column comment that we might include in the DB schema. We can get this informations from `information_schema`, from the `tables` and `columns` tables respectively.

Create a `cmd/db-schema-cli/types.go` file:

```
1 package main
2
3 type Table struct {
4     Name      string `db:"TABLE_NAME"`
5     Comment   string `db:"TABLE_COMMENT"`
6
7     Columns []*Column
8 }
9
10 var TableFields = []string{"TABLE_NAME", "TABLE_COMMENT"}
11
12 type Column struct {
13     Name      string `db:"COLUMN_NAME"`
14     Type      string `db:"COLUMN_TYPE"`
15     Key       string `db:"COLUMN_KEY"`
16     Comment   string `db:"COLUMN_COMMENT"`
17
18     // Holds the clean data type (`int` not `int(11) unsigned` ...)
```

---

<sup>19</sup><https://github.com/golang/protobuf/issues/52>

```

19     DataType string `db:"DATA_TYPE"`
20 }
21
22 var ColumnFields = []string{"COLUMN_NAME", "COLUMN_TYPE", "COLUMN_KEY", "COLUMN_COMMENT",
23 "DATA_TYPE"}

```

And we can already anticipate that we will need a function to convert the SQL field names we declare into camel case. We can do it without regular expressions, like this:

```

1 func camel(input string) string {
2     parts := strings.Split(input, "_", -1)
3     for k, v := range parts {
4         parts[k] = strings.ToUpper(v[0:1]) + v[1:]
5     }
6     return strings.Join(parts, "")
7 }

```

## Querying information schema

In our `cmd/db-schema-cli/main.go` file, we need to set up the database connection, query the schema and produce the raw structures that we need to generate any kind of code output.

```

1 func main() {
2     var config struct {
3         db struct {
4             DSN string
5             Driver string
6         }
7         Schema string
8         Format string
9     }
10    flag.StringVar(&config.db.Driver, "db-driver", "mysql", "Database driver")
11    flag.StringVar(&config.db.DSN, "db-dsn", "", "DSN for database connection")
12    flag.StringVar(&config.Schema, "schema", "", "Schema name to print tables for")
13    flag.StringVar(&config.Format, "format", "go", "Output formatting")
14    flag.Parse()
15
16    handle, err := sqlx.Connect(config.db.Driver, config.db.DSN)
17    if err != nil {
18        log.Fatalf("Error connecting to database: %+v", err)
19    }

```

We set up configuration flags similarly to db-migrate cli program, taking the database driver, connection DSN, the schema we want to inspect, and finally an output format. By default we will generate Go code, but other output formats (and possibly new options) will come in handy for generating the database documentation.

Next, we read the table information, and fill out the column information for each table.

```

1  // List tables in schema
2  tables := []*Table{}
3  fields := strings.Join(TableFields, ", ")
4  err = handle.Select(&tables, "select "+fields+" from information_schema.tables where\
5  table_schema=? order by table_name asc", config.Schema)
6  if err != nil {
7      log.Println("Error listing database tables")
8      log.Fatal(err)
9  }
10
11 // List columns in tables
12 for _, table := range tables {
13     fields := strings.Join(ColumnFields, ", ")
14     err := handle.Select(&table.Columns, "select "+fields+" from information_schema.co\
15 lumns where table_schema=? and table_name=? order by ordinal_position asc", config.S\
16 chema, table.Name)
17     if err != nil {
18         log.Println("Error listing database columns for table:", table.Name)
19         log.Fatal(err)
20     }
21 }

```

All we need now is some bridging code to invoke our Go code rendering:

```

1  // Render go structs
2  if config.Format == "go" {
3      if err := renderGo(config.Schema, tables); err != nil {
4          log.Fatal(err)
5      }
6  }

```

We are passing the schema name in order to set the generated package name, as well as all the table information.

## Type conversions

When we're talking about database columns, we have two type columns to consider: `data_type`, which holds the raw type, and `column_type`, which is the type augmented with its restrictions, like the length, or if a numeric type is a signed/unsigned number. The latter is important to differentiate `int64` and `uint64`.

We can split the types into three distinct requirements:

- numeric types which can be signed/unsigned (combined type),
- simple types which have direct translations into Go,
- special types that need a package import (`datetime` -> `*time.Time`)

The mappings for each type should be simple:

```

1  var numericTypes map[string]string = map[string]string{
2      "tinyint": "int8",
3      "smallint": "int16",
4      // `mediumint` - this one would, technically, be int24 (3 bytes), but
5      "mediumint": "int32",
6      "int": "int32",
7      "bigint": "int64",
8  }
9
10 func isNumeric(column *Column) (string, bool) {
11     val, ok := numericTypes[column.DataType]
12     return val, ok
13 }

```

All the numeric types have the characteristic, that an unsigned value just prepends `u` in front of the signed mapped type. This way an unsigned `int32` will become an `uint32`.

```

1  var simpleTypes map[string]string = map[string]string{
2      "char": "string",
3      "varchar": "string",
4      "text": "string",
5      "longtext": "string",
6      "mediumtext": "string",
7      "tinytext": "string",
8      "longblob": "[]byte",
9      "blob": "[]byte",
10     "varbinary": "[]byte",

```

```

11  // `float` and `double` are here since they don't have unsigned modifiers
12  "float": "float32",
13  "double": "float64",
14  // `decimal` - double stored as string, \o/
15  "decimal": "string",
16  }
17
18  func isSimple(column *Column) (string, bool) {
19      val, ok := simpleTypes[column.DataType]
20      return val, ok
21  }

```

Simple types, like numeric types, return a 1-1 type mapping. As these values are not signed/unsigned, the returned mapped type will stay as-is.

```

1  type specialType struct {
2      Import string
3      Type   string
4  }
5
6  var specialTypes map[string]specialType = map[string]specialType{
7      "date":      specialType{"time", "*time.Time"},
8      "datetime": specialType{"time", "*time.Time"},
9      "time":      specialType{"time", "*time.Time"},
10     "timestamp": specialType{"time", "*time.Time"},
11     // `enum` and `set` aren't implemented
12     // `year` isn't implemented
13 }
14
15 func isSpecial(column *Column) (specialType, bool) {
16     val, ok := specialTypes[column.DataType]
17     return val, ok
18 }

```

The special types are a bit more complex in the sense that they also provide an import. We cover the built in date/time fields, but we do have other fields that are currently omitted. MySQL has a JSON field type, for which we'd have to import [github.com/jmoiron/sqlx/types](https://github.com/jmoiron/sqlx/types), and use `types.JSONText` in order to have it usable. There are also other types there, like `GzippedText`, and an implementation of a `BIT(1)` type that scans the value into a `bool` field. We won't be implementing these for our use case, but the example just goes to show, how quickly the requirements can grow.

Since a schema can have many types that import a package, we need to keep these packages unique:

```

1 func contains(set []string, value string) bool {
2     for _, v := range set {
3         if v == value {
4             return true
5         }
6     }
7     return false
8 }

```

The contains() helper helps us out by having a way to check if an import has already been added. Which leaves us with just the actual function to generate the relevant Go code:

```

1 func renderGo(schema string, tables []*Table) error {
2     imports := []string{}
3
4     resolveType := func(column *Column) (string, error) {
5         if val, ok := isSimple(column); ok {
6             return val, nil
7         }
8         if val, ok := isNumeric(column); ok {
9             isUnsigned := strings.Contains(strings.ToLower(column.Type), "unsigned")
10            if isUnsigned {
11                return "u" + val, nil
12            }
13            return val, nil
14        }
15        if val, ok := isSpecial(column); ok {
16            if !contains(imports, val.Import) {
17                imports = append(imports, val.Import)
18            }
19            return val.Type, nil
20        }
21        return "", errors.Errorf("Unsupported SQL type: %s", column.DataType)
22    }
23
24    // Loop through tables/columns, return type error if any
25    // This also builds the `imports` slice for codegen lower
26    for _, table := range tables {
27        for _, column := range table.Columns {
28            if _, err := resolveType(column); err != nil {
29                return err
30            }
31        }
32    }
33 }

```



```

31     }
32 }

```

Before generating any code, we pass through all columns and call `resolveType`, to declare all imports, and to error out if any of the used types on the table can't be resolved to a Go type. We want to do that as soon as possible.

```

1  fmt.Printf("package %s\n", schema)
2  fmt.Println()
3
4  // Print collected imports
5  if len(imports) > 0 {
6      fmt.Println("import (")
7      for _, val := range imports {
8          fmt.Printf("\t%s\n", val)
9      }
10     fmt.Println(")")
11     fmt.Println()
12 }

```

Printing the declared imports is trivial. If we aren't relying on any special types, the imports lines will be omitted fully.

```

1  for _, table := range tables {
2      fields := []string{}
3      primary := []string{}
4      if table.Comment != "" {
5          fmt.Println("//", table.Comment)
6      }
7      fmt.Printf("type %s struct {\n", camel(table.Name))
8      for idx, column := range table.Columns {
9          fields = append(fields, column.Name)
10         if column.Key == "PRI" {
11             primary = append(primary, column.Name)
12         }
13
14         if column.Comment != "" {
15             if idx > 0 {
16                 fmt.Println()
17             }
18             fmt.Printf("    // %s\n", column.Comment)
19         }

```

```

20     columnType, _ := resolveType(column)
21     fmt.Printf("    %s %s `db:\"%s\"`\n", camel(column.Name), columnType, column.Nam\
22 e)
23 }
24 fmt.Println("")
25 fmt.Println()
26 fmt.Printf("func (*%s) Fields() []string {\n", camel(table.Name))
27 if len(fields) > 0 {
28     fmt.Printf("\treturn []string{\"%s\"}\n", strings.Join(fields, "\", \"))
29 } else {
30     fmt.Printf("\treturn []string{}\n")
31 }
32 fmt.Println("")
33 fmt.Println()
34 fmt.Printf("func (*%s) PrimaryFields() []string {\n", camel(table.Name))
35 if len(primary) > 0 {
36     fmt.Printf("\treturn []string{\"%s\"}\n", strings.Join(primary, "\", \"))
37 } else {
38     fmt.Printf("\treturn []string{}\n")
39 }
40 fmt.Println("")
41 }
42 return nil

```

Finally, for each table we collect the table fields, the primary key fields, and print out the relevant Go structures and functions to support them. A big part of generating the `Fields` and `PrimaryFields` functions is to avoid reflection. A migration to add a new field could break the service, if a `select * from [table]...` style SQL query would be used. A newly added field would result in an error executing the query. By selecting fields explicitly, we can avoid this scenario.

In fact, there's little use to keep `Fields` and `PrimaryFields` as functions on the structure, so we'll immediately modify this to generate a public, service-prefixed variable.

```

1  fmt.Printf("var %sFields = ", camel(table.Name))
2  if len(fields) > 0 {
3      fmt.Printf("[[]string{\"%s\"}", strings.Join(fields, "\", \"))
4  } else {
5      fmt.Printf("[[]string{}")
6  }
7  fmt.Println()
8  fmt.Printf("var %sPrimaryFields = ", camel(table.Name))
9  if len(primary) > 0 {
10     fmt.Printf("[[]string{\"%s\"}", strings.Join(primary, "\", \"))

```

```

11 } else {
12     fmt.Printf("[string{}")
13 }
14 fmt.Println()

```

All that's left to do at this point is to run and verify the generated code.

```

1 package stats
2
3 type Migrations struct {
4     // Microservice or project name
5     Project string `db:"project"`
6
7     // yyyy-mm-dd-HHMMSS.sql
8     Filename string `db:"filename"`
9
10    // Statement number from SQL file
11    StatementIndex int32 `db:"statement_index"`
12
13    // ok or full error message
14    Status string `db:"status"`
15 }
16
17 var MigrationsFields = []string{"project", "filename", "statement_index", "status"}
18 var MigrationsPrimaryFields = []string{"project", "filename"}

```

That's exactly what we wanted. We generate the full Go structure with all required metadata to support our Go development workflows. The comments are generated from the database schema, so we have this information available at any time - if a DBA is managing the database schema in their favorite editor, or if a developer is reading the Go struct definitions or internal godoc pages. Documentation accessibility is built into our process.

All that's left to do now is to include our generator into the Makefile, under the migrate.%: target:

```

1 ./build/db-schema-cli-linux-amd64 -schema $(SERVICE) -db-dsn "root:$(MYSQL_ROOT_PASS\
2 WORD)@tcp(mysql-test:3306)/$(SERVICE)" > server/$(SERVICE)/types_db.go

```

Now, we have some specific requirements to consider, which we are solving by generating types\_db.go under individual services which we are making. Let's consider how having a global types package would be problematic:

- It's likely we would have naming conflicts for tables (example: log table existing in multiple services),

- Prefixing the structures with the service name would introduce stutter (`stats.stats_log` becomes `StatsStatsLog`)
- We really only need the database structures in our service implementation. We need only to convert them to PB.

In some cases, the possible mapping between DB and PB structures could be 1-1, but it's likely we're going to have different types for PB (javascript client's can't really handle uint64 types so they have to be strings), and we will always have data which is available in the database, which must never be exposed publicly (JSON?).

In fact, I have half of mind to add `json:"-"` tags to all the fields and see how far that brings me. It really isn't a bad idea, so...

```

1  --- a/cmd/db-schema-cli/render-go.go
2  +++ b/cmd/db-schema-cli/render-go.go
3  @@ -128,7 +128,7 @@ func renderGo(schema string, tables []*Table) error {
4
5             fmt.Printf("        // %s\n", column.Comment)
6
7             }
8             columnType, _ := resolveType(column)
9             -           fmt.Printf("        %s %s `db:\"%s\"`\n", camel(column.Name), co\
10 lumnType, column.Name)
11             +           fmt.Printf("        %s %s `db:\"%s\"` json:\"-\"`\n", camel(column\
12 n.Name), columnType, column.Name)
13             }
14             fmt.Println("{}")
15             fmt.Println()

```

I think this is what people are referring to when they say you have to future-proof your code. I'm not counting on the fact that the structures defined here will never be exported, we might have to come back and patch our code generation in this case, but it's way better to start with paranoid defaults, and then open up your system. It's way harder to disable something, than to enable it.

## Moving forward

As we didn't create a real migration that would create any tables yet, the only table that's dumped as Go code is the Migrations table. Our next steps will be to create some real migrations for our table, and see if we can already define some corner cases where we're going to need to go outside of the current implementation.

As we said, it's our goal to actively generate the documentation from SQL as our data source, which also means we're going to be generating some markdown output from the database tables we'll be working with.

# Go: Generating database schema documentation

Since we are testing migrations and generating Go struct types from the database schema, we also have all the available information to generate documentation snippets for each schema. We will generate markdown formatted tables with the final database schema, after all the migrations have been applied.

## Updating the Go generators

We need to adjust the generator code, since each table will need to be rendered in an individual markdown file. We need to add a parameter to db-schema-cli, which we will pass to specify an output folder.

```
1  --- a/cmd/db-schema-cli/main.go
2  +++ b/cmd/db-schema-cli/main.go
3  @@ -18,11 +18,13 @@ func main() {
4      }
5      Schema string
6      Format string
7  +      Output string
8  }
9      flag.StringVar(&config.db.Driver, "db-driver", "mysql", "Database driver")
10     flag.StringVar(&config.db.DSN, "db-dsn", "", "DSN for database connection")
11     flag.StringVar(&config.Schema, "schema", "", "Schema name to print tables fo\
12 r")
13     flag.StringVar(&config.Format, "format", "go", "Output formatting")
14  +     flag.StringVar(&config.Output, "output", "", "Output folder (mandatory)")
15     flag.Parse()
16
17  +     if config.Output == "" {
18  +         log.Fatal("Missing -output parameter, please specify output folder")
19  +     }
```

We can now specify `-output [folder]` as an argument to db-schema-cli. Let's adjust the signature for `renderGo` to take this parameter, so we will generate Go code in the folder specified with the same parameter. We can also add `renderMarkdown` at the same time:

```

1  // Render go structs
2  if config.Format == "go" {
3      if err := renderGo(config.Output, config.Schema, tables); err != nil {
4          log.Fatal(err)
5      }
6  }
7
8  // Render markdown tables
9  if config.Format == "markdown" {
10     if err := renderMarkdown(config.Output, config.Schema, tables); err != nil {
11         log.Fatal(err)
12     }
13 }

```

Please create a `render-md.go` file, which contains the following:

```

1 package main
2
3 func renderMarkdown(basePath string, schema string, tables []*Table) error {
4     return nil
5 }

```

The stub is here just so we can compile the source as we need to do some housekeeping for the `renderGo` function. Since we want to write to a file from `renderGo`, we will create a bytes buffer, and modify our various `fmt.Print*` calls into `fmt.Fprint*`, which takes an `io.Writer` as an additional first parameter. The code in the `renderGo` function becomes something like this:

```

1 buf := bytes.NewBuffer([]byte{})
2
3 fmt.Fprintf(buf, "package %s\n", schema)
4 fmt.Fprintln(buf)
5 // ...

```

Now, we can get the generated source code by calling `buf.Bytes()`. We can also load the `go/format` package, and push the source code through a formatter. At the tail end of the `renderGo` function, add the following code which formats the source and saves `types_db.go`.

```

1 filename := path.Join(basePath, "types_db.go")
2 contents := buf.Bytes()
3
4 formatted, err := format.Source(contents)
5 if err != nil {
6     // fall back to unformatted source to inspect
7     // the saved file for the error which occurred
8     formatted = contents
9     log.Println("An error occurred while formatting the go source: %s", err)
10    log.Println("Saving the unformatted code")
11 }
12
13 fmt.Println(filename)
14
15 return ioutil.WriteFile(filename, formatted, 0644)

```

The Makefile command to invoke the go struct generator changes. Let's just explicitly add `-format` and `-output` so we can generate the file in the specified folder.

```

1 - > server/${SERVICE}/types_db.go
2 + -format go -output server/${SERVICE}

```

We can verify if `types_db.go` is still generated correctly. Let's run `make` and verify. Somewhere in the output of `make`, the following line shows up:

```

1 [migrations:18] server/stats/types_db.go

```

This means that the `fmt.Println` was executed, and the file was written just after that. If we open the file and verify, we will see that the generated struct is there, so we didn't break anything.

## Implementing a markdown output renderer

We started with an empty stub, and now it's up to us to implement the renderer to Markdown. For this, we need to set some simple requirements:

- we want to create the output dir
- each table needs to be in its own file (`[table_name].md`),
- we will print the table name as the title,
- optionally we will print the table description if any,
- and produce a padded, human readable, markdown table with the sql table structure

We can start with the very simple one:

```

1 // create output folder
2 if err := os.MkdirAll(basePath, 0755); err != nil {
3     return err
4 }

```

With a call to `MkdirAll` we recursively create the output folder. Since `renderGo` also needs to create a folder, we copy the snippet into `renderGo` as well. Now, we can move on to loop over the tables in our schema, and individually render the markdown contents for them.

```

1 // generate individual markdown files with schema
2 for _, table := range tables {
3     filename := path.Join(basePath, table.Name+".md")
4
5     contents := renderMarkdownTable(table)
6     if err := ioutil.WriteFile(filename, contents, 0644); err != nil {
7         return err
8     }
9
10    fmt.Println(filename)
11 }
12 return nil

```

This function will break out if an error occurs on `ioutil.WriteFile`. Possible errors include you not having write permissions for the file, or out of disk space. As `renderMarkdown` returns an error, we can pass this information back to `main()` and error out with a relevant error message.

Now we can move to the individual table markdown output generator. Now, markdown can read tables in the following format:

```

1 |Name|Type|Key|Comment|
2 |--|--|--|--|
3 |1|2|3|4|

```

The obvious issue is that without correct padding, this isn't very human readable. We want to generate output as close to this as possible:

```

1 | Name | Type | Key | Comment |
2 |-----|-----|-----|-----|
3 | 1    | 2    | 3    | 4          |

```

Now, we immediately notice something - the heading of the table also counts as the length of a column, and adds to the padding below. We need to set the initial padding to the length of the column title:



```

1 // calculate initial padding from table header
2 titles := []string{"Name", "Type", "Key", "Comment"}
3 padding := map[string]int{}
4 for _, v := range titles {
5     padding[v] = len(v)
6 }

```

After this, we need to loop through each table column, and adjust the padding for each markdown column depending on the data returned. To make the code a bit clearer, we also create a local `max(a, b int) int` function.

```

1 max := func(a, b int) int {
2     if a > b {
3         return a
4     }
5     return b
6 }
7
8 // calculate max length for columns for padding
9 for _, column := range table.Columns {
10     padding["Name"] = max(padding["Name"], len(column.Name))
11     padding["Type"] = max(padding["Type"], len(column.Type))
12     padding["Key"] = max(padding["Key"], len(column.Key))
13     padding["Comment"] = max(padding["Comment"], len(column.Comment))
14 }

```

Now, we should have the required length for each column in the padding variable. Naively, we could resort to something like `strings.Repeat` and length checks to calculate the padding required based on the cell contents length, but we can resort to the formatting features of `fmt` print functions, namely the width value for any type.

```

1 %f      default width, default precision
2 %9f     width 9, default precision
3 %.2f    default width, precision 2
4 %9.2f   width 9, precision 2
5 %9.f    width 9, precision 0

```

With our particular case, we need to print `%10s` to pad a string to the length of 10. By default, the string is padded on the left, but there's a flag for that as well. We can print `%-10s`, and the field will be left justified, like we want.

```

1 // use fmt.Sprintf to add padding to columns, left align columns
2 format := strings.Repeat("| %-%ds ", len(padding)) + "| \n"
3
4 // %-%ds becomes %-10s, which right-pads string to len=10
5 paddings := []interface{}{
6     padding["Name"],
7     padding["Type"],
8     padding["Key"],
9     padding["Comment"],
10 }
11 format = fmt.Sprintf(format, paddings...)

```

Ultimately, we end up with a formatting string which we can pass to `fmt.Sprintf` and can be used for each row of the table. The string may be something like `| %-10s | %-5s | %-3s | %-50s |`. Now, the print functions in `fmt` take `...interface{}` as the parameters, so unfortunately, we can't just use `titles...` since the types don't match. We need to coerce the type into `[]interface{}`, just like we did with the `paddings` above.

```

1 // create initial buffer with table name
2 buf := bytes.NewBufferString(fmt.Sprintf("# %s\n\n", table.Name))
3
4 // and comment
5 if table.Comment != "" {
6     buf.WriteString(fmt.Sprintf("%s\n\n", table.Comment))
7 }
8
9 // write header row strings to the buffer
10 row := []interface{}{"Name", "Type", "Key", "Comment"}
11 buf.WriteString(fmt.Sprintf(format, row...))

```

After the header row, we need to output the divider. All the padding from the values should just be replaced with dash characters, all we need is to pass a row with empty columns.

```

1 // table header/body delimiter
2 row = []interface{}{"", "", "", ""}
3 buf.WriteString(strings.Replace(fmt.Sprintf(format, row...), " ", "-", -1))

```

And now we are left with printing the columns:

```

1 // table body
2 for _, column := range table.Columns {
3     row := []interface{}{column.Name, column.Type, column.Key, column.Comment}
4     buf.WriteString(fmt.Sprintf(format, row...))
5 }
6
7 // return byte slice for writing to file
8 return buf.Bytes()

```

Edit the Makefile to add the db-schema-cli renderer with the following parameters:

- -format markdown - sets the renderer,
- -output docs/schema/\$(SERVICE) - sets the output path

At the end of the migrate.% target, copy the following line:

```

1 ./build/db-schema-cli-linux-amd64 -schema $(SERVICE) -db-dsn "root:$(MYSQL_ROOT_PA\
2 SSWORD)@tcp(mysql-test:3306)/$(SERVICE)" -format markdown -output docs/schema/$(SERV\
3 ICE)

```

When running make we can verify that the migrations for our services are written.

```

1 ...
2 [migrations:17] ./build/db-schema-cli-linux-amd64 -schema stats -db-dsn "root:default\
3 t@tcp(mysql-test:3306)/stats" -format go -output server/stats
4 [migrations:18] server/stats/types_db.go
5 [migrations:19] ./build/db-schema-cli-linux-amd64 -schema stats -db-dsn "root:default\
6 t@tcp(mysql-test:3306)/stats" -format markdown -output docs/schema/stats
7 [migrations:20] docs/schema/stats/migrations.md
8 [migrations:21] OK.
9 ...

```

And we can check the output to verify that it's padded correctly:

```

1 # migrations
2
3 | Name          | Type          | Key | Comment
4 |-----|-----|-----|-----|
5 | project       | varchar(16)   | PRI | Microservice or project name
6 | filename      | varchar(255)  | PRI | yyyy-mm-dd-HHMMSS.sql
7 | statement_index | int(11)       |     | Statement number from SQL file
8 | status        | text         |     | ok or full error message

```

## A caveat emptor about database migrations

We have now a system in place that:

- tests SQL migrations with Drone CI integration tests,
- generates structs based on the SQL tables in our schema,
- generates documentation snippets for SQL tables

By approaching our service development in this way, we have made possible to create SQL schema and migrations with database specific tooling like [MySQL workbench](https://www.mysql.com/products/workbench/)<sup>20</sup>, while at the same time keeping a single source of truth for the data structures in use.

The SQL schema can be updated with migrations, and the Drone CI steps will ensure that the Go structures are updated as well. If some field is renamed it will be a breaking change and you will need to rename the Go field names in use by hand. Since we're not using the database structures for JSON encoding, we can at least be sure, that any breaking changes in regards to field naming will not trickle out to your end users.

The most significant warning about running migrations separately from the service is that breaking changes in the service need to be deployed in a coordinated way. The SQL migration will break the deployed service and there will be some time before a new version of the service may be running. This is an inherent problematic of distributed systems.

If this is a particular pain point for you, and it should be, you should perform the database migrations in a non breaking way. For example, renaming a field safely might mean the following:

1. Migration: Add a new field,
2. Service: Write both to new and old fields,
3. Migration: `UPDATE table set new_field=old_field;`,
4. Service: Stop writing/referencing old fields,
5. Migration: DROP the old field from the table

---

<sup>20</sup><https://www.mysql.com/products/workbench/>

The process isn't easy to automate but illustrates why in production service deployments, it's good to have a DBA for assistance. Coupling the service and migrations together brings other problems, like having long running migration steps and concurrency issues since your service might be deployed over many replicas and hosts.

Controlled upgrades are usually a task that's performed by humans, as much as we would like to automate this there can be various technical circumstances which limit you as to when you can perform schema migrations or a redeploy of your services, especially in distributed systems.

# Go: Dependency injection with Wire

Our microservices ultimately need at least a database connection in order to query and store data in a MySQL database. We need to pass this object, possibly from the `main()` function, to our service implementation.

## The Go way

I have learned through trial and error, that “Dependency Injection” is a term which manages to evoke a strong emotional response in gophers. As I’ve said before, all that the term represents in Go is “a way to pass stuff into our handlers”, while it means a lot of different things to people familiar with the concept from other programming languages.

The most trivial way of dependency injection would be to declare function arguments in our server constructor. For example, if we would need a `*sqlx.DB`, we would declare a constructor like:

```
1 func New(ctx context.Context, db *sqlx.DB) {  
2     ...  
3 }
```

It would be up to the caller to implement the correct argument list to fill the required dependencies for the constructor. When dealing with a possible large scale RPC scaffolding project, like we are, we are faced with a few pitfalls of this approach.

The proto file definition for our service doesn’t provide any way to set up a list of dependencies which are required for our service. Obviously the service block in the proto file only declared the interface for our RPCs, but it has absolutely no standard way to reference Go structures (like `*sqlx.DB`).

The obvious second pitfall is that our code generators would become exponentially more complex with each added service, and custom service dependencies. While we can assume that we use a single database connection for a service, we can’t really know how unique the dependencies for any given service will be.

Ideally we need a way where we can add new dependencies, and not have to refactor every service we have written until then to keep some idea of a consistent Go API.

## Runtime vs. compile time

In addition to the standard Go way to pass parameters to constructors, there are two pattern to dependency injection in the wild. Each one of them has a set of constraints, which basically boil down to it’s effect on when an issue is detected.

Runtime dependency injections, like [codegangsta/inject](https://github.com/codegangsta/inject)<sup>21</sup> resort to using reflection from the internal `reflect` package, to inspect objects or functions written in the Go way above, and invoke functions that provide results for individual parameters. As an example, injection would look like:

```
1 injector := inject.New()
2 injector.Map(context.TODO()) // literal type
3 injector.Map(func() *sqlx.DB { return new(sqlx.DB) }) // object factory
4 injector.Invoke(server.New)
```

The idea is, that you declare an object which holds either concrete values or functions that return values of a required type and then figure out with reflection which values should be passed to `server.New`, hidden behind `injector.Invoke()` call.

Obviously this has a few issues like error handling (it's likely that a database connection can produce a timeout error on connection, we can't handle it here and thus we can only resort to a panic call), and ultimately a runtime error when a value or construction for a particular type is not defined.

Runtime errors are an unwanted side effect when reflection is used in such a way. The structure of the application doesn't have any kind of type hinting that would suggest to the go compiler that a dependency isn't satisfied, and possibly dangerous errors could bleed into production because of this.

Google has also realized this problem, and because of it has designed their own DI framework, [google/wire](https://github.com/google/wire)<sup>22</sup>. The intent of the project is to provide a compile-time dependency injection, which could error out in case a dependency isn't declared or provided at build time.

## Wire and dependency providers

Wire relies on code generation and reflection to analyze the source code and produce injection code that is equal to what you would have to write, in order to fill the dependencies required by your service.

The concepts introduced by Wire are called:

- The Injector - the code that introduces dependencies to your object,
- The Provider - the code that provides an instance of a particular type

An injector is any function that returns a particular type, like in our case, the database connection handle, `*sqlx.DB`, or in the case of our currently only RPC client, a `stats.StatsService` interface. Each of these can be returned with an error, or without one. If returned with an error, wire will generate the supporting code for error checking and bubble up the error from the generated Injector.

Let's create `db/connect.go` with our provider for a database connection:

---

<sup>21</sup><https://github.com/codegangsta/inject>

<sup>22</sup><https://github.com/google/wire>

```

1 package db
2
3 import (
4     "errors"
5     "os"
6
7     "github.com/jmoiron/sqlx"
8 )
9
10 func Connect() (*sqlx.DB, error) {
11     dsn := os.Getenv("DB_DSN")
12     driver := os.Getenv("DB_DRIVER")
13     if dsn == "" {
14         return nil, errors.New("DB_DSN not provided")
15     }
16     if driver == "" {
17         driver = "mysql"
18     }
19     return sqlx.Connect(driver, dsn)
20 }

```

In this case, the `db.Connect` function is a provider for our database connection. As for our clients, we will resort to code generation again, to produce another `ProviderSet`. Add the following to your Makefile under your templates: target (the main one):

```

1 @./templates/client_wire.go.sh

```

And then create `templates/client_wire.go.sh` with the following contents:

```

1 #!/bin/bash
2 cd $(dirname $(dirname $(readlink -f $0)))
3
4 ## list all services
5 schemas=$(ls -d rpc/* | xargs -n1 basename)
6
7 function render_wire {
8     echo "package client"
9     echo
10    echo "import ("
11    echo -e "\t\"github.com/google/wire\""
12    echo
13    for schema in $schemas; do

```



```

14     echo -e "\t\"${MODULE}/client/${schema}\""
15 done
16 echo ")"
17 echo
18 echo "var Inject = wire.NewSet("
19 for schema in $schemas; do
20     echo -e "\t${schema}.New,"
21 done
22 echo ")"
23 }
24
25 render_wire > client/wire.go
26 echo "~ client/wire.go"

```

For each service we have declared, a new provider is registered in `wire.NewSet()` and made available over `client.Inject`. In our current case, this file is generated:

```

1 package client
2
3 import (
4     "github.com/google/wire"
5
6     "github.com/titpetric/microservice/client/stats"
7 )
8
9 var Inject = wire.NewSet(
10     stats.New,
11 )

```

As we would add a new service, the import for the service would be added, and a new provider would be added to the `wire.NewSet` call. As such, new services immediately become available to all existing services, all we need is to add the appropriate type into the `Server{}` structure and it will get filled with the generated injector.

As these injectors are stand-alone, we can create a global inject package, which binds them together in a single `ProviderSet` (`inject/inject.go`):

```

1 package inject
2
3 import (
4     "github.com/google/wire"
5
6     "github.com/titpetric/microservice/client"
7     "github.com/titpetric/microservice/db"
8 )
9
10 var Inject = wire.NewSet(db.Connect, client.Inject)

```

And from here on we can move towards generating the Injector with wire.

## Wire and dependency Injector

We have scaffolded all the requirements for wire, to be able to analyze the code and generate the injector that will fill out dependencies for our services. First, we need to install wire in our build/Dockerfile:

```

1 +# Install google wire for DI
2 +RUN go get -u github.com/google/wire/cmd/wire

```

Rebuild and push the dockerfile to your registry, and you can start using wire from Drone. As wire will create our new service constructor, remove `New()` from `templates/server_server.go.tpl` and add `*sqlx.DB` as a dependency by declaring it inside the `Server{}` struct:

```

1 package ${SERVICE}
2
3 import (
4     "context"
5
6     "github.com/jmoiron/sqlx"
7
8     "${MODULE}/rpc/${SERVICE}"
9 )
10
11 type Server struct {
12     db *sqlx.DB
13 }
14
15 var _ ${SERVICE}.${SERVICE_CAMEL}Service = &Server{}

```

This will ensure that there isn't a conflict between the old constructor, and the new constructor created with Wire. We will now create an injector definition in our service, under `server/stats/wire.go`, by using a template for our code generator (`templates/server_wire.go.tpl`):

```
1  //+build wireinject
2
3  package ${SERVICE}
4
5  import (
6      "context"
7
8      "github.com/google/wire"
9
10     "${MODULE}/inject"
11 )
12
13 func New(ctx context.Context) (*Server, error) {
14     wire.Build(
15         inject.Inject,
16         wire.Struct(new(Server), "*"),
17     )
18     return nil, nil
19 }
```

As you see from the template, the file begins with `//+build wireinject`, which excludes this file from our build process. Wire uses this file to load and analyze the source files that it references, to figure out which dependencies are defined.

Make sure that you invoke the following line in your Makefile `templates.%` target to generate the required `wire.go` file for each existing service:

```
1  @envsubst < templates/server_wire.go.tpl > server/${SERVICE}/wire.go
```

After we build the docker image for our build environment, all we need to do is run `wire ./...` on our code base. We will add this to our build steps in drone:

```
1 - name: build
2   image: titpetric/microservice-build
3   pull: always
4   commands:
5     - make tidy
6     - wire ./...
7     - make build
```

Running `wire ./...` produces our `wire_gen.go` files:

```
1 # wire ./...
2 wire: github.com/titpetric/microservice/server/stats: wrote server/stats/wire_gen.go
```

And if we inspect the file itself, we can see it's written just as well if we would write it by hand. Each of the required dependencies by our server is filled out, and the others are omitted. This in our case means that we get our DB handle, but as we aren't using any RPC clients they are omitted until we add them to the `Server` struct.

```
1 // Code generated by Wire. DO NOT EDIT.
2
3 //go:generate wire
4 //+build !wireinject
5
6 package stats
7
8 import (
9     "context"
10    "github.com/titpetric/microservice/db"
11 )
12
13 // Injectors from wire.go:
14
15 func New(ctx context.Context) (*Server, error) {
16     sqlxDB, err := db.Connect()
17     if err != nil {
18         return nil, err
19     }
20     server := &Server{
21         db: sqlxDB,
22     }
23     return server, nil
24 }
```

The providers that return errors also have error handling in this function. As the errors are preserved and returned, we can use them outside of our service constructor - in our case, we issue a `log.Fatal` in `main()` and exit.

With wire set up, adding a new dependency into our service is trivial, we just need to add a new field into the `Server{}` struct. In case we want to omit a field from being filled out with wire, we can tag it with `wire:"-"`. We future proofed our service layer in such a way that we can add new dependencies into the main `inject` package, and they will be picked up by all services at the same time.

There are some caveats with wire, notably that you can't pass a `wire.ProviderSet` as the argument for an `Injector`. This is because wire relies on parsing the AST of your packages source files in order to generate provider invocations. In the sense of a public API which could be imported from a third party package, wire has no way to know on the code generation step, which providers are defined inside a `ProviderSet`.

There are probably other notable exceptions where wire might fail. In the sense of something you can pick up and randomly throw at an object, it's definitely not as robust as writing individual server constructors by hand.

As a possibility, instead of a generic `ProviderSet`, we could have a concrete `Providers` interface, that must satisfy all the fields declared on `Server{}`. The interface, as it is a concrete type, can cross package boundaries and the individual provider implementations (or a singleton of them) could be provided as an external package. Currently wire doesn't really take the concept into it's logical conclusion, and is probably too complex for a DI framework. But, as we don't really have a better option and won't start to write our own, this is what we have and we can work with it.

# Docker: Building images with security in mind

When it comes to running our microservices in production, we need to build docker images. Security is unfortunately an afterthought, so let's try to figure out what we can do to increase security so it's better than most of the stuff out there.

## Configuring Makefile targets

We will start by adding Makefile rules to enable us to build and push docker images.

```
1  # docker image build
2
3  IMAGE_PREFIX := titpetric/service-
4
5  docker: $(shell ls -d cmd/* | sed -e 's/cmd\\//docker./')
6      @echo OK.
7
8  docker.%: export SERVICE = $(shell basename $*)
9  docker.%:
10     @figlet $(SERVICE)
11     docker build --rm --no-cache -t $(IMAGE_PREFIX)$(SERVICE) --build-arg service_name\
12     =$(SERVICE) -f docker/serve/Dockerfile .
13
14  # docker image push
15
16  push: $(shell ls -d cmd/* | sed -e 's/cmd\\//push./')
17      @echo OK.
18
19  push.%: export SERVICE = $(shell basename $*)
20  push.%:
21     @figlet $(SERVICE)
22     docker push $(IMAGE_PREFIX)$(SERVICE)
```

There's nothing magical about this target, it's the same principle which we are already using for our code generation, and service building. To use it, you can just run `make docker` to build the images, and `make push` to push the images to docker hub or your registry, based on the `IMAGE_PREFIX` value.

## The basic Dockerfile image

Let's create our basic service Dockerfile under `docker/serve/Dockerfile`. We will start with some reasonable defaults and then try to improve the security of the built images even further.

```
1 FROM alpine:latest
2
3 ARG service_name
4 ENV service_name=$service_name
5
6 WORKDIR /app
7
8 ENV TZ Europe/Ljubljana
9 RUN apk --no-cache add ca-certificates tzdata && ln -snf /usr/share/zoneinfo/$TZ /etc\
10 c/localtime && echo $TZ > /etc/timezone
11
12 COPY /build/${service_name}-linux-amd64 /app/service
13
14 RUN adduser -S www-data -u 1000
15 USER www-data
16
17 EXPOSE 3000
18
19 ENTRYPOINT ["/app/service"]
```

We are actually building a fully featured distro image on alpine here. We add the few required packages that actually enable our service to function on some common baseline:

- `ca-certificates` adds the SSL certificates required for issuing HTTPS requests,
- `tzdata` configures the timezone (usually optional if your apps will be built on UTC)

The `tzdata` package is required not only for setting a default timezone for the Docker image, but to format times based on timezone data in Go:

```
1 location, err := time.LoadLocation("Europe/Ljubljana")
2 if err != nil {
3     fmt.Println(err)
4 }
5
6 t := time.Now().In(location)
7
8 fmt.Println("Time in Ljubljana:", t.Format("02.01.2006 15:04"))
```

Security wise, we are doing two things:

- we add an unprivileged `www-data` user, so our service doesn't run as root,
- we run our service on port 3000 (a privileged user would be required to run on port 80).

The attack surface here requires that a person should first exploit our service, and then exploit the running kernel to escalate privileges from `www-data` to root, and then could use `apk` to install packages that could either do whatever inside the container, or attempt to break out of docker to own your host.

Can we improve this further? I believe we can. Two things come to mind:

- we can remove tools and binaries that enable root to have an usable container (remove `apk` to start),
- instead of relying on `alpine`, we could build our images on `scratch`, which is absolutely empty.

## Security implications of our docker image

In order to figure out all the files that are bundled in the container, we can use docker to list the files built in the image. Since the container has `find`, this can be done with:

```
1 docker run --rm --entrypoint=/usr/bin/find titpetric/service-stats / -type f > files\
2 .txt
```

Inspecting the file, we can list the root folders to figure out which are safe and unsafe:



```
1 # cat files.txt | perl -p -e 's/^(\[^\]/)+\./\1/g' | sort | uniq -c | sort -nr
2   9904 /sys
3   1362 /usr
4   1130 /proc
5    51 /etc
6     8 /lib
7     3 /sbin
8     1 /.dockeren
9     1 /bin
10    1 /app
```

We can immediately ignore the `/sys` and `/proc` folders, since these are coming from our docker container environment and don't include executable files. We could just inspect the executables found in the `PATH` environment:

```
1 # docker run -it --rm --entrypoint=/bin/sh titpetric/service-stats -c set | grep PATH
2 PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
```

Since all the executable files are contained in either the `bin` or `sbin` folders, we can list them particularly just by looking for `bin/` with `grep`:

```
1 # cat files.txt | grep bin/
2 /usr/bin/c_rehash
3 /usr/bin/posixtz
4 /usr/bin/iconv
5 /usr/bin/getent
6 /usr/bin/getconf
7 /usr/bin/ssl_client
8 /usr/bin/scanelf
9 /usr/sbin/zdump
10 /usr/sbin/update-ca-certificates
11 /usr/sbin/zic
12 /bin/busybox
13 /sbin/apk
14 /sbin/ldconfig
15 /sbin/mkmtndirs
```

Now, the simpler attack surface for privilege escalation needs something that's called a `setuid` bit set on the executable. This means that an unprivileged user like `www-data` can run something as root. You can inspect the binaries by invoking `find` with `-perm -4000` as the parameter. On our host, we'll end up a list like this:

```
1 # find / -perm -4000
2 /usr/lib/openssh/ssh-keysign
3 /usr/lib/eject/dmccrypt-get-device
4 /usr/lib/dbus-1.0/dbus-daemon-launch-helper
5 /usr/bin/chfn
6 /usr/bin/sudo
7 /usr/bin/gpasswd
8 /usr/bin/newgrp
9 /usr/bin/passwd
10 /usr/bin/chsh
11 /bin/su
12 /bin/mount
13 /bin/umount
```

Alpine doesn't come with any `setuid` executables (at least with our current package selection), which means the only way to exploit the running container would be to attack the kernel. From here, the attack surface can still theoretically be limited to the running container, meaning an attacker could run anything in the container with elevated root privileges. The other way would be to break out of the `cgroup` of the running process, and effectively attack the host. We can defend against the first scenario, just by cleaning up the executables in the built image. Let's just do that:

```
1 # delete all the bundled binaries on standard PATH locations
2 RUN rm -rf /bin /sbin /usr/bin /usr/sbin /usr/local/bin /usr/local/sbin
```

By cleaning these up, we can be sure that the attacker can't just run `/bin/busybox` under a privileged process and end up with a shell to your container. There are other ways to improve the security of your container, and ultimately the security issue might be in your app and the libraries that are compiled in, so we can't be sure that there is absolutely no way to exploit it, but at least we came pretty damn close.

## Possible improvements

There are two notable projects that deal with container security which you might want to look at.

- [Clair](https://github.com/quay/clair)<sup>23</sup> - a static analyzer for common vulnerabilities,
- [docker-slim](https://github.com/docker-slim/docker-slim)<sup>24</sup> - merge image layers and remove unused files

With Clair, the intent of the project is to continuously scan your images for newly published vulnerabilities. It will not solve them, but it will let you know if you need to upgrade or remove some of the packages that are contained in your container.

---

<sup>23</sup><https://github.com/quay/clair>

<sup>24</sup><https://github.com/docker-slim/docker-slim>

With `docker-slim`, the project uses static analysis for doing what we did above by hand. We know that our application doesn't rely on anything under `bin/` or `sbin/` folders, so we could safely delete them without impacting our service. Docker slim goes further than that, and not only removes binaries, but rebuilds the complete image without referencing the base image, and deleting everything that may be unused, possibly even stripping debug symbols from your app, and ending up with a tiny image which goal is both optimized for size and security. With it, your final build image would be similar to what you would get if you started with `FROM scratch`.

Another approach would be to use a multi stage Dockerfile, which first installs everything we need in a builder image, and then copies the required files into an image built from scratch. What we would probably need is just the `/etc` folder, or a subset of it. The exercise is left to the reader.

# Go: implementing a microservice

We should move towards writing actual code for our service. We are writing a simple statistics collection service for tracking page views. We are going to create a database schema for collecting page views and improve things along the way.

## The database schema

To refresh, we will be implementing our Push() RPC call:

```
1 service StatsService {
2     rpc Push(PushRequest) returns (PushResponse);
3 }
4
5 message PushRequest {
6     string property = 1;
7     uint32 section = 2;
8     uint32 id = 3;
9 }
10
11 message PushResponse {}
```

I've created a database schema which serves my use case for logging the request. In addition to the request data for the RPC, I added fields `remote_ip`, and `stamp` to track the user making the page view and the time the page view was logged. I also defined an `id` `uint64` primary key, which I'll fill out with a K-sortable [Sonyflake ID](https://github.com/sony/sonyflake)<sup>25</sup>.

Name	Type	Key	Comment
id	bigint(20) unsigned	PRI	Tracking ID
property	varchar(32)		Property name (human readable, a-z)
property_section	int(11) unsigned		Property Section ID
property_id	int(11) unsigned		Property Item ID
remote_ip	varchar(255)		Remote IP from user making request
stamp	datetime		Timestamp of request

As you can see, our docs generator is already paying for itself. This is our migration, with the

---

<sup>25</sup><https://github.com/sony/sonyflake>

comments stripped out to cut down verbosity:

```

1 CREATE TABLE `incoming` (
2   `id` bigint(20) unsigned
3   `property` varchar(32)
4   `property_section` int(11) unsigned
5   `property_id` int(11) unsigned
6   `remote_ip` varchar(255)
7   `stamp` datetime
8   PRIMARY KEY (`id`)
9 );
10
11 CREATE TABLE `incoming_proc` LIKE `incoming`;

```

We are automatically creating a copy of incoming into incoming\_proc. This has to do with processing caveats - renaming tables in mysql is an atomic operation, meaning we can issue the following queries and replace the tables between each other:

```

1 RENAME TABLE incoming TO incoming_old, incoming_proc TO incoming, incoming_old TO in\
2 coming_proc;

```

The rename table statement would replace the tables between each other. This is to separate our reads from our writes - one table is always getting written to, the other is getting read and aggregated into other tables, grouped by various time intervals (hourly, daily, monthly,...).

In the previous version of this service, the table schema had an ID field which was an AUTO\_INCREMENT. In InnoDB, an AUTO\_INCREMENT column triggers a special table-level lock, which blocks other transactions so that they receive consecutive primary key values. We aren't resorting to auto incrementing fields so we already eliminated a possible source of performance issues.

After adding the migration, we run make, so all our assets are generated.

## Improving the microservice environment

We do a few improvements of our code generators in order to increase consistency. Since wire generates wire\_gen.go, we rename our types\_db.go to types\_gen.go so we can be consistent. We also notice that the fields from our migration result in a few oddly named struct fields:

- Id
- PropertyId
- RemoteIp

When we were writing our camelizer, we didn't consider that we want to stay compliant with common Go naming styles, e.g. "JSON" instead of "Json". Let's quickly improve that part of our code generator, by importing a package that handles common initialisms.

```

1 package main
2
3 import (
4     "github.com/serenize/snaker"
5 )
6
7 func camel(input string) string {
8     return snaker.SnakeToCamel(input)
9 }

```

And now we can use ID, PropertyID and RemoteIP respectively. The package [serenize/snaker](https://github.com/serenize/snaker)<sup>26</sup> lists common initialisms which it sources from [golang/lint#lint.go](https://github.com/golang/lint#lint.go)<sup>27</sup>. We can even add golint to our CI jobs? Let's do it. Add the following at the end of `docker/build/Dockerfile`:

```

1 # Install golint
2 RUN go get -u golang.org/x/lint/golint

```

Rebuild the image by running `make` and `make push` so it can be used with Drone, and then let's add a Makefile target and Drone CI step to lint our code:

```

1 lint:
2     golint -set_exit_code ./...

```

And our `.drone.yml`:

```

1 --- a/.drone.yml
2 +++ b/.drone.yml
3 @@ -29,6 +29,7 @@ steps:
4     commands:
5         - make tidy
6         - wire ./...
7 +     - make lint
8         - make build

```

I fixed the issues that showed up, mostly about having comments for exported fields, like:

---

<sup>26</sup><https://github.com/serenize/snaker>

<sup>27</sup><https://github.com/golang/lint/blob/fdd1cda4f05fd1fd86124f0ef9ce31a0b72c8448/lint.go#L770>

```

1  --- a/templates/client_wire.go.sh
2  +++ b/templates/client_wire.go.sh
3  @@ -15,6 +15,7 @@ function render_wire {
4      done
5      echo ")"
6      echo
7  +    echo "// Inject produces a wire.ProviderSet with our RPC clients"
8      echo "var Inject = wire.NewSet("
9      for schema in $schemas; do
10         echo -e "\t${schema}.New,"

```

And after adding all the possible comments to our code generated code and a few stylistic fixes around our db package, we are ready to implement our service up to spec.

## Implementing Push

As mentioned before, we will need an additional dependency - as we don't have an AUTO\_INCREMENT column, we will use a Sonyflake ID generator to provide us with k-sortable uint64 values. In our inject package, define the following provider:

```

1  import (
2      "os"
3      "strconv"
4
5      "github.com/sony/sonyflake"
6  )
7
8  // Sonyflake produces a sonyflake ID generator dependency
9  func Sonyflake() *sonyflake.Sonyflake {
10     var serverID uint16
11     if val, err := strconv.ParseInt(os.Getenv("SERVER_ID"), 10, 16); err == nil {
12         serverID = uint16(val)
13     }
14     if serverID > 0 {
15         return sonyflake.NewSonyflake(sonyflake.Settings{
16             MachineID: func() (uint16, error) {
17                 return serverID, nil
18             },
19         })
20     }
21     return sonyflake.NewSonyflake(sonyflake.Settings{})
22 }

```

Sonyflake has 16 bits reserved for a Machine ID. By default, Sonyflake calculates the Machine ID from the bottom 16 bits of a valid LAN address from your container. This may prove problematic if you run your services on multiple stand alone hosts, as the individual networks may provision the same IP to multiple instances of your service. I added in support for a `SERVER_ID` environment variable to be prepared for this.

Update the ProviderSet in the inject package:

```

1 // Inject is the main ProviderSet for wire
2 var Inject = wire.NewSet(
3     db.Connect,
4     Sonyflake,
5     client.Inject,
6 )

```

and include `*sonyflake.Sonyflake` in the `Server{}` definition for the stats service. Wire will regenerate the `wire_gen.go` file, where the sonyflake generator will be created.

Implementing our Push RPC with all the scaffolding which we have now is pretty trivial:

```

1 package stats
2
3 import (
4     "context"
5     "fmt"
6     "strings"
7     "time"
8
9     "github.com/titpetric/microservice/rpc/stats"
10 )
11
12 // Push a record to the incoming log table
13 func (svc *Server) Push(ctx context.Context, r *stats.PushRequest) (*stats.PushResponse, error) {
14     var err error
15     row := Incoming{}
16
17     row.ID, err = svc.sonyflake.NextID()
18     if err != nil {
19         return nil, err
20     }
21
22     row.Property = r.Property
23     row.PropertySection = r.Section

```



```

25     row.PropertyID = r.Id
26     row.RemoteIP = "127.0.0.1"
27     row.SetStamp(time.Now())
28
29     fields := strings.Join(IncomingFields, ",")
30     named := ":" + strings.Join(IncomingFields, ",:")
31
32     query := fmt.Sprintf("insert into %s (%s) values (%s)", IncomingTable, fields, nam\
33 ed)
34     _, err = svc.db.NamedExecContext(ctx, query, row)
35     return nil, err
36 }

```

This is an initial implementation. There are two notable things to take care of. Since the Stamp field was mapped into a `*time.Time`, we can't assign the output of `time.New()` `time.Time` directly to it. Since I don't particularly like wrapping a `time.NewPtr()` `*time.Time` in every service, I modified the code generator to add a setter for this field type:

```

1  ...
2  for _, table := range tables {
3      fields := []string{}
4      primary := []string{}
5      setters := []string{}
6  ...
7      if columnType == "*time.Time" {
8          setters = append(setters, []string{
9              fmt.Sprintf("// Set%s sets %s which requires a *time.Time", columnName, co\
10 lumnName),
11              fmt.Sprintf("func (s %s) Set%s(t time.Time) { s.%s = &t }", tableName, co\
12 lumnName, columnName),
13          }...)
14      }
15  ...
16  for _, v := range setters {
17      fmt.Fprintln(buf, v)
18  }
19  if len(setters) > 0 {
20      fmt.Fprintln(buf)
21  }
22
23  ...

```

And the other thing is still a TODO item - I want to log the Remote IP of the request. Now, since we're

dealing with a Twirp implementation, we know that we have a `*http.Request` as the entrypoint. This is the first real divergence between the implementation of a Twirp or a gRPC service.

What we need to do to get the IP here is to wrap our twirp handler in our own `http.Handler`, which gets the information about the IP from the `*http.Request`, and updates the context so we can get the value for this field from the context passed to our handler.

1. `http.Request` provides a `Context()` `context.Context` function to get the request,
2. `http.Request` provides a `WithContext(context.Context) *Request` to build a new `*Request`

This should be pretty simple. Let's see if we can do it on the first try (no cheating!). To help you out with some starting code, this is a wrapped handler:

```

1  --- a/templates/cmd_main.go.tpl
2  +++ b/templates/cmd_main.go.tpl
3  @@ -15,6 +15,12 @@ import (
4      server "${MODULE}/server/${SERVICE}"
5  )
6
7  +func wrapWithIP(h http.Handler) http.Handler {
8  +    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
9  +        h.ServeHTTP(w, r)
10 +    })
11 +}
12 +
13 func main() {
14     ctx := context.TODO()
15
16 @@ -25,5 +31,5 @@ func main() {
17
18     twirpHandler := ${SERVICE}.New${SERVICE_CAMEL}ServiceServer(srv, nil)
19
20 -    http.ListenAndServe(":3000", twirpHandler)
21 +    http.ListenAndServe(":3000", wrapWithIP(twirpHandler))
22 }
```

Now stop reading and go write your wrapper!

I said no cheating? Read the IP from a HTTP Request, set it into a context, and then come back when you have it working!

Ok, enough of that. Let's see what I did on the first try:

```

1 func wrapWithIP(h http.Handler) http.Handler {
2     return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
3         // get IP address
4         ip := func() string {
5             headers := []string{
6                 http.CanonicalHeaderKey("X-Forwarded-For"),
7                 http.CanonicalHeaderKey("X-Real-IP"),
8             }
9             for _, header := range headers {
10                 if addr := r.Header.Get(header); addr != "" {
11                     return strings.SplitN(addr, " ", 2)[0]
12                 }
13             }
14             return strings.SplitN(r.RemoteAddr, ":", 2)[0];
15         }()
16
17         ctx := r.Context()
18         ctx = context.WithValue(ctx, "ip.address", ip)
19
20         h.ServeHTTP(w, r.WithContext(ctx))
21     })
22 }

```

I implemented a wrapper that checks X-Forwarded-For and X-Real-IP headers, and returns the first relevant IP listed in there. In case none of these headers are present, I take `r.RemoteAddr`, and strip away the port number which is expected from reading the docs.

Let's modify our Push implementation to read from the context:

```

1 if remoteIP, ok := ctx.Value("ip.address").(string); ok {
2     row.RemoteIP = remoteIP
3 }

```

So, at worst - the RemoteIP field will be empty. Hopefully I didn't do too bad of a job above and we will see a real IP get logged into the database when we issue our first request to the service.

# Docker/Bash: Issuing requests against our microservice

In order to test our microservice, we require a database, running migrations, and our service itself. We're going to build a docker-compose.yml file which takes care of all of this.

## The docker-compose file

In order to use the docker-compose.yml file, first you need to run `make` && `make docker` to build the required docker images.

I started with a simple docker-compose.yml file, like this:

```
1  version: '3.4'
2
3  services:
4    stats:
5      image: titpetric/service-stats
6      restart: always
7      environment:
8        DB_DSN: "stats:stats@tcp(db:3306)/stats"
9
10   migrations:
11     image: titpetric/service-db-migrate-cli
12     command: [
13       "-db-dsn=stats:stats@tcp(db:3306)/stats",
14       "-service=stats",
15       "-real=true"
16     ]
17
18   db:
19     image: percona/percona-server:8.0.17
20     environment:
21       MYSQL_ALLOW_EMPTY_PASSWORD: "true"
22       MYSQL_USER: "stats"
23       MYSQL_DATABASE: "stats"
24       MYSQL_PASSWORD: "stats"
25     restart: always
```

Running a microservice requires some coordination between containers, which we will do by hand, because:

- migrations aren't part of our service,
- we don't have database reconnect logic yet so migrations don't wait for the database to become available

We will fix that, but for now let's do some of the manual legwork in order to make this work. Issue the following commands:

```
1 # first run the database
2 docker-compose up -d db
3 # wait for a bit so it gets provisioned
4 sleep 30
5 # run migrations
6 docker-compose run --rm migrations
7 # run our service
8 docker-compose up -d
```

Now, generally anytime you'd just do `docker-compose up -d` should bring up our service, database, and update the migrations (leaving a container behind since `docker-compose` doesn't have a `rm: true` option we could use here). This is why we should update the migrations to be run from our service with a particular command line parameter or environment flag. We'll come back to that.

If you did everything correctly, `docker-compose ps` should output something like:

	Name	Command	State	Ports
1				
2	-----			
3	microservice_db_1	/docker-entrypoint.sh mysqld	Up	3306/tcp, 33060/tcp
4	microservice_stats_1	/app/service	Up	3000/tcp

## Our first request

Now, we didn't define any exposed ports on the container, but that's not important. We can access any publicly exposed port from the container, all we need to do is find out the IP that it's using. We can do that like this:

```

1 # docker inspect microservice_stats_1 | grep IPAddress
2     "SecondaryIPAddresses": null,
3     "IPAddress": "",
4     "IPAddress": "172.24.0.3",

```

We can quickly verify by issuing a simple curl request:

```

1 # curl -s http://172.24.0.3:3000 | jq .
2 {
3     "code": "bad_route",
4     "msg": "unsupported method \"GET\" (only POST is allowed)",
5     "meta": {
6         "twirp_invalid_route": "GET /"
7     }
8 }

```

The actual endpoint for our service is available on the following link:

- /twitch/stats.StatsService/Push

Let's first start with bogus request with invalid data:

```

1 curl -s -X POST -H 'Content-Type: application/json' \
2     http://172.24.0.3:3000/twitch/stats.StatsService/Push \
3     -d '{"userID": "2"}' | jq .

```

The response we get back is:

```

1 {
2     "code": "internal",
3     "msg": "received a nil *PushResponse and nil error while calling Push. nil respons\
4 es are not supported"
5 }

```

Well, the error isn't entirely expected, but it's an easy fix. Instead of returning nil at the end of our Push function, we'll just create a PushResponse instance with new():

```

1  --- a/server/stats/server_push.go
2  +++ b/server/stats/server_push.go
3  @@ -31,5 +31,5 @@ func (svc *Server) Push(ctx context.Context, r *stats.PushRequest)\
4   (*stats.PushR
5
6       query := fmt.Sprintf("insert into %s (%s) values (%s)", IncomingTable, field\
7   s, named)
8       _, err = svc.db.NamedExecContext(ctx, query, row)
9   -   return nil, err
10  +   return new(stats.PushResponse), err
11  }

```

Re-issuing the requests returns a valid but empty response (JSON: {}). But wait, we literally sent invalid data to our RPC, how come we didn't error out? Let's revisit the structure for PushRequest:

```

1  type PushRequest struct {
2   Property string `json:"property,omitempty"`
3   Section  uint32  `json:"section,omitempty"`
4   Id       uint32  `json:"id,omitempty"`
5  }

```

Inspecting the JSON tags on the PB generated fields, we see that they have `omitempty` set. This is the reason why we don't get a JSON decoder error on the request, as none of the fields are mandatory. The encoding/json package also works in a way, where you don't need to decode every bit of the JSON structure, so our bogus user ID payload just gets ignored.

Obviously, this a job for validation. Let's add some basic checks for valid input. Add the following checks at the beginning of our Push() implementation:

```

1  validate := func() error {
2   if r.Property == "" {
3       return errors.New("Missing Property")
4   }
5   if r.Property != "news" {
6       return errors.New("Invalid Property")
7   }
8   if r.Id < 1 {
9       return errors.New("Missing ID")
10  }
11  if r.Section < 1 {
12      return errors.New("Missing Section")
13  }
14  return nil

```

```
15     }
16     if err := validate(); err != nil {
17         return nil, err
18     }
```



```

11 -12-14 11:17:18 |
12 | 279833914468466691 | | 0 | 0 | 172.24.0.1 | 2019\
13 -12-14 11:40:11 |
14 | 279835838647369731 | news | 1 | 1 | 172.24.0.1 | 2019\
15 -12-14 11:59:18 |
16 +-----+-----+-----+-----+-----+-----+-----\
17 -----+

```

It seems we're good. The REMOTE\_IP seems to be working as well, but since we have curl here, let's forge some headers and verify that too? We need to verify XFF and XRI headers respectively:

- XFF with -H "X-Forwarded-For: 8.8.8.8, 127.0.0.1"
- XRI with -X "X-Real-IP: 9.9.9.9"

```

1 +-----+-----+-----+-----+-----+-----+-----\
2 -----+
3 | id | property | property_section | property_id | remote_ip | stamp\
4 | | | | | | |
5 +-----+-----+-----+-----+-----+-----+-----\
6 -----+
7 | 279836335454289923 | news | 1 | 1 | 9.9.9.9 | 2019-\
8 12-14 12:04:14 |
9 | 279836312486281219 | news | 1 | 1 | 8.8.8.8 | 2019-\
10 12-14 12:04:00 |
11 +-----+-----+-----+-----+-----+-----+-----\
12 -----+

```

In the words of John “Hannibal” Smith: *“I love it when a plan comes together”*.

# Go: Improving our database handling

As we start to develop real services, we found out that our database handling needs some improvements. We are going to do a series of improvements of the current state by adding some new features, and restructuring some existing ones.

## Data source name - DSN

We aren't doing any kind of data source handling so far, but we need to inject some options into the DSN, like the following:

- `parseTime=true` - needed for decoding date/datetime values into a `time.Time`, instead of `[]byte`,
- `loc=Local` - set the location for `time.Time` values, [see `time.LoadLocation`](#)<sup>28</sup>,
- `collation=utf8mb4_general_ci` - set the default collation (utf8mb4 is a given these days if you want emojis)

Create `db/dsn.go`:

```
1 package db
2
3 import "strings"
4
5 func cleanDSN(dsn string) string {
6     dsn = addOptionToDSN(dsn, "?", "?")
7     dsn = addOptionToDSN(dsn, "collation=", "&collation=utf8_general_ci")
8     dsn = addOptionToDSN(dsn, "parseTime=", "&parseTime=true")
9     dsn = addOptionToDSN(dsn, "loc=", "&loc=Local")
10    dsn = strings.Replace(dsn, "?&", "?", 1)
11    return dsn
12 }
13
14 func addOptionToDSN(dsn, match, option string) string {
15     if !strings.Contains(dsn, match) {
16         dsn += option
17     }
18     return dsn
19 }
```

---

<sup>28</sup><https://golang.org/pkg/time/#LoadLocation>

The `cleanDSN` function will append default options to the `dsn`, if they aren't already provided. If you will specify `?loc=UTC` to your microservice DSN environment, that option will be used, instead of the default `loc=Local`.

We would also like to output the DSN into the logs, without outputting the credentials. We'll resort to a simple regular expression to mask the DSN username and password. Create `db/dsn_mask.go`:

```

1 package db
2
3 import "regexp"
4 )
5
6 var dsnMasker = regexp.MustCompile("(.) (? : .* ) ( . ) : ( . ) (? : .* ) ( . ) @")
7
8 func maskDSN(dsn string) string {
9     return dsnMasker.ReplaceAllString(dsn, "$1****$2:$3****$4@")
10 }
```

The regular expression takes care of masking any number of characters in the username and password part of the DSN, outputting only the first and last characters of each, with 4 asterisks in between.

## Multiple database connections

As we figured out, we want to run the migrations for our app from the app itself. In order for migrations to work, they need extended privileges, which aren't usually needed or wanted in a microservice.

A microservice usually needs permissions to select, insert, update and delete records, while a migration needs permissions to create, alter, index and possibly drop tables. The [full list of grants](https://dev.mysql.com/doc/refman/8.0/en/grant.html#grant-privileges)<sup>29</sup> is quite long, and adding them all to the account your microservice might be using is a security issue.

To sum it up - we will need to provide two DSN credentials, one for migrations and one for the microservice. By doing this we are enabling a security barrier between the service, and between the migrations which have elevated permissions.

To enable multiple database configurations, we will add option types (`db/types.go`):

---

<sup>29</sup><https://dev.mysql.com/doc/refman/8.0/en/grant.html#grant-privileges>

```

1  package db
2
3  import (
4      "context"
5      "time"
6
7      "database/sql"
8  )
9
10 type (
11     // Credentials contains DSN and Driver
12     Credentials struct {
13         DSN      string
14         Driver   string
15     }
16
17     // ConnectionOptions include common connection options
18     ConnectionOptions struct {
19         Credentials Credentials
20
21         // Connector is an optional parameter to produce our
22         // own *sql.DB, which is then wrapped in *sqlx.DB
23         Connector func(context.Context, Credentials) (*sql.DB, error)
24
25         Retries      int
26         RetryDelay   time.Duration
27         ConnectTimeout time.Duration
28     }
29 )

```

So, to connect to any database, a client needs to provide the `ConnectionOptions`{}, with an optional `Connector` that produces an `*sql.DB`. The connector is optional and will be used to instrument our sql client without having to modify or wrap the existing functions, but by replacing the driver in use. We'll come back to this later.

## Updating database connections

With the introduced `ConnectionOptions` struct, we have options to configure connection retry. The `Retries` gives us a maximum connection try count, `RetryDelay` gives us a delay duration after particular connection attempt, while the `ConnectTimeout` gives us a global cancellation timeout.

Let's modify our main `Connect` function to use the new structures, and add a `ConnectWithOptions` call that produces the same result as `Connect`:

```

1 // Connect connects to a database and produces the handle for injection
2 func Connect(ctx context.Context) (*sqlx.DB, error) {
3     options := ConnectionOptions{}
4     options.Credentials.DSN = os.Getenv("DB_DSN")
5     options.Credentials.Driver = os.Getenv("DB_DRIVER")
6     return ConnectWithOptions(ctx, options)
7 }
8
9 // ConnectWithOptions connect to host based on ConnectionOptions{}
10 func ConnectWithOptions(ctx context.Context, options ConnectionOptions) (*sqlx.DB, error) {
11     credentials := options.Credentials
12     if credentials.DSN == "" {
13         return nil, errors.New("DSN not provided")
14     }
15     if credentials.Driver == "" {
16         credentials.Driver = "mysql"
17     }
18     credentials.DSN = cleanDSN(credentials.DSN)
19     if options.Connector != nil {
20         handle, err := options.Connector(ctx, credentials)
21         if err == nil {
22             return sqlx.NewDb(handle, credentials.Driver), nil
23         }
24         return nil, errors.WithStack(err)
25     }
26     return sqlx.ConnectContext(ctx, credentials.Driver, credentials.DSN)
27 }
28

```

Notable changes:

- we now take a `context.Context`, so we can support cancellation (CTRL+C) from `main()`,
- DSN validation was moved into `ConnectWithOptions()`
- We implemented the connector code to wrap a `*sql.DB` into `*sqlx.DB`

Re-running wire picks up our new `Connect` signature:

```

1  --- a/server/stats/wire_gen.go
2  +++ b/server/stats/wire_gen.go
3  @@ -14,7 +14,7 @@ import (
4   // Injectors from wire.go:
5
6   func New(ctx context.Context) (*Server, error) {
7   -     sqlxDB, err := db.Connect()
8   +     sqlxDB, err := db.Connect(ctx)
9       if err != nil {
10         return nil, err
11     }

```

We can now move to implement connection retrying.

## Database connection retry

Let's create a `db/connector.go` file with our retry logic:

```

1  package db
2
3  import (
4   "context"
5   "log"
6   "time"
7
8   "github.com/jmoiron/sqlx"
9   "github.com/pkg/errors"
10 )
11
12 // ConnectWithRetry uses retry options set in ConnectionOptions{}
13 func ConnectWithRetry(ctx context.Context, options ConnectionOptions) (db *sqlx.DB, \
14 err error) {
15     dsn := maskDSN(options.Credentials.DSN)
16
17     // by default, retry for 5 minutes, 5 seconds between retries
18     if options.Retries == 0 && options.ConnectTimeout.Seconds() == 0 {
19         options.ConnectTimeout = 5 * time.Minute
20         options.RetryDelay = 5 * time.Second
21     }
22
23     connErrCh := make(chan error, 1)
24     defer close(connErrCh)

```

```

25
26 log.Println("connecting to database", dsn)
27
28 go func() {
29     try := 0
30     for {
31         try++
32         if options.Retries > 0 && options.Retries <= try {
33             err = errors.Errorf("could not connect, dsn=%s, tries=%d", dsn, try)
34             break
35         }
36
37         db, err = ConnectWithOptions(ctx, options)
38         if err != nil {
39             log.Printf("can't connect, dsn=%s, err=%s, try=%d", dsn, err, try)
40
41             select {
42                 case <-ctx.Done():
43                     break
44                 case <-time.After(options.RetryDelay):
45                     continue
46             }
47         }
48         break
49     }
50     connErrCh <- err
51 }()
52
53 select {
54 case err = <-connErrCh:
55     break
56 case <-time.After(options.ConnectTimeout):
57     return nil, errors.Errorf("db connect timed out, dsn=%s", dsn)
58 case <-ctx.Done():
59     return nil, errors.Errorf("db connection cancelled, dsn=%s", dsn)
60 }
61
62 return
63 }

```

I tried to be brief here, but let's go over the notable parts:

- a goroutine is created to loop and retry connecting to the database,

- an error channel is created to receive the final error from the goroutine,
- we are using named returns to keep the SLOC down somewhat,
- a final select statement listens for:
  - context cancellation (e.g. signal from outside)
  - overall connect timeout option as an event,
  - the exit from the goroutine (connErrCh)

Now, I'm pretty sure there's an error in there. Let me ask you a question: What happens to the goroutine if the global ConnectTimeout is reached?

## Migration improvements

As a helper/utility, we want to run migrations from our service, if we provide an argument like `-migrations true` (by default this would be false). We also want to use the connection retry for the migrations, as well as our service which will run behind it.

Let's improve `cmd/db-migrate-cli/main.go` first:

```

1  -         db struct {
2  -             DSN    string
3  -             Driver string
4  -         }
5  +         db      db.ConnectionOptions
6  -         real    bool
7  -         service string
8  -     }
9  -     flag.StringVar(&config.db.Driver, "db-driver", "mysql", "Database driver")
10 -     flag.StringVar(&config.db.DSN, "db-dsn", "", "DSN for database connection")
11 +     flag.StringVar(&config.db.Credentials.Driver, "db-driver", "mysql", "Database driver")
12 +     flag.StringVar(&config.db.Credentials.DSN, "db-dsn", "", "DSN for database connection")

```

We replace the already existing structure with DSN and Driver name with the expected `db.ConnectionOptions`. We can now pass this to a `ConnectWithRetry` method along with a context:



```

1  ctx := context.Background()
2
3  switch config.real {
4  case true:
5      handle, err := db.ConnectWithRetry(ctx, config.db)
6      if err != nil {
7          log.Fatalf("Error connecting to database: %v", err)
8      }
9      if err := db.Run(config.service, handle); err != nil {
10         log.Fatalf("An error occurred: %v", err)
11     }

```

There's no changes to the API apart from that the connection retry is now built in.

To make this easier to test, we provision a specific migrations user and database with the definition for the database service in `drone.yml`:

```

1  services:
2  - name: mysql-test
3    pull: always
4    image: percona/percona-server:8.0.17
5    ports:
6    - 3306
7    environment:
8      MYSQL_ROOT_PASSWORD: default
9      MYSQL_USER: migrations
10     MYSQL_PASSWORD: migrations
11     MYSQL_DATABASE: migrations

```

After running the migrations for a given service, we need to empty this database. We don't want to error out if two services decide to have the same table name like `log` or something generic. Let's add a `-drop` option to the `db-schema-cli`. At the same time we also add `service`, since currently we only have `schema` and our code generation would produce package migration, which would be incorrect.

1. to the config variable, add `service` string and `drop` bool,
2. invoke `flag.StringVar` and `flag.BoolVar` respectively,
3. when checking `config.output == ""`, add `' && !config.drop'`,
4. after listing all the tables in the database, add the following snippet:

```

1 // Drop all tables in schema
2 if config.drop {
3     for _, table := range tables {
4         query := "DROP TABLE " + table.Name
5         log.Println(query)
6         if _, err := handle.Exec(query); err != nil {
7             log.Fatal(err)
8         }
9     }
10    return
11 }

```

And finally, update renderGo and renderMarkdown parameter config.schema to config.service:

```

1 -             if err := renderGo(config.output, config.schema, tables); err != nil\
2 {
3 +             if err := renderGo(config.output, config.service, tables); err != ni\
4 l {
5 ...
6 -             if err := renderMarkdown(config.output, config.schema, tables); err \
7 != nil {
8 +             if err := renderMarkdown(config.output, config.service, tables); err\
9 != nil {

```

After cleaning up our Makefile and drone.yml file, we have the following:

```

1 migrate.%: export SERVICE = $*
2 migrate.%: DSN = "migrations:migrations@tcp(mysql-test:3306)/migrations"
3 migrate.%:
4     ./build/db-migrate-cli-linux-amd64 -service $(SERVICE) -db-dsn $(DSN) -real=true
5     ./build/db-migrate-cli-linux-amd64 -service $(SERVICE) -db-dsn $(DSN) -real=true
6     @find -name types_gen.go -delete
7     ./build/db-schema-cli-linux-amd64 -service $(SERVICE) -schema migrations -db-dsn $\
8 (DSN) -format go -output server/$(SERVICE)
9     ./build/db-schema-cli-linux-amd64 -service $(SERVICE) -schema migrations -db-dsn $\
10 (DSN) -format markdown -output docs/schema/$(SERVICE)
11     ./build/db-schema-cli-linux-amd64 -schema migrations -db-dsn $(DSN) -drop=true

```

And .drone.yml respectively:

```

1  steps:
2  - name: codegen
3    image: titpetric/microservice-build
4    pull: always
5    commands:
6      - make rpc
7      - make templates
8      - make build-cli
9      - make migrate
10
11 - name: build
12   image: titpetric/microservice-build
13   pull: always
14   commands:
15     - make tidy
16     - wire ./...
17     - make lint
18     - make build

```

The changes have allowed us to remove the following hack in the drone.yml config:

```

1  - "bash -c 'while : ; do sleep 1 ; $(cat < /dev/null > /dev/tcp/mysql-test/3306\
2  ) && break ; done' 2>/dev/null"

```

In fact, we removed the complete drone migration step in the cleanup. Since the schema is already provisioned with the database service, we eliminated the need to highjack the percona image for the database client, or installing it in our build image.

All we need to do now is to verify everything works as expected by running make:

```

1  ...
2  [codegen:27] + make migrate
3  [codegen:28] ./build/db-migrate-cli-linux-amd64 -service stats -db-dsn "migrations:m\
4  igrations@tcp(mysql-test:3306)/migrations" -real=true
5  [codegen:29] 2019/12/15 13:04:00 connecting to database m****s:m****s@tcp(mysql-test\
6  :3306)/migrations
7  [codegen:30] 2019/12/15 13:04:00 can't connect, dsn=m****s:m****s@tcp(mysql-test:330\
8  6)/migrations, err=dial tcp 192.168.96.2:3306: connect: connection refused, try=1
9  [codegen:31] 2019/12/15 13:04:05 can't connect, dsn=m****s:m****s@tcp(mysql-test:330\
10  6)/migrations, err=dial tcp 192.168.96.2:3306: connect: connection refused, try=2
11 [codegen:32] 2019/12/15 13:04:10 can't connect, dsn=m****s:m****s@tcp(mysql-test:330\
12  6)/migrations, err=dial tcp 192.168.96.2:3306: connect: connection refused, try=3
13 [codegen:33] 2019/12/15 13:04:15 Running migrations from migrations.sql
14 ...

```

And finally, we need to add an option to our service main.go files that will enable us to run the migrations on service startup, simplifying our docker-compose.yml. Let's also take care of context cancellation via signals, using the small but effective [SentimensRG/sigctx](https://github.com/sentimensRG/sigctx)<sup>30</sup> package:

```

1  --- a/templates/cmd_main.go.tpl
2  +++ b/templates/cmd_main.go.tpl
3  @@ -4,20 +4,41 @@ package main
4  // generator and template: templates/cmd_main.go.tpl
5
6  import (
7  +     "flag"
8       "log"
9  -     "context"
10
11     "net/http"
12
13     _ "github.com/go-sql-driver/mysql"
14  +     "github.com/SentimensRG/sigctx"
15
16  +     "${MODULE}/db"
17     "${MODULE}/internal"
18     "${MODULE}/rpc/${SERVICE}"
19     server "${MODULE}/server/${SERVICE}"
20 )
21
22 func main() {
23 -     ctx := context.TODO()
24  +     var config struct {
25  +         migrate bool
26  +         migrateDB db.ConnectionOptions
27  +     }
28  +     flag.StringVar(&config.migrateDB.Credentials.Driver, "migrate-db-driver", "m\
29  ysql", "Migrations: Database driver")
30  +     flag.StringVar(&config.migrateDB.Credentials.DSN, "migrate-db-dsn", "", "Mig\
31  rations: DSN for database connection")
32  +     flag.BoolVar(&config.migrate, "migrate", false, "Run migrations?")
33  +     flag.Parse()
34  +
35  +     ctx := sigctx.New()
36  +
37  +     if config.migrate {
38  +         handle, err := db.ConnectWithRetry(ctx, config.migrateDB)

```

---

<sup>30</sup><https://github.com/sentimensRG/sigctx>

```

39 +         if err != nil {
40 +             log.Fatalf("Error connecting to database: %v", err)
41 +         }
42 +         if err := db.Run("${SERVICE}", handle); err != nil {
43 +             log.Fatalf("An error occurred: %v", err)
44 +         }
45 +     }

```

We also add a bit of output before we start listening for requests:

```

1  log.Println("Starting service on port :3000")
2  http.ListenAndServe(":3000", internal.WrapWithIP(twirpHandler))

```

## Re-testing the bundled migrations

We can now simplify our `docker-compose.yml` config to run migrations directly from our service. Automatic migrations aren't something you want to run in production, but it simplifies our testing/development a lot, since the database contents are usually thrown away.

```

1  version: '3.4'
2
3  services:
4    stats:
5      image: titpetric/service-stats
6      restart: always
7      environment:
8        DB_DSN: "stats:stats@tcp(db:3306)/stats"
9      command: [
10         "-migrate-db-dsn=stats:stats@tcp(db:3306)/stats",
11         "-migrate"
12       ]
13
14  db:
15    image: percona/percona-server:8.0.17
16    environment:
17      MYSQL_ALLOW_EMPTY_PASSWORD: "true"
18      MYSQL_USER: "stats"
19      MYSQL_DATABASE: "stats"
20      MYSQL_PASSWORD: "stats"
21    restart: always

```

Don't forget to run `make && make docker`, and after you can run `docker-compose up` and watch the service getting set up without manual migration steps.

# Go: Instrumenting the HTTP service with Elastic APM

Elastic APM is an application performance monitoring product Elastic, the makers of Elasticsearch, and most notably the ELK stack (Elasticsearch, Logstash, Kibana). APM is another plug and play product of theirs, that hooks up to your existing ELK installation and provides endpoints for application agents to receive performance metrics data.

Elastic APM provides a Go agent, which we will use to instrument various parts of our code. The instrumentation will help us not only with performance metrics, but with debugging and optimization of our applications as well.

## Wrapping our existing handler

We already wrapped the HTTP handler with our own, that adds IP information into the request context. APM works in the same way, providing their own wrapper for a `http.Handler`. Since we can reasonably assume that we will need other wrappers in the future, we will create another function under `internal/wrap.go` to keep `main()` cleaner:

```
1  import (  
2      "strings"  
3  
4      "net/http"  
5  
6      "go.elastic.co/apm/module/apmhttp"  
7  )  
8  
9  // WrapAll wraps a http.Handler with all needed handlers for our service  
10 func WrapAll(h http.Handler) http.Handler {  
11     h = WrapWithIP(h)  
12     h = apmhttp.Wrap(h)  
13     return h  
14 }  
15  
16 ...
```

The code imports the APM module with the HTTP handler wrapper. There are also routing framework specific wrappers available, notably: `httprouter`, `gorilla`, `cgi`, `gin`, `echo`, `beego`,... The full list is quite extensive, you can review them on [elastic/apm-agent-go](https://github.com/elastic/apm-agent-go)<sup>31</sup>.

So, as we have `WrapAll` defined now, we can fix our generator for `main.go`:

```

1  --- a/templates/cmd_main.go.tpl
2  +++ b/templates/cmd_main.go.tpl
3  @@ -48,5 +48,5 @@ func main() {
4      twirpHandler := ${SERVICE}.New${SERVICE_CAMEL}ServiceServer(srv, nil)
5
6      log.Println("Starting service on port :3000")
7  -    http.ListenAndServe(":3000", internal.WrapWithIP(twirpHandler))
8  +    http.ListenAndServe(":3000", internal.WrapAll(twirpHandler))
9  }
```

Run `make` to rebuild your app with HTTP server instrumentation.

The wrapper takes care of creating what is called a “Transaction”. A transaction in our case consists of the incoming request to our handler, all the way until the completion of that request. For each request, various data is logged to Elastic APM, the most obvious one of which is the request duration.

## Logging errors too

Any request coming to your service may error out. When creating `twirpHandler`, we omit the second parameter to the `NewXXXServiceServer` call, a parameter of `*twirp.ServerHooks` type. This structure defines various hooks implemented by the Twirp RPC server handler. We are especially interested in the `Error` hook:

```

1  type ServerHooks struct {
2      // Error hook is called when an error occurs while handling a request. The
3      // Error is passed as argument to the hook.
4      Error func(context.Context, Error) context.Context
5  }
```

Particularly, we want to log the error with another Elastic APM API call, `CaptureError`. Let’s prepare a function that will construct an instance of `*twirp.ServerHooks` in our internal package with a defined error logger, under `internal/twirp.go`:

---

<sup>31</sup><https://github.com/elastic/apm-agent-go/tree/master/module>

```

1 package inject
2
3 import (
4     "context"
5
6     "github.com/twitchtv/twirp"
7     "go.elastic.co/apm"
8 )
9
10 func NewServerHooks() *twirp.ServerHooks {
11     return &twirp.ServerHooks{
12         Error: func(ctx context.Context, err twirp.Error) context.Context {
13             apm.CaptureError(ctx, err).Send()
14             return ctx
15         },
16     }
17 }

```

The function `apm.CaptureError` takes the second parameter of the type error. As the interface `twirp.Error` is a superset of the standard error type, it's already compatible with this parameter.

In our `cmd_main.tpl` we just replace the `nil` parameter with `internal.NewServerHooks()`.

```

1 -     twirpHandler := ${SERVICE}.New${SERVICE_CAMEL}ServiceServer(srv, nil)
2 +     twirpHandler := ${SERVICE}.New${SERVICE_CAMEL}ServiceServer(srv, internal.NewServerHooks())
3

```

## Setting up ELK

There's an utility docker image that provides a bundled ELK installation:

- [sebp/elk](https://hub.docker.com/r/sebp/elk)<sup>32</sup> docker hub image,
- [spujadas/elk-docker](https://github.com/spujadas/elk-docker)<sup>33</sup> github repository

The container supports ELK up to version 7.4.0. It should be easy enough to add it into our development `docker-compose.yml` file. Append the following under `services`:

---

<sup>32</sup><https://hub.docker.com/r/sebp/elk/>

<sup>33</sup><https://github.com/spujadas/elk-docker>



```
1 elk:
2   image: sebp/elk
```

And run the following in order:

```
1 sysctl vm.max_map_count=512000
2 docker-compose up -d
```

It possibly takes a few minutes in order for ELK to be reachable. When it completes, you can open `http://localhost:5601` in your browser, and you'll find yourself on the ELK console. Now we need to add APM.

## Setting up APM

Turns out, we don't really need much for APM, just a small config file that tells it where the elasticsearch instance is where it wants to send data to. Let's create `docker/elk`, and inside `apm-server.yml`:

```
1 apm-server:
2   host: "0.0.0.0:8200"
3
4   output.elasticsearch:
5     hosts: ["elk:9200"]
```

And the Dockerfile (we could avoid this with a volume mount, but it's trivial):

```
1 FROM docker.elastic.co/apm/apm-server:7.4.0
2
3 USER root
4 COPY apm-server.yml /usr/share/apm-server/apm-server.yml
5 RUN chown root:apm-server /usr/share/apm-server/apm-server.yml
6
7 USER apm-server
```

And finally, add APM to our `docker-compose.yml` file:

```
1 apm:
2   build: docker/apm
```

Re-run `docker-compose up -d` to start the APM service.

## Configuring our sender

We have configured ELK and APM, now we need to configure our service to send data to APM. This is done with two environment variables, which we can add to our service, just after DB\_DSN:

```
1     environment:
2         DB_DSN: "stats:stats@tcp(db:3306)/stats"
3 +     ELASTIC_APM_SERVICE_NAME: "stats"
4 +     ELASTIC_APM_SERVER_URL: "http://apm:8200"
5     command: [
```

Run `docker-compose up -d` to reload the changed services.

Now we can re-run the curl request which we used for testing:

```
1  #!/bin/bash
2  payload='{
3      "property": "news",
4      "section": 1,
5      "id": 1
6  }'
7
8  curl -s -X POST -H 'Content-Type: application/json' \
9      -H "X-Real-IP: 9.9.9.9" \
10     http://172.22.0.5:3000/twirp/stats.StatsService/Push \
11     -d "$payload" | jq .
```

Be sure to update the IP of the service by running `docker inspect microservice_stats_1` and finding the actual IP address of the service. Run the curl request a couple of times, so it will generate/feed some data into APM.

If you don't see any requests to APM from Kibana, make sure your service build is up to date (run `make docker.stats`), and then run `docker-compose up -d` again to recreate it.

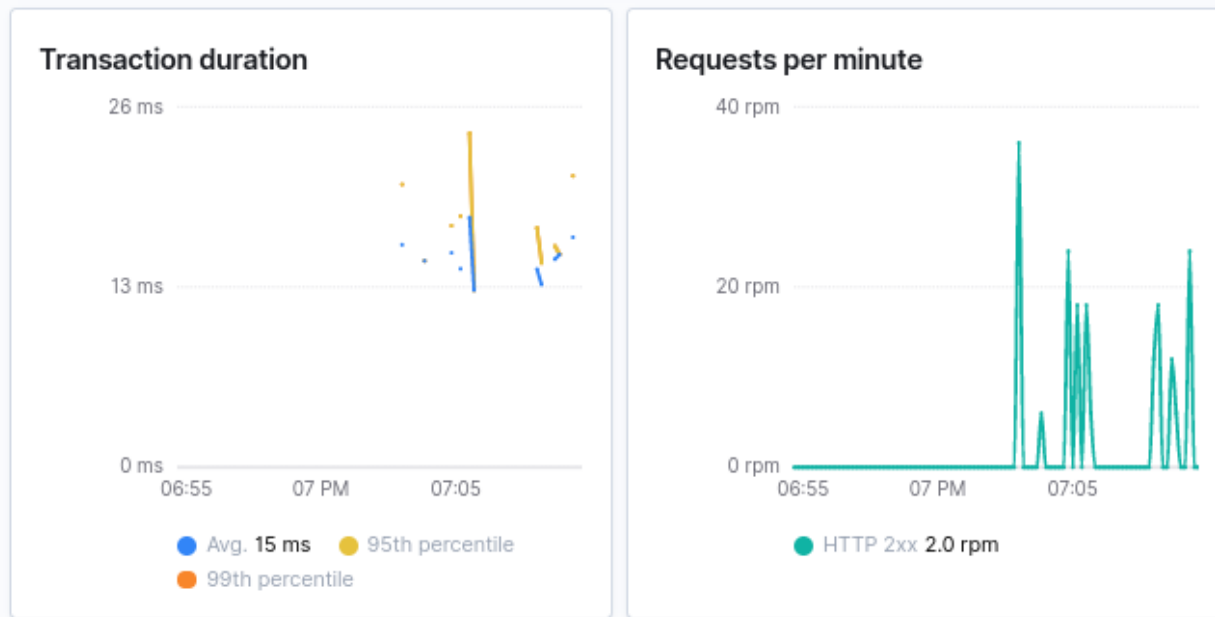
## Reviewing ELK data

As we started collecting ELK data for our service, it's a good time to review a few key features. After opening up the APM menu from the left side-bar, we get a listing of every service that we are logging, along with basic metrics like average response time, transactions/minute, and errors/minute.

Services		Traces				
Filters						
HOST						
AGENT NAME						
		Name ↑	Environment	Agent	Avg. r...	Trans...
		stats		go	16 ms	< 0.1 tpm
						0 err.

APM: The list of registered services

After this, we can drill down into our service. Set an interval for the last 15 minutes, so you get some usable graphs on the next screen. The graphs show the overall request rate and durations for your service, and at the end of the page - individual transactions.



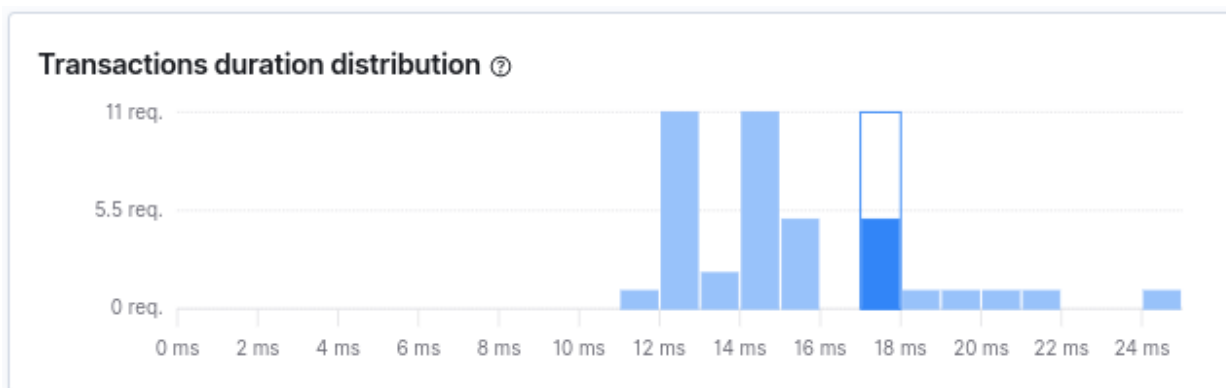
APM: Service details

At the bottom of the screen, individual transactions are listed:

Transactions				
Name	Avg. dura...	95th perc...	Trans. pe...	Impact ↓
<a href="#">POST /twirp/stats.StatsService/Push</a>	15 ms	21 ms	2.0 tpm	

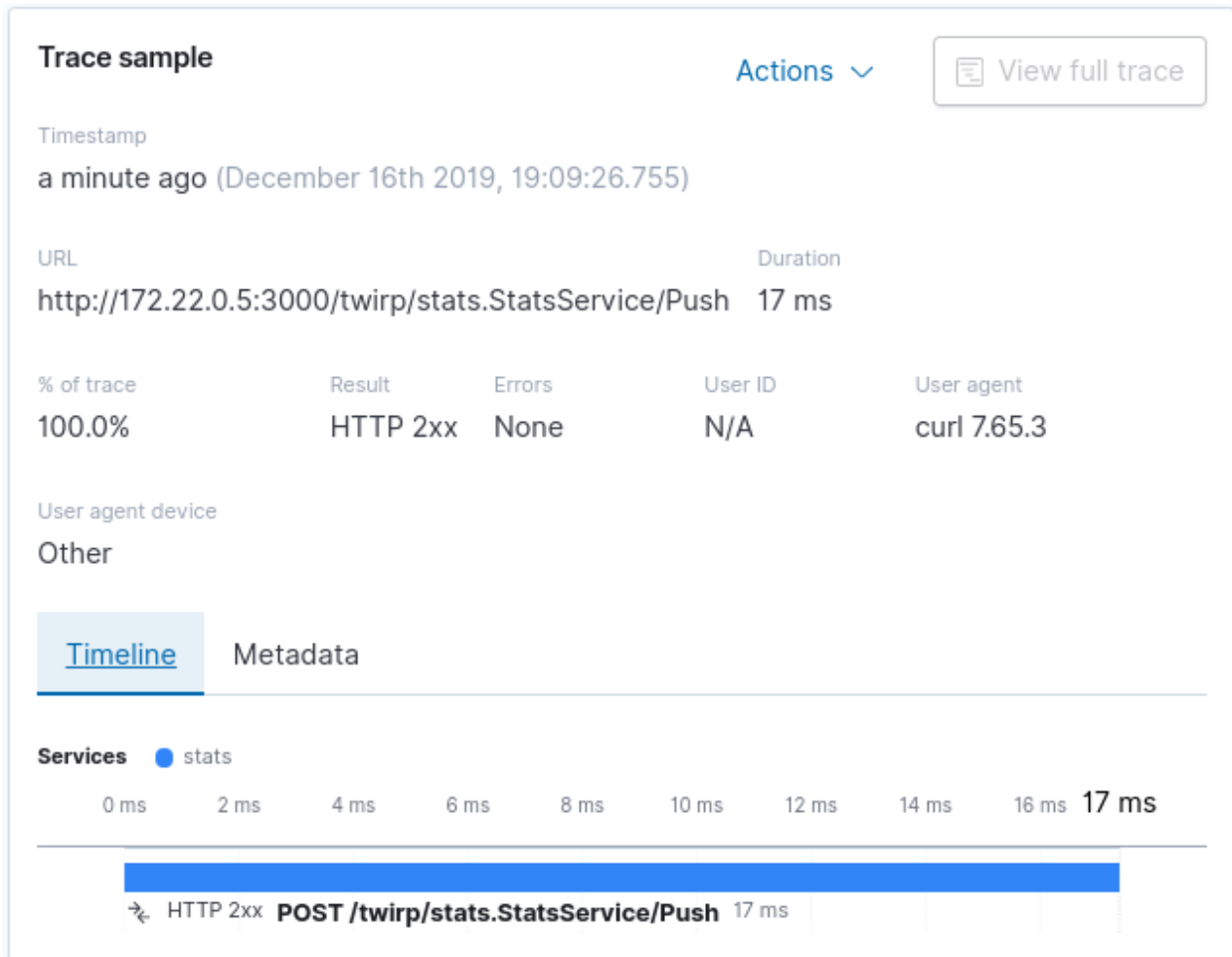
APM: Transaction list

Drilling down into transactions gives us observability beyond the simple access log line. APM aggregates our requests and provides a distribution graph, so you can figure out which requests took an extraordinary amount of time. After all, we should be optimizing the 95th or 99th percentile, and not just the overall average case. The low hanging fruit usually comes from edge cases which aren't optimized well.



APM: Transaction details

There are two important things to note on the bottom of the screen:



APM: Transaction details

- Timeline
- Metadata

The timeline currently contains only our transaction, but we will enrich this with spans. For example, a span would be an SQL query, and the time it took to execute it. We can then subdivide the transaction into implementation details and answer questions like:

- How many queries against the database did the request perform?
- What was the longest database query inside the RPC?
- How many HTTP requests did we make?
- What was the longest HTTP request inside the RPC?
- How long does connecting to a database take?

The metadata contains the complete HTTP request data, information about the host/container where our service is running, which Go version our service was built with, which arguments our service was run with, and we can extend this data with our own custom fields.

Our next step will be to instrument our database, so we can see our database queries and things like connect latency. With that, we will have enough data about our service, so that we can optimize and improve it.

# Go: Instrumenting the Database client with Elastic APM

After we set up Elastic APM to log our request transaction, the following thing we wish to instrument are the SQL queries going to our database endpoint. Elastic APM provides instrumentation that wraps the database/sql driver, which produces an `*sql.DB`.

## Extending DB connection

We already planned to produce a `*sqlx.DB` for this eventuality with the `Connector` field function in `db.ConnectionOptions`:

```
1 // Connector is an optional parameter to produce our
2 // own *sql.DB, which is then wrapped in *sqlx.DB
3 Connector func(context.Context, Credentials) (*sql.DB, error)
```

We can now just modify the `Connect()` function in the DB package, to extend it with an APM connector.

```
1 // Connect connects to a database and produces the handle for injection
2 func Connect(ctx context.Context) (*sqlx.DB, error) {
3     options := ConnectionOptions{
4         Connector: func(ctx context.Context, credentials Credentials) (*sql.DB, error) {
5             db, err := apmsql.Open(credentials.Driver, credentials.DSN)
6             if err != nil {
7                 return nil, err
8             }
9             if err = db.PingContext(ctx); err != nil {
10                 db.Close()
11                 return nil, err
12             }
13             return db, nil
14         },
15     }
16     options.Credentials.DSN = os.Getenv("DB_DSN")
17     options.Credentials.Driver = os.Getenv("DB_DRIVER")
18     return ConnectWithRetry(ctx, options)
19 }
```

There are about three notable parts to this change. First, by default we were using `sqlx.Connect` to create the database handle and issue a Ping request and error out. As this is an `sqlx` addon, we need to re-implement some functionality here by calling `Open`, followed with `PingContext`.

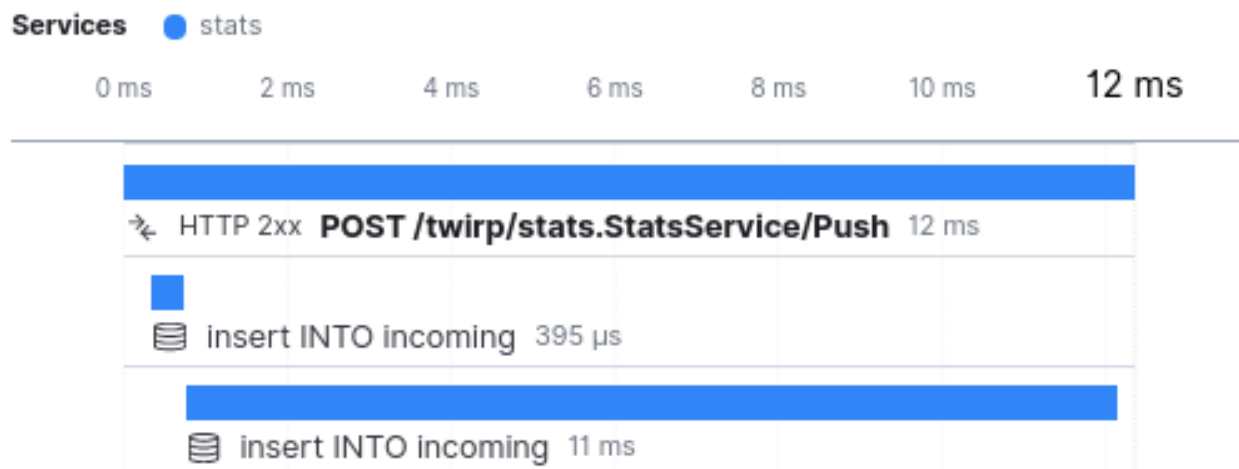
What `apmsql` does under the scenes is to produce a `sql.Driver` interface, that wraps the original driver for the drivers you're already familiar with. The Elastic APM Go Agent provides the following packages to register popular SQL drivers:

- [go.elastic.co/apm/module/apmsql/pq](https://go.elastic.co/apm/module/apmsql/pq) ([github.com/lib/pq](https://github.com/lib/pq))
- [go.elastic.co/apm/module/apmsql/mysql](https://go.elastic.co/apm/module/apmsql/mysql) ([github.com/go-sql-driver/mysql](https://github.com/go-sql-driver/mysql))
- [go.elastic.co/apm/module/apmsql/sqlite3](https://go.elastic.co/apm/module/apmsql/sqlite3) ([github.com/matttn/go-sqlite3](https://github.com/matttn/go-sqlite3))

## Verifying it's working

Each SQL query issued will produce what is called a "Span". Like the transactions, the APM client sends the query metadata and duration, and nests it under the main request transaction. This way you can see particularly which queries have executed within a given request.

Let's rebuild our service with `make` and `make docker`, and run our development stack with `docker-compose up -d`. Let's re-run some requests with `curl`, and navigate to the new transactions in the APM interface.





APM: Listing query SPANs

The query shows up twice, the first one is the prepare of the statement, and the second one is the exec of the statement. We can click the span and get a detailed view.



Span details

 [View span in Discover](#)



Service		Transaction	
stats		POST /twirp/stats.StatsService/Push	
Name		Duration	
insert INTO incoming		11 ms	
% of transaction	Type	Subtype	Action
92.2%	DB	MySQL	exec

Database statement

```
insert into incoming (id,property,property_section,property_id,remote_ip,stamp)
values (?, ?, ?, ?, ?, ?)
```

APM: The query details

The particular query which ran against the database is logged in it's normalized form. It does not include the actual parameters of the query being inserted.

## Stack Trace   Labels

---

### > 8 library frames

```
named_context.go in NamedExecContext at line 131
sqlx_context.go in (*DB).NamedExecContext at line 134
server_push.go in (*Server).Push at line 53
stats.twirp.go in (*statsServiceServer).servePushJSON.func1 at line 216
stats.twirp.go in (*statsServiceServer).servePushJSON at line 217
stats.twirp.go in (*statsServiceServer).servePush at line 186
stats.twirp.go in (*statsServiceServer).ServeHTTP at line 168
wrap.go in WrapWithIP.func1 at line 38
```

### > 5 library frames

#### APM: The query stack trace

And finally, we can see the stack trace of the query being executed. We can know the path of the application call, all the way to the calling of the database function. It's useful to have this but I fear that perhaps it's not the most quick, as stack traces in Go are notably slow, and need to be avoided if you're working on performance.

APM makes provisions for tuning this - a stack trace is only recorded if the span duration is longer than 5ms, and you can tune that with an environment variable:

1 ELASTIC\_APM\_SPAN\_FRAMES\_MIN\_DURATION=5ms

Check out other possible configuration options for the APM agent on the [agent configuration page](https://www.elastic.co/guide/en/apm/agent/go/current/configuration.html)<sup>34</sup>.

Everything in your service stays the same, now there is just the APM agent under the hood, sending query metrics to Elastic APM as we wanted. We can use this data to analyze the query performance and improve our application significantly.

Like, I can't believe that query took 12ms. We have not tuned the database AT ALL.

---

<sup>34</sup><https://www.elastic.co/guide/en/apm/agent/go/current/configuration.html>

# Go: Stress testing our service

As we implemented our service that adds pageviews to the database, we realize an unfortunate circumstance. Each insert query takes about 12ms, which means our request rate, at least per single CPU core, is poor. We're talking about 83 req/s poor.

But as we know that scaling isn't linear and you shouldn't judge a single request, we need to perform some stress testing to give us a better idea of where we are.

Now, the best practices of stress testing suggest that you need isolation. In our case this means that we would need to have at least 3 servers running, one with our stress testing tool, another with our service, and another with the database, and possibly a fourth server with APM. This way the benchmarks won't be tainted.

Now, considering I'm doing this on a low-end-ish thin client machine, we can be sure that my stress tests here will be tainted. I'm running everything on a single machine, along with a full featured desktop, so the measurements will be skewed, but hopefully they will be good enough for some basic evaluation.

## Setting up our stress test

What we need to do first is figure out how to stress test some requests to our service. We need a tool that supports POST requests with custom payloads, concurrency, and something that will spit out a total request rate at the end.

A short search leads us to wrk, and there seems to be a modern up to date docker image with wrk available ([skandyla/wrk](https://hub.docker.com/r/skandyla/wrk/)<sup>35</sup>), so let's try to use it straight from docker. In order to craft a POST request, a lua script for it needs to be created. Let's create test.lua:

```
1 wrk.method = "POST"
2 wrk.headers['Content-Type'] = "application/json"
3 wrk.body    = '{"property":"news","section":1,"id":1}'
```

And let's create a test.sh, that also takes care of getting our service container IP:

---

<sup>35</sup><https://hub.docker.com/r/skandyla/wrk/>

```

1  #!/bin/bash
2  function get_stats_ip {
3      docker inspect \
4          microservice_stats_1 | \
5      jq -r '.[0].NetworkSettings.Networks["microservice_default"].IPAddress'
6  }
7
8  url="http://$(get_stats_ip):3000/twirp/stats.StatsService/Push"
9
10 docker run --rm --net=host -it -v $PWD:/data skandyla/wrk -d15s -t4 -c400 -s test.lua \
11 a $url

```

The command will run tests for 15 seconds, 4 threads, 400 connections, and run the script test.lua for each request. Running ./test.sh against our service gives us this:

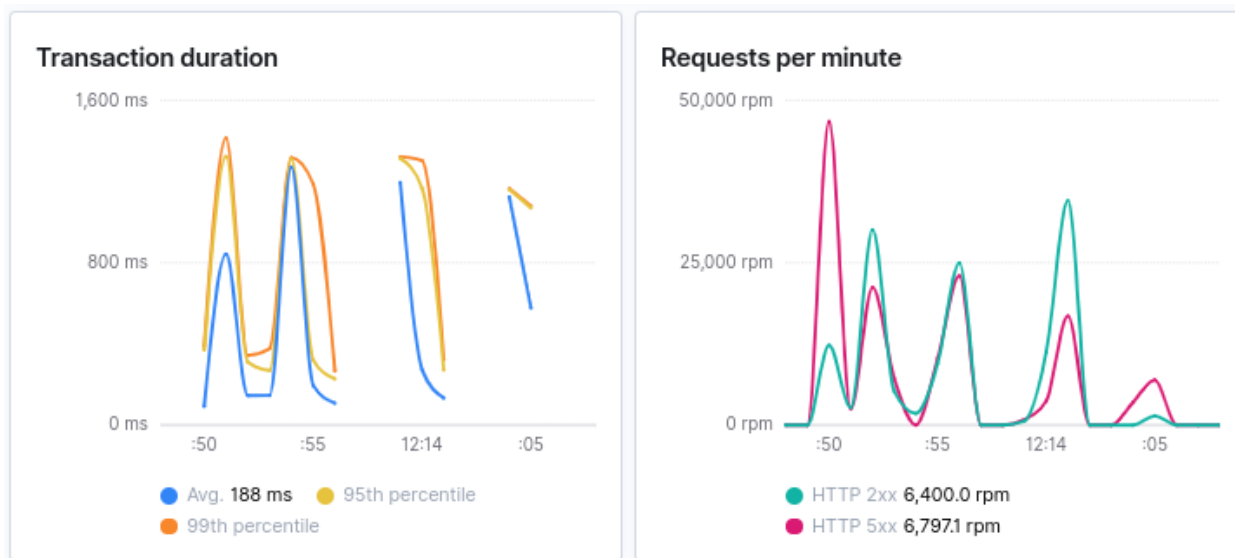
```

1  Running 15s test @ http://172.22.0.5:3000/twirp/stats.StatsService/Push
2      4 threads and 400 connections
3      Thread Stats   Avg      Stdev     Max    +/-  Stdev
4      Latency    224.79ms  256.55ms   1.56s    88.49%
5      Req/Sec    579.91    191.59    1.26k    69.63%
6      34493 requests in 15.04s, 5.54MB read
7      Socket errors: connect 0, read 0, write 0, timeout 90
8      Non-2xx or 3xx responses: 17834
9      Requests/sec:   2293.06
10     Transfer/sec:   377.23KB

```

It seems that stress testing our service resulted in 17834 errors out of 34493 requests, along with 90 timeout errors. That is an error rate of about 52%. Since we are using APM, we can check what kind of errors are logged and start fixing the service.

The APM graphs confirm the error rate:



The errors tab also gives us some insight into the errors:

### Errors

Group ID	Error message and culprit
3f22f	twirp error internal: Error 1040: Too many connections NewServerHooks.func1
e52cc	twirp error internal: context canceled NewServerHooks.func1

## Resolving detected errors

The Error 1040: Too many connections is tunable both on the MySQL side, and the Go side. On the Go side, we have the following options to tune MySQL usage on our DB instance:

- `SetMaxOpenConns()` - Set a maximum number of connections the service will hold open,
- `SetMaxIdleConns()` - Sets the most idle connections we can have open at any time,
- `SetConnMaxLifetime()` - Sets the time how much the connection can stay open for

All of these options tune our database usage. By default, there is no limit to the number of connections that can be opened. If we set a limit with `SetMaxOpenConns`, a connection must be freed up, so it can be reused. Generally you want this limit to be high, because you don't want to wait on connections too much if the pool is too small.

We want to simulate real life behaviour of the application to estimate the number of connections that work for our case. A stress test isn't real life behaviour, so let's just blindly set it to something highly unreasonable. Since we already set 400 connections as our target for the stress test, let's double that on the back-end for our SQL connections.

By default, `SetMaxIdleConns` only allows 2 idle connections, meaning that there would be a lot of churn as soon as clients go idle. We don't want to throw away idle connections as they die down, or at least not so fast, so we will set this value to the same as `SetMaxOpenConns`. This doesn't mean that the connection won't be closed (there's also MySQL timeout for that).

And finally, `SetConnMaxLifetime` is not set by default, meaning any connection that is opened and isn't reaped with `SetMaxIdleConns` can be reused within the duration set here.

Connection clean up operations run every second, so we will also reduce stress on the cleanup, if we can hold the connections open for a longer time. This one is up to you, but generally unless there's a good technical reason for setting a max collection lifetime, like a ProxySQL load balancer in front of your MySQL cluster, you probably don't need to change this.

Open up `db/connect.go` and replace `ConnectWithOptions`:

```

1 // ConnectWithOptions connect to host based on ConnectionOptions{}
2 func ConnectWithOptions(ctx context.Context, options ConnectionOptions) (*sqlx.DB, error) {
3     credentials := options.Credentials
4     if credentials.DSN == "" {
5         return nil, errors.New("DSN not provided")
6     }
7     if credentials.Driver == "" {
8         credentials.Driver = "mysql"
9     }
10    credentials.DSN = cleanDSN(credentials.DSN)
11
12    connect := func() (*sqlx.DB, error) {
13        if options.Connector != nil {
14            handle, err := options.Connector(ctx, credentials)
15            if err == nil {
16                return sqlx.NewDb(handle, credentials.Driver), nil
17            }
18            return nil, errors.WithStack(err)
19        }
20        return sqlx.ConnectContext(ctx, credentials.Driver, credentials.DSN)
21    }
22
23    db, err := connect()
24    if err != nil {

```

```

26     return nil, err
27 }
28 db.SetMaxOpenConns(800)
29 db.SetMaxIdleConns(800)
30 return db, nil
31 }

```

And let's configure our database to have a maximum connection limit of 1000 connections. Edit `docker-compose.yml` and update the database definition to this:

```

1  db:
2    image: percona/percona-server:8.0.17
3    command: [
4      "--max_connections=1000"
5    ]
6    environment:
7      MYSQL_ALLOW_EMPTY_PASSWORD: "true"
8      MYSQL_USER: "stats"
9      MYSQL_DATABASE: "stats"
10     MYSQL_PASSWORD: "stats"
11     restart: always

```

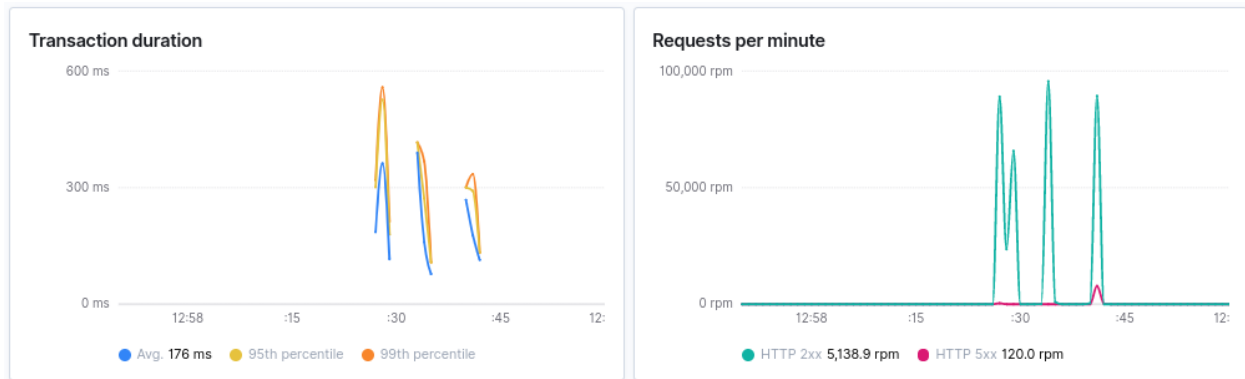
Rebuild the docker images for our service with `make && make docker` and re-run `docker-compose up -d`, and finally, let's run our benchmark again:

```

1  Running 15s test @ http://172.22.0.5:3000/twirp/stats.StatsService/Push
2    4 threads and 400 connections
3    Thread Stats   Avg      Stdev     Max    +/-  Stdev
4      Latency    185.91ms  158.08ms   1.47s    94.41%
5      Req/Sec    590.52    261.76    1.33k    62.03%
6    34813 requests in 15.05s, 3.62MB read
7  Requests/sec:   2313.32
8  Transfer/sec:   246.24KB

```

It seems we resolved all of our connection/response errors here. There are some 5xx errors still reported in APM which we need to look at.



*Note:* the request rate is strangely jagged and not sustained as we would like. I'd guess this comes from queueing on the network level, but we won't investigate it at this point.

## Removing context cancellation

We are still getting `twirp error internal: context canceled`. Now, the issue with request cancellation is that even the stack trace doesn't help much when you want to figure out where the context was canceled from, or even which context it's supposed to be.

Given this stack trace:

```

1 gocontext.go in CaptureError at line 131
2 twirp.go in NewServerHooks.func1 at line 13
3 stats.twirp.go in callError at line 795
4 stats.twirp.go in writeError at line 376
5 stats.twirp.go in (*statsServiceServer).writeError at line 138
6 stats.twirp.go in (*statsServiceServer).servePushJSON at line 220
7 stats.twirp.go in (*statsServiceServer).servePush at line 186
8 stats.twirp.go in (*statsServiceServer).ServeHTTP at line 168
9 wrap.go in WrapWithIP.func1 at line 38

```

We might assume that the error originated from twirp, given the `twirp error internal` message, but when we read the code we find this little bit:



```

1 // Non-twirp errors are wrapped as Internal (default)
2 twerr, ok := err.(twirp.Error)
3 if !ok {
4     twerr = twirp.InternalErrorWith(err)
5 }

```

So, our context canceled error came from our `Push()` function and is wrapped into a twirp internal error. And we see that the only function it could have come from is `_, err = svc.db.NamedExecContext(ctx, query, row)`.

One might think that this issue is database related, but it isn't. There's a simpler explanation: the request `Context()` function is cancellable and is cancelled as soon as the client drops the connection. The errors show up at the tail end of our stress test, so we can make an educated guess and say that the last few connections wrk makes, are closed forcefully and thus cancel the request.

Let's verify our guess with some godoc sleuthing (`http.Request.Context()`):

For incoming server requests, the context is canceled when the client's connection closes, the request is canceled (with HTTP/2), or when the `ServeHTTP` method returns.

In our case, we don't want to use context cancellation here, our `Push` request already has all the data that needs to be logged to the database, so resolving the issue means that we need to provide a context without the cancellation and let `Push()` finish regardless.

We also need to keep the context values we have, so we can log the span correctly into APM. We can't just use a `context.Background()`, we need to provide our own context implementation that keeps all the values which are already set by the APM HTTP handler wrapper.

Let's provide our context without timeout/cancellation. We need to implement two functions from the interface, while the rest can stay as is. We can use embedding, so we don't implement the complete context interface, but only the two functions required.

Navigate to the `internal` package and move `context.go` to `context_ip.go`, and create a new `context.go` file with the following contents:

```

1 package internal
2
3 import (
4     "context"
5 )
6
7 type ctxWithoutCancel struct {
8     context.Context
9 }
10

```

```

11 // Done returns nil (not cancellable)
12 func (c ctxWithoutCancel) Done() <-chan struct{} { return nil }
13
14 // Err returns nil (not cancellable)
15 func (c ctxWithoutCancel) Err() error { return nil }
16
17 // ContextWithoutCancel returns a non-cancelable ctx
18 func ContextWithoutCancel(ctx context.Context) context.Context {
19     return ctxWithoutCancel{ctx}
20 }

```

We are only partially wrapping the provided context here, so we override `Done` and `Err`, but don't touch the provided `Deadline` or `Value`. Those remaining function calls will go to the original context that we embedded here.

We can fix our `Push` function now to clear the context cancellation:

```

1 // Push a record to the incoming log table
2 func (svc *Server) Push(ctx context.Context, r *stats.PushRequest) (*stats.PushResponse, error) {
3     ctx = internal.ContextWithoutCancel(ctx)
4     ...
5 }

```

From now on, each request that reaches `Push()` will be completed, regardless if the context for the request was cancelled. Depending on if you wish to still error out on some sort of timeout value, you can add a `context.WithTimeout` call after this one.

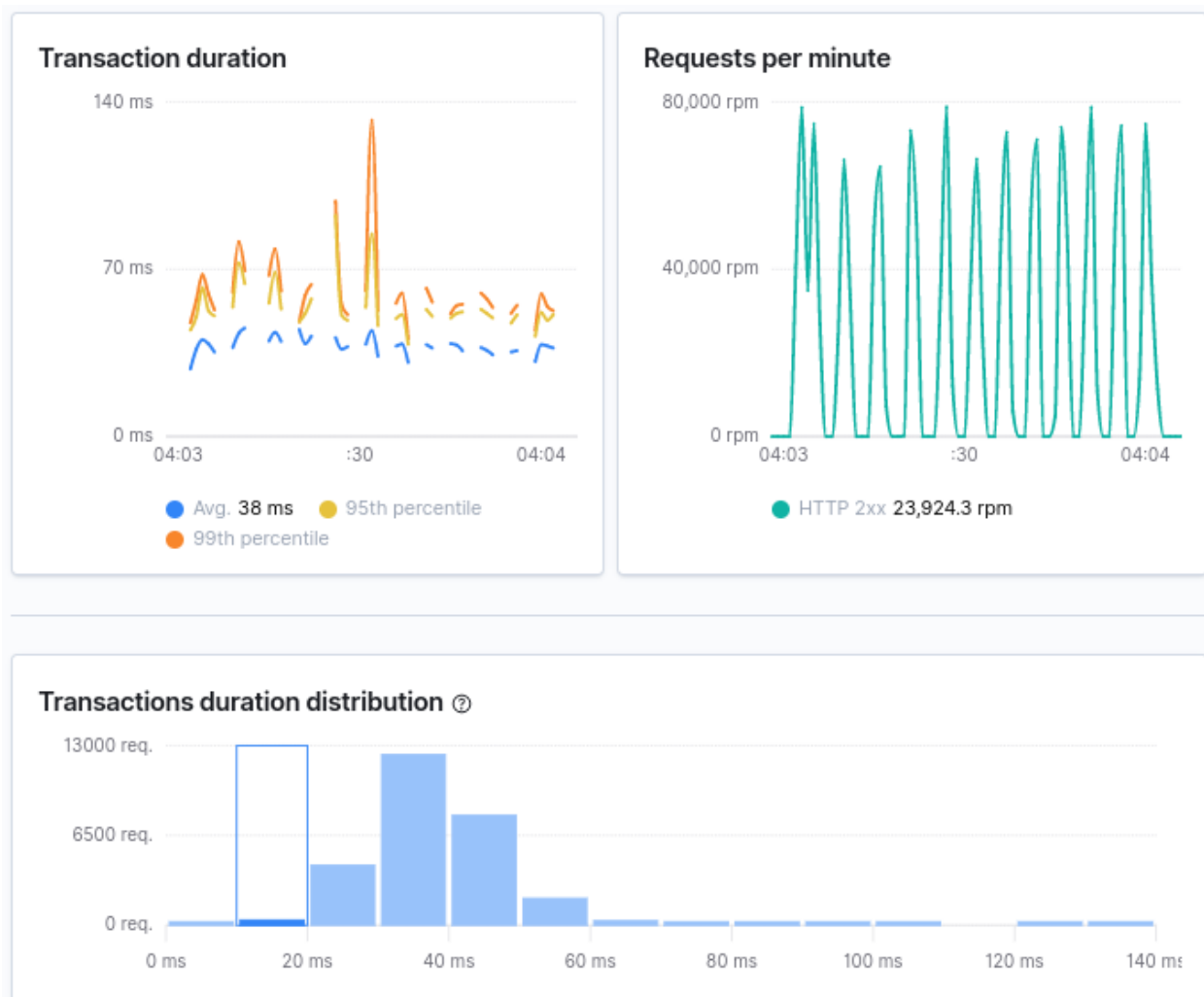
Now, let's make two changes to our `wrk` script.

- Let's use `-t60s` so we can test for a longer time period,
- Let's use `-c100` so we use a bit less connections overall

Rebuild and rerun the service, and run the test again:

```
1 Running 1m test @ http://172.22.0.5:3000/twirp/stats.StatsService/Push
2 4 threads and 100 connections
3 Thread Stats Avg Stdev Max +/- Stdev
4 Latency 47.69ms 11.86ms 143.28ms 73.10%
5 Req/Sec 527.04 85.10 800.00 73.25%
6 125930 requests in 1.00m, 13.09MB read
7 Requests/sec: 2097.16
8 Transfer/sec: 223.23KB
```

We can look at APM and see that zero errors have been logged during this stress test run. This means we have successfully resolved this issue.



## Wrapping up

You can already see the benefit of using Elastic APM. We already resolved a misconfiguration issue for our service and database in regards to database connection handling and connection limits, as well as fixing a potential bug with data loss due to behaviour of the HTTP server implementation and context cancellation.

Overall, this chapter is more of a cautionary tale what happens if you aren't super familiar with some internals of database connection handling and `http.Request` context. Elastic APM was just the tool that pointed us to our pitfalls.

# Go: Background jobs

API services, especially low latency ones, will resort to background jobs to process data faster on ingesting it, and aggregating and flushing it to backend services in an optimal way.

## Adding background jobs to our service

In our stats microservice, we issue insert queries for each request directly into the database. The database supports extended inserts, meaning we could flush thousands of rows at the same time, and actually increase throughput, since adding 10K rows to the database with a single query is significantly faster than adding 1 row 10 thousand times over multiple connections.

To take advantage of the bulk insert features of the database, we need to start a goroutine that will periodically aggregate and flush collected to the database. How often this runs is up to you, but if you care about data loss due to service restarts or shutdowns, you'll need to set a reasonably low interval.

To facilitate creating background jobs (and other possible initialization), we can extend our Server codegen to satisfy the following interface:

```
1 Init(context.Context) error
2 Shutdown()
```

The `Init` function is there to take the global cancellation context and start any background jobs that (hopefully), react to this context as well. When the service shuts down, your background jobs can stop accepting new data and flush whatever is collected to the database. The `Shutdown` function is there so the main program can wait for the background jobs flush to complete.

To sum up: `Init()` starts the job, context controls job cancellation, and `Shutdown` is there to wait for the cleanup if needed.

```

1  --- a/templates/server_server.go.tpl
2  +++ b/templates/server_server.go.tpl
3  @@ -13,4 +13,15 @@ type Server struct {
4      db *sqlx.DB
5  }
6
7  +// Init can start background jobs
8  +func (*Server) Init(_ context.Context) error {
9  +    // Any background jobs go here,
10 +    // ctx has global signal cancellation
11 +    return nil
12 +}
13 +
14 +// Shutdown is a cleanup hook after SIGTERM
15 +func (*Server) Shutdown() {
16 +}

```

Let's modify the template for main.go to call Init and Shutdown appropriately. We also start the service itself as a goroutine, and wait for context cancellation in main. As soon as the context will be cancelled, `srv.Shutdown()` will be called.

```

1  --- a/templates/cmd_main.go.tpl
2  +++ b/templates/cmd_main.go.tpl
3  @@ -43,9 +43,14 @@ func main() {
4      if err != nil {
5          log.Fatalf("Error in service.New(): %v", err)
6      }
7  -
8  +    if err := srv.Init(ctx); err != nil {
9  +        log.Fatal(err)
10 +    }
11     twirpHandler := ${SERVICE}.New${SERVICE_CAMEL}ServiceServer(srv, internal.NewServerHooks())
12
13
14     log.Println("Starting service on port :3000")
15  -    http.ListenAndServe(":3000", internal.WrapAll(twirpHandler))
16  +    go http.ListenAndServe(":3000", internal.WrapAll(twirpHandler))
17  +    <-ctx.Done()
18  +
19  +    srv.Shutdown()
20  }

```

## Do we really need Init?

Looking at the code, the `Init()` function may be redundant. Wire already takes `ctx`, we just need to see if we can actually start a goroutine from there. Let's see what wire can do in our case, let's try to modify `wire.go`:

```
1 func New(ctx context.Context) (*Server, error) {
2     s := new(Server)
3     wire.Build(
4         inject.Inject,
5         wire.Struct(new(Server), "*"),
6     )
7     if err := s.Background(); err != nil {
8         return nil, err
9     }
10    return s, nil
11 }
```

Unfortunately this results in the following error message:

wire: a call to `wire.Build` indicates that this function is an injector, but injectors must consist of only the `wire.Build` call and an optional return

Another option would be to just create a dependency for the background job. This way you would just add the field to the `Server{}` struct, and let wire invoke it and provide dependencies to it as well.

Starting up the background job would be taken care of, but we'd still need to issue the Shutdown signal here. The main questions we should answer here:

- Do we want the background Job(s?) to have their own API?
- Is the responsibility between `Server{}` and `Background{}` shared?

These are tough questions. From the perspective of a microservice, the complete implementation serves a single responsibility. From the perspective of individual components, a background service is distinctly different from the API layer, and the relationship between them may be 1:N.

If we implement multiple background jobs, the microservice `Shutdown()` function will be just a proxy for individual background jobs shutdowns. If we don't, it may be just spaghetti code, which is already a smell of shared responsibility.

If we implement the background jobs on the same level as the `Server{}`, we wouldn't need to re-specify dependencies for each job. That might seem like a good idea until you realize that it again blurs the boundary between what the microservice needs, and what the particular background job

needs. If we learned anything, we learned that we need to favour explicit declarations over implicit scope. This way it will be clear what particularly is needed by background jobs.

As we would only be adding a level of indirection with `Init`, and blurring the various responsibilities and dependencies between the background job and the microservice API, let's remove `Init()` and implement our background job with the single responsibility principle in mind.

As we make this decision, we also realize that replacing the implementation of the background job in this case is trivial - we only need to satisfy whatever interface we create for the background job API. If this was merged with `Server{}`, it would, at best, be problematic to isolate and refactor at some later point.

Remove `Init` from our generators, and let's create a background job.

## The background job

It's good to follow some naming conventions here, to ensure that our background job is designed with a single responsibility in mind. With that, I quote [Effective Go - Interface names](#)<sup>36</sup>:

By convention, one-method interfaces are named by the method name plus an `-er` suffix or similar modification to construct an agent noun: `Reader`, `Writer`, `Formatter`, `CloseNotifier` etc.

We are not creating a generic interface, but as the struct has a specific utility here, we can resort to the same naming convention for our struct. We will create a `Flusher`.

Under `server/stats/` create `flusher.go`:

```
1 package stats
2
3 import (
4     "context"
5     "log"
6     "time"
7
8     "github.com/jmoiron/sqlx"
9     "go.uber.org/atomic"
10 )
```

We're importing [go.uber.org/atomic](#)<sup>37</sup> for their wrappers around `stdlib sync/atomic`. They provide wrapped data types that make it easier to work with values in a thread-safe way.

<sup>36</sup>[https://golang.org/doc/effective\\_go.html#interface-names](https://golang.org/doc/effective_go.html#interface-names)

<sup>37</sup><https://godoc.org/go.uber.org/atomic>



```

1 // Flusher is a context-driven background data flush job
2 type Flusher struct {
3     context.Context
4     finish func()
5
6     enabled *atomic.Bool
7
8     db *sqlx.DB
9 }

```

Our `Flusher{} struct` embeds `context.Context` so we don't need to write our own `Shutdown` function, but can invoke `<-flusher.Done()` to wait for it to gracefully finish. This is a local cancellation context.

```

1 // NewFlusher creates a *Flusher
2 func NewFlusher(ctx context.Context, db *sqlx.DB) (*Flusher, error) {
3     job := &Flusher{
4         db:      db,
5         enabled: atomic.NewBool(true),
6     }
7     job.Context, job.finish = context.WithCancel(context.Background())
8     go job.run(ctx)
9     return job, nil
10 }

```

When creating the flusher, we set the dependencies, enable pushing data into the flusher (`enabled=true`), and set the cancellation context.

```

1 func (job *Flusher) run(ctx context.Context) {
2     log.Println("Started background job")
3
4     defer job.finish()
5
6     ticker := time.NewTicker(time.Second)
7
8     for {
9         select {
10            case <-ticker.C:
11                job.flush()
12                continue
13            case <-ctx.Done():
14                log.Println("Got cancel")

```

```

15     job.enabled.Store(false)
16     job.flush()
17 }
18 break
19 }
20
21 log.Println("Exiting Run")
22 }

```

The background job itself listens to the global context cancellation (`ctx.Done`), and when received disables pushing new data to the internal queue. We also create a ticker to run our flush job once a second, and use `defer` to invoke context cancellation as soon as we exit from the function.

```

1 func (job *Flusher) flush() {
2     log.Println("Background flush")
3 }

```

At this point, our flusher is a no-op, and we just print something to the log, to see that everything is working as it should. Let's adjust `server.go` to include flusher as a dependency, and create our Shutdown function:

```

1 // Server implements stats.StatsService
2 type Server struct {
3     db *sqlx.DB
4
5     sonyflake *sonyflake.Sonyflake
6     flusher  *Flusher
7 }
8
9 // Shutdown is a cleanup hook after SIGTERM
10 func (s *Server) Shutdown() {
11     <-s.flusher.Done()
12 }

```

All we need to do now is some housekeeping.

## Housekeeping

We need to add `NewFlusher` to `wire.go` as a `wire.Inject` parameter. In order to modify `wire.go` which is auto generated, we need to modify our `Makefile` and create a conditional generator script (fix under `templates.% target`):

```

1 -      @envsubst < templates/server_wire.go.tpl > server/${SERVICE}/wire.go
2 -      @echo "~ server/${SERVICE}/wire.go"
3 +      @./templates/server_wire.go.sh

```

And we create templates/server\_wire.go.sh and set it as executable:

```

1  #!/bin/bash
2  cd $(dirname $(dirname $(readlink -f $0)))
3
4  if [ -z "${SERVICE}" ]; then
5      echo "Usage: SERVICE=[name] MODULE=... $0"
6      exit 255
7  fi
8
9  OUTPUT="server/${SERVICE}/wire.go"
10
11 if [ ! -f "$OUTPUT" ]; then
12     envsubst < templates/server_wire.go.tpl > $OUTPUT
13     echo "~ $OUTPUT"
14 fi

```

With these changes, wire.go will only be written if it doesn't exist. This means we can now add NewFlusher to wire.go for our stats service.

## Verifying everything works

As you should already be used to by now, run:

- make,
- make docker,
- docker-compose up -d

This will build everything and restart your stats container locally. To inspect if everything works, issue `docker-compose stop stats` and `docker-compose logs stats` after some time.

```
1 stats_1 | 2020/01/04 23:20:13 Started background job
2 stats_1 | 2020/01/04 23:20:14 Background flush
3 stats_1 | 2020/01/04 23:20:15 Background flush
4 stats_1 | 2020/01/04 23:20:16 Background flush
5 stats_1 | 2020/01/04 23:20:17 Background flush
6 stats_1 | 2020/01/04 23:20:18 Got cancel
7 stats_1 | 2020/01/04 23:20:18 Background flush
8 stats_1 | 2020/01/04 23:20:18 Exiting Run
9 stats_1 | 2020/01/04 23:20:18 Done.
```

We now created a background job runner which we can access from our API implementation. This means that we can queue up data in the RAM and flush it from a background worker, bringing our API edge latency down.

# Go: Optimizing requests with a queue

With our background jobs interface in place, we can create an API that will queue our incoming data and flush it to the database at a defined frequency. Flushing our data in bulk will give us performance advantages.

## The Queue

A basic Queue in Go is a slice. To extend the queue, you can call `append`, which allocates a larger slice if required. Since this is not thread safe, we need to use locking to ensure that the queue is modified safely.

```
1 // Queue provides a queuing structure for Incoming{}
2 type Queue struct {
3     sync.RWMutex
4     values []*Incoming
5 }
6
7 // NewQueue creates a new *Queue instance
8 func NewQueue() *Queue {
9     return &Queue{
10         values: make([]*Incoming, 0),
11     }
12 }
```

We also need to hold several Queues to split up incoming write requests. If we only used one queue, we would have high lock contention between all the clients trying to write to the queue, as well as delays because of the flusher which needs to clear out the queue as it's written to the database.

```

1 // NewQueues creates a slice of *Queue instances
2 func NewQueues(size int) []*Queue {
3     result := make([]*Queue, size)
4     for i := 0; i < size; i++ {
5         result[i] = NewQueue()
6     }
7     return result
8 }

```

The queue needs a few functions, in order to write and flush data from the queue:

- Push(\*Incoming) error - add a new item to the queue, return error if any,
- Clear() []\*Incoming - return the current items in the queue and clear it,
- Length() int - report the current queued item count

The Push function should return an error when the queue is full. It is up to you to limit the queue size, so that your application doesn't eat up all your memory in case of traffic spikes or a delay in flushing the data to the database.

There is no reasonable limit here, so it's up to you to implement this.

```

1 // Push adds a new item to the queue
2 func (p *Queue) Push(item *Incoming) error {
3     p.Lock()
4     defer p.Unlock()
5     p.values = append(p.values, item)
6     return nil
7 }

```

Flushing the queue to the database requires us to read the current items of the queue, and clear them for any future calls to Push(). The current implementation keeps memory usage low, but allocates memory when values are pushed to the queue.

```

1 // Clear returns current queue items and clears it
2 func (p *Queue) Clear() (result []*Incoming) {
3     length := p.Length()
4
5     p.Lock()
6     defer p.Unlock()
7
8     result, p.values = p.values[:length], p.values[length:]
9     return
10 }

```

And finally - an utility function to get the size of the queue:

```
1 // Length returns the current queue size
2 func (p *Queue) Length() int {
3     p.RLock()
4     defer p.RUnlock()
5     return len(p.values)
6 }
```

In order for a queue to be functional it needs two additional things:

1. the producer: writing data to the queue,
2. the consumer: reading data from the queue

## The Producer

Our producer is easy to implement, we can just modify our service `Push()` function:

```
1 -     fields := strings.Join(IncomingFields, ",")
2 -     named := ":" + strings.Join(IncomingFields, ",:")
3 -
4 -     query := fmt.Sprintf("insert into %s (%s) values (%s)", IncomingTable, field\
5 s, named)
6 -     _, err = svc.db.NamedExecContext(ctx, query, row)
7 -     return new(stats.PushResponse), err
8 +     return pushResponseDefault, svc.flusher.Push(row)
```

Basically, instead of calling the database query directly, we now call `svc.flusher.Push(*Incoming)`, which returns a possible error in case the request couldn't be queued up.

A small change to make our API a bit more optimal is replacing `new(stats.PushResponse)` with a pre-allocated variable. Put the following snippet just before the `Push` function in `server_push.go`:

```
1 // Keep returning a single object to avoid allocations
2 var pushResponseDefault = new(stats.PushResponse)
```

When implementing the producer in our `Flusher` struct, we need some additional fields:

```

1  // queueIndex is a key for []queues
2  queueIndex *atomic.Uint32
3  // queueMask is a masking value for queueIndex -> key
4  queueMask  uint32
5  // queues hold a set of writable queues
6  queues    []*Queue

```

Go provides the [sync/atomic](https://godoc.org/sync/atomic)<sup>38</sup> package in the standard library, which provides atomic operations over some numeric types and pointers. However, we will use the [go.uber.org/atomic](https://go.uber.org/atomic)<sup>39</sup> package that provides convenience types wrapped around the standard library.

We need to update NewFlusher in order to initialize the new fields:

```

1  func NewFlusher(ctx context.Context, db *sqlx.DB) (*Flusher, error) {
2      queueSize := 1 << 4
3      job := &Flusher{
4          db:      db,
5          enabled:  atomic.NewBool(true),
6          queueIndex: atomic.NewUint32(0),
7          queueMask:  uint32(queueSize - 1),
8          queues:    NewQueues(queueSize),
9      }
10 ...

```

We set the queueSize to be a power of 2 value - we are shifting the value 1 left by 4 bits, meaning that 1 << 4 is the same as writing 16 (binary 10000). If you wanted a larger queue, you'd shift by a larger number of bits.

The queueMask is the binary masking value - for a queueSize=16, this evaluates to binary 1111. We can use the queueMask over any uint32 value to get only the lower 4 bits of that value.

This is an optimisation for speed. If your queue size would be 100, you'd have to use the modulo (%100) operator to give you the remainder of the division against this number. We could have used %16, but if we use the binary AND & with &15, this operation will be slightly faster.

We can now write the following Push() function to take new \*Incoming values:

---

<sup>38</sup><https://godoc.org/sync/atomic>

<sup>39</sup><https://godoc.org/go.uber.org/atomic>



```

1 // Push spreads queue writes evenly across all queues
2 func (job *Flusher) Push(item *Incoming) error {
3     if job.enabled.Load() {
4         index := job.queueIndex.Inc() & job.queueMask
5         return job.queuees[index].Push(item)
6     }
7     return errFlusherDisabled
8 }

```

We are using the atomic `queueIndex` value to designate into which queue the new item should be written. If the flusher isn't enabled, we return an expected error - we are disabling Push when we are shutting down the process. The client(s) may retry on this error.

This takes care of writing our producer.

## The Consumer

We only need to replace the `flush()` function in order to implement our queue consumer:

```

1 func (job *Flusher) flush() {
2     var err error
3
4     fields := strings.Join(IncomingFields, ",")
5     named := ":" + strings.Join(IncomingFields, ",:")
6     query := fmt.Sprintf("insert into %s (%s) values (%s)", IncomingTable, fields, nam\
7 ed)
8
9     var batchSize int
10    for k, queue := range job.queuees {
11        rows := queue.Clear()
12
13        for len(rows) > 0 {
14            batchSize = 1000
15            if len(rows) < batchSize {
16                batchSize = len(rows)
17            }
18            log.Println("[flush] queue", k, "remaining", len(rows))
19            if _, err = job.db.NamedExec(query, rows[:batchSize]); err != nil {
20                log.Println("Error when flushing data:", err)
21            }
22            rows = rows[batchSize:]
23        }

```

```

24     }
25
26     log.Println("[flush] done")
27 }

```

As we are using [jmoiron/sqlx](https://github.com/jmoiron/sqlx)<sup>40</sup>, we have the capability to issue bulk inserts to the database, just by providing a slice of values for insertion. In order of implementation:

1. we prepare a query for inserting our data,
2. we read and clear individual queues of all items,
3. we flush the read items to the database, 1000 rows at a time

In case you have issues with this, try issuing `go get github.com/jmoiron/sqlx@master` to pull in the latest version of sqlx into your project. Bulk inserts have been implemented only recently, and you might have to use the master version of sqlx.

## Benchmarking

Since we already have the tests set up from the previous implementation, let's compare how much the service is improved with the queuing:

```

1 Running 1m test @ http://172.29.0.2:3000/twirp/stats.StatsService/Push
2   4 threads and 100 connections
3   Thread Stats   Avg      Stdev     Max   +/-  Stdev
4     Latency    12.54ms   21.19ms  416.79ms   91.30%
5     Req/Sec    3.80k     1.63k   11.03k    63.05%
6   909154 requests in 1.00m, 94.51MB read
7 Requests/sec:  15129.35
8 Transfer/sec:    1.57MB

```

And the previous version without queuing:

---

<sup>40</sup><https://github.com/jmoiron/sqlx>

```

1 Running 1m test @ http://172.22.0.5:3000/twirp/stats.StatsService/Push
2   4 threads and 100 connections
3   Thread Stats   Avg      Stdev     Max   +/-  Stdev
4     Latency    47.69ms   11.86ms 143.28ms   73.10%
5     Req/Sec    527.04     85.10  800.00    73.25%
6   125930 requests in 1.00m, 13.09MB read
7 Requests/sec:   2097.16
8 Transfer/sec:   223.23KB

```

Let's also verify that our data is written:

```

1 mysql> select count(*) from incoming;
2 +-----+
3 | count(*) |
4 +-----+
5 |   909251 |
6 +-----+
7 1 row in set (0.06 sec)

```

The small discrepancy in requests may be explained by wrk closing the connection(s) before they could receive the final response, but the request was logged in the database.

So, by queuing our incoming data in memory, we reduced our average latency by about 74%. Our throughput in requests/sec increased from 100% to 721% (7.2x!).

## Notes and pitfalls

The flush process takes its own time when being slammed with benchmark requests:

```

1 [flush] begin
2 [flush] queue 0 rows 23388
3 [flush] queue 1 rows 24330
4 [flush] queue 2 rows 24578
5 [flush] queue 3 rows 23283
6 [flush] queue 4 rows 22134
7 [flush] queue 5 rows 20695
8 [flush] queue 6 rows 19347
9 [flush] queue 7 rows 17880
10 [flush] queue 8 rows 16416
11 [flush] queue 9 rows 14987
12 [flush] queue 10 rows 13359
13 [flush] queue 11 rows 11659

```

```
14 [flush] queue 12 rows 9964
15 [flush] queue 13 rows 7989
16 [flush] queue 14 rows 6301
17 [flush] queue 15 rows 4596
18 [flush] done
```

We can see that the data is queueing up nicely. The flush for the data does take a while - longer than the 5 second flush interval which we specified. Lets consider the pitfalls of the current implementation:

- Memory allocations,
- Error handling

We don't limit memory allocations - theoretically, given similar stressful circumstances as our benchmark provides, it is possible for our program to reach resource exhaustion. One way to reach a level of safety is to introduce backpressure when our queues fill up. Backpressure means that new writes to the queue are rejected, until the existing queue is flushed to the database.

This also means that we could pre-allocate the queues, and avoid allocations when adding new data to the queue, by implementing a circular buffer. New items would overwrite older items until the queue fills up and rejects new writes.

Error handling: currently we don't do any error handling, we just print an error when flushing to the database. A possible reason for this would be an unplanned database outage. We could wait and retry the writes, having some data security. This requires us to implement that circular buffer and backpressure, as resource exhaustion is more likely if we are handling this error scenario.

To sum up: set reasonable queue limits and keep allocations down to make your service more robust in case of unplanned traffic spikes or outages.