



HYPERWORKS

"Use GO to build scalable Backends"

Microservices in GO



Matthew Campbell

Founder of Langfight and Errplane

Microservices in Go

Matthew Campbell

Microservices in Go

Matthew Campbell

Table of Contents

.....	vi
1. Performance	1
1.1. Metrics	1
1.2. Why Metrics are important	1
1.3. Tools of the trade	1
1.4. Never get slower	1
1.5. net.Context	2
1.6. InfluxDB	2
1.6.1. Install InfluxDB	2
1.7. Metrics in Go	3
1.8. Using Grafana with InfluxDB	5
1.8.1. Install Grafana	5
1.9. Monitor	6
1.10. Stacking Metrics	13
1.11. Distributed Tracing with ZipKin	13
1.11.1. ZipKin	14
1.11.2. Setting up a ZipKin server	15
1.11.3. Tracing a Go program.	16
1.11.4. Tracing across services.	19
1.12. Role of Caches, Queues and Databases	21
1.12.1. Overview of why they are important	21
1.12.2. Delay everything Queues	21
1.13. Go Profiling	21
1.14. Load tests	21
1.14.1. Boom	22
1.14.2. Vegeta	23
1.14.3. Load test with Jenkins and jMeter Report	24
1.15. General coding tips	29
1.15.1. Garbage collection pointers	29
1.15.2. Concatenation string performance	33

List of Figures

1.1. An example trace diagram.	14
1.2. Finding traces on ZipKin web UI.	18
1.3. Span inspection	19
1.4. Trace spanning more than one services	21

Chapter 1. Performance

Nowadays having a fast service is not optional, it's a requirement from day one. Slow apps will lose customers. Mobile apps tend to be on slow connections, it's important monitoring performance and bundling calls together to optimize this. Sometimes it's not your code that's slow, but the infrastructure, dns, or even the location of your datacenter. We will go into how we monitor both inside and outside your infrastructure to get the best performance. We will dig into how we can use InfluxDB and Grafana to monitor them.

1.1. Metrics

Runtime metrics are the corner of all performance monitoring. If you want to see an amazing intro to the topic watch CodeHale's talk on metrics (here [<https://www.youtube.com/watch?v=czes-0a0yik>]). Basically the idea is as your application runs in a production environment you are keeping counters and timers of everything that is happening. So you can pinpoint sql performance issues on node #6 of your cluster.

1.2. Why Metrics are important

In the past most performance testing was done with weeks of load tests before we put an application in production. Then we would proceed to have nearly any statistics on how the app performed for actual users, you know the ones we actually care about. These days most people have runtime metrics inside their applications. Sometimes as granular as per user or per http transaction.

1.3. Tools of the trade

Talk about Graphite, Statsd, Influxdb and Grafana, Prometheus. A quick overview of the choices for metrics. Graphite and Statsd are probably the oldest and most stable solutions for graphing, in fact they are great and I've been using them for the last three years. There has been so much writing on them, I wanted to showcase some of the newer tools. Also I want to show some stacked metrics which are difficult to achieve in Graphite, but fit nicely into InfluxDb/Prometheus. For the sake of brevity we are going to be using InfluxDb as our backend stats system, with a Grafana frontend. This book continues to make opinionated choices on tools, to not overload the user. The concepts can be applied to any tool, however we like to have one concrete example.

1.4. Never get slower

Number 1 rule of performance, is to never get slower. That's why you need to do monitoring at three different stages.

1. performance unit tests on the local dev workstation.
2. Automated load tests on Jenkins
3. Application level metrics in production running instances of our applications.

1.5. net.Context

Net context is a really cool way to pass data throughout pieces of your code, similar to thread local storage. Except it's extremely explicit. There is a great intro to it <https://blog.golang.org/context>. We are going to use Context for two things in this book, structured logging and performance tracking. We can do cool things like tabulate the mysql performance for a full http request, then we can even go one step further and track it across multiple servers.

```
package main

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

    "golang.org/x/net/context"
)

func contextHandler(w http.ResponseWriter, req *http.Request) {
    var timeInMilliseconds time.Duration = 0
    ctx := context.WithValue(context.Background(), "time", &timeInMilliseconds)

    longTimeSqlFunc(ctx)
    longTimeSqlFunc(ctx)
    longTimeSqlFunc(ctx)

    val := ctx.Value("time").(*time.Duration)
    s := fmt.Sprintf("Took in expensive functions(%f)\n", val.Seconds()/1000)
    w.Write([]byte(s))
}

func longTimeSqlFunc(ctx context.Context) {
    defer func(s time.Time) {
        val := ctx.Value("time").(*time.Duration)
        *val = *val + time.Since(s)
    }(time.Now())

    time.Sleep(time.Second)
}

func main() {
    http.HandleFunc("/", contextHandler)
    http.ListenAndServe(":8080", nil)
}
```

1.6. InfluxDB

InfluxDb is a json timeseries database. Disclosure: I was a previous founder of the company Errplane, which spun out into this. I never directly worked on influx, however have watched it from a distance and I am very impressed. We are going to use that as the backend for all of our samples.

1.6.1. Install InfluxDB

We use docker help us install.


```
docker run --name influxdb -d -p 8083:8083 -p 8086:8086 -e PRE_CREATE_DB="metric" tutum/
```

InfluxDB use port 8083 for web-ui and we can access the database with port 8086. `PRE_CREATE_DB` will create database after running success.

1.7. Metrics in Go

In GO we are going to use the metrics library `go-metrics`, inspired by the original CodeHale Metrics library in scale. Its really easy to setup

In this example we will create web application. The collect number of request and response time to InfluxDB.

```
import (
    "github.com/GeertJohan/go-metrics/influxdb"
    "github.com/rcrowley/go-metrics"
    "net/http"
    "time"
)

func MetricToInfluxDB(d time.Duration) {
    go influxdb.Influxdb(metrics.DefaultRegistry, d, &influxdb.Config{
        Host:      "192.168.99.100:8086",
        Database:  "example",
        Username:  "root",
        Password:  "root",
    })
}

func IndexHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello world"))
}

func main() {
    MetricToInfluxDB(time.Second * 1)

    http.HandleFunc("/", IndexHandler)

    http.ListenAndServe(":3000", nil)
}
```

In `MetricToInfluxDB()`, we create goroutine for monitor metric and save them into influxdb every `d` duration. We set duration in `main()` to 1 second, `IndexHandler()` is a http handler for display Hello world in `http://localhost:3000`.

Next we will add a counter, when we go to `http://localhost:3000`, it will increase 1.

```
...

var requestCounter metrics.Counter

...

func IndexHandler(w http.ResponseWriter, r *http.Request) {
    requestCounter.Inc(1)

    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello world"))
}

...

func main() {
    requestCounter = metrics.NewCounter()
    metrics.Register("count_request", requestCounter)

    MetricToInfluxDB(time.Second * 1)

    http.HandleFunc("/", IndexHandler)

    http.ListenAndServe(":3000", nil)
}
```

We created `requestCounter` with `metrics.NewCounter()` and register to the metric name `count_request`. So we can see this counter in InfluxDB with column `count_request`. On `IndexHandler()` we added `requestCounter.Inc(1)` to increase counter by 1.

```
...

var responseTime metrics.Timer

...

func IndexHandler(w http.ResponseWriter, r *http.Request) {
    requestCounter.Inc(1)
    startReqTime := time.Now()
    defer responseTime.Update(time.Since(startReqTime))

    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello world"))
}

func main() {
    requestCounter = metrics.NewCounter()
    metrics.Register("count_request", requestCounter)

    responseTime = metrics.NewTimer()
    metrics.Register("response_time", responseTime)

    MetricToInfluxDB(time.Second * 1)

    http.HandleFunc("/", IndexHandler)

    http.ListenAndServe(":3000", nil)
}
```

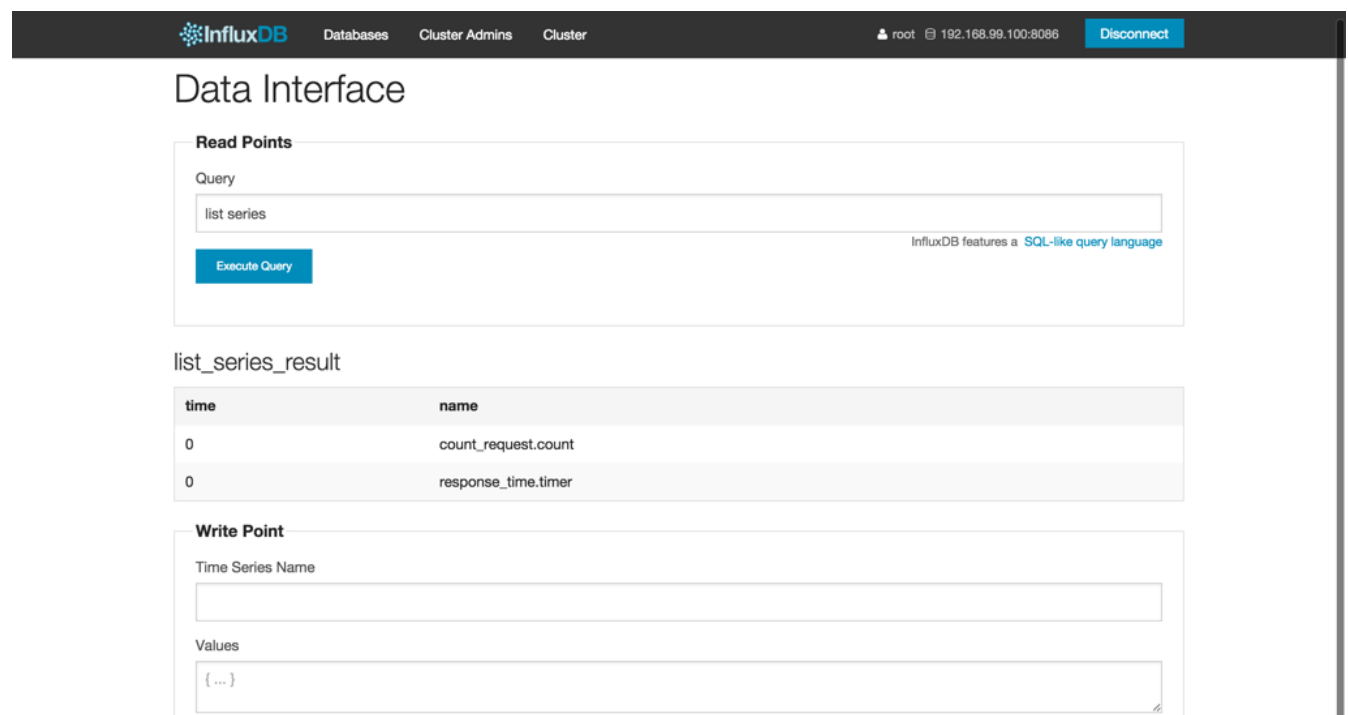
Next we will add response-time timer. In above code, we created a new timer call `responseTime` and register to the metric name `response_time`. On `IndexHandler`, we created `startReqTime` and defer `responseTime.Update(time.Since(startReqTime))` to make a response-time.

Now we finished in go code, you can run `http://localhost:3000` many times to make the data and send to the InfluxDB. After that we will go to InfluxDB web-ui. Assume that you run InfluxDB in your local machine, go `http://localhost:8083`. You will see the login page. The default InfluxDB docker will create a root account, Fill following information into this page and login.

- Username: `root`
- Password: `root`
- Port: `8086`

After login, Select metric database which we created on docker `run`. You will go to the query page.

To see all column in our database you can run this query `list series` This query will show all of series in your database. You can bring the series to use on the grafana.



The screenshot shows the InfluxDB Data Interface. At the top, there's a navigation bar with 'InfluxDB', 'Databases', 'Cluster Admins', and 'Cluster'. On the right, it shows the user 'root' and the address '192.168.99.100:8086' with a 'Disconnect' button. The main section is titled 'Data Interface' and contains a 'Read Points' section. In this section, the 'Query' input field contains 'list series', and there is an 'Execute Query' button. Below the query input, it says 'InfluxDB features a SQL-like query language'. The results are displayed in a table titled 'list_series_result'.

time	name
0	count_request.count
0	response_time.timer

Below the table, there is a 'Write Point' section with a 'Time Series Name' input field and a 'Values' input field containing '{ ... }'.

1.8. Using Grafana with InfluxDB

Grafana is the tool we are going to use for all the charts. As we go through the sections i'll show you also how to visualize step by step.

1.8.1. Install Grafana

We already created Grafana docker image which working with InfluxDB.

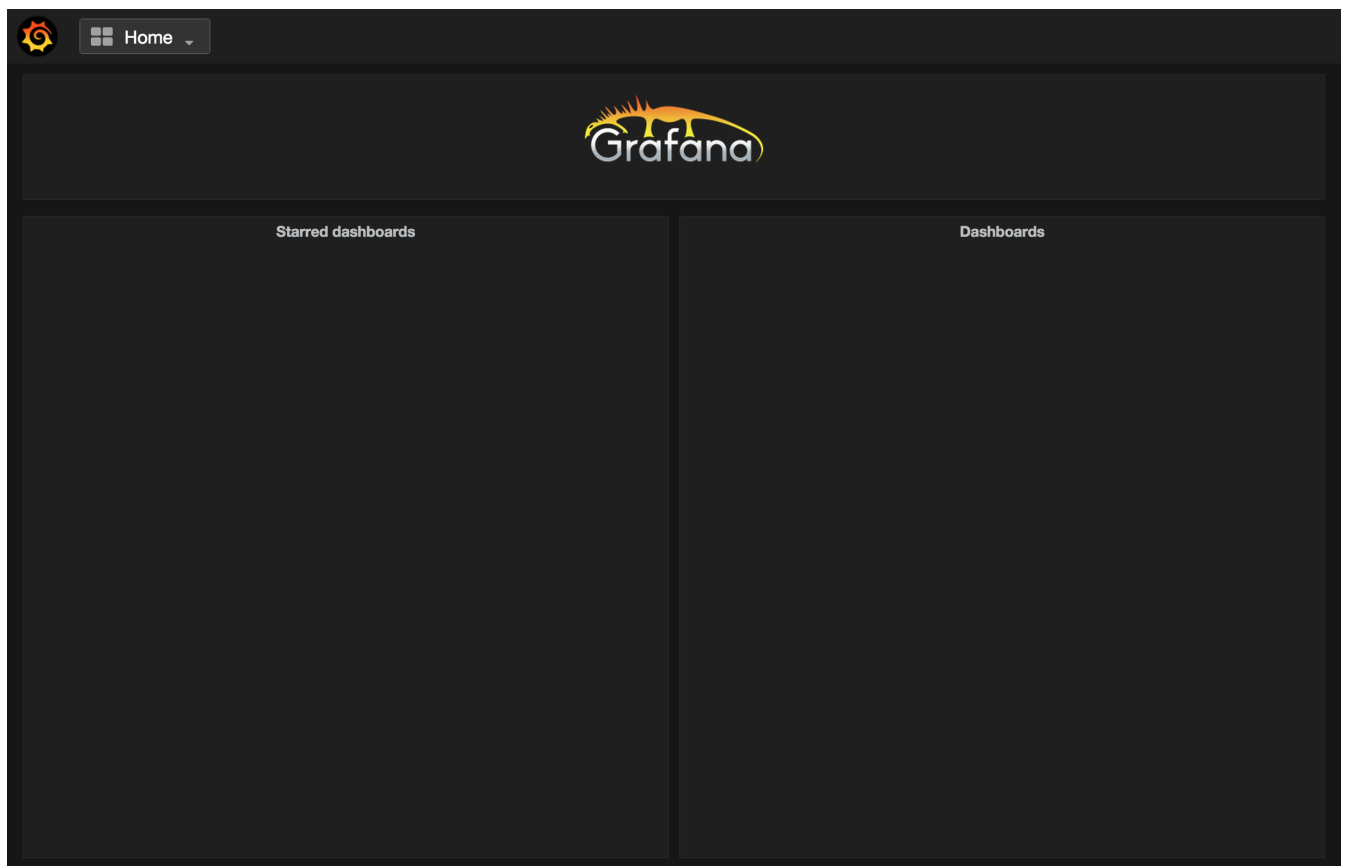
```
docker run --name grafana -d --link influxdb:influxdb \  
-e INFLUXDB_HOST=influxdb \  
-e INFLUXDB_PORT=8086 \  
-e INFLUXDB_NAME=metric \  
-e INFLUXDB_USER=root \  
-e INFLUXDB_PASS=root \  
-p 3300:80 \  
hyperworks/grafana
```

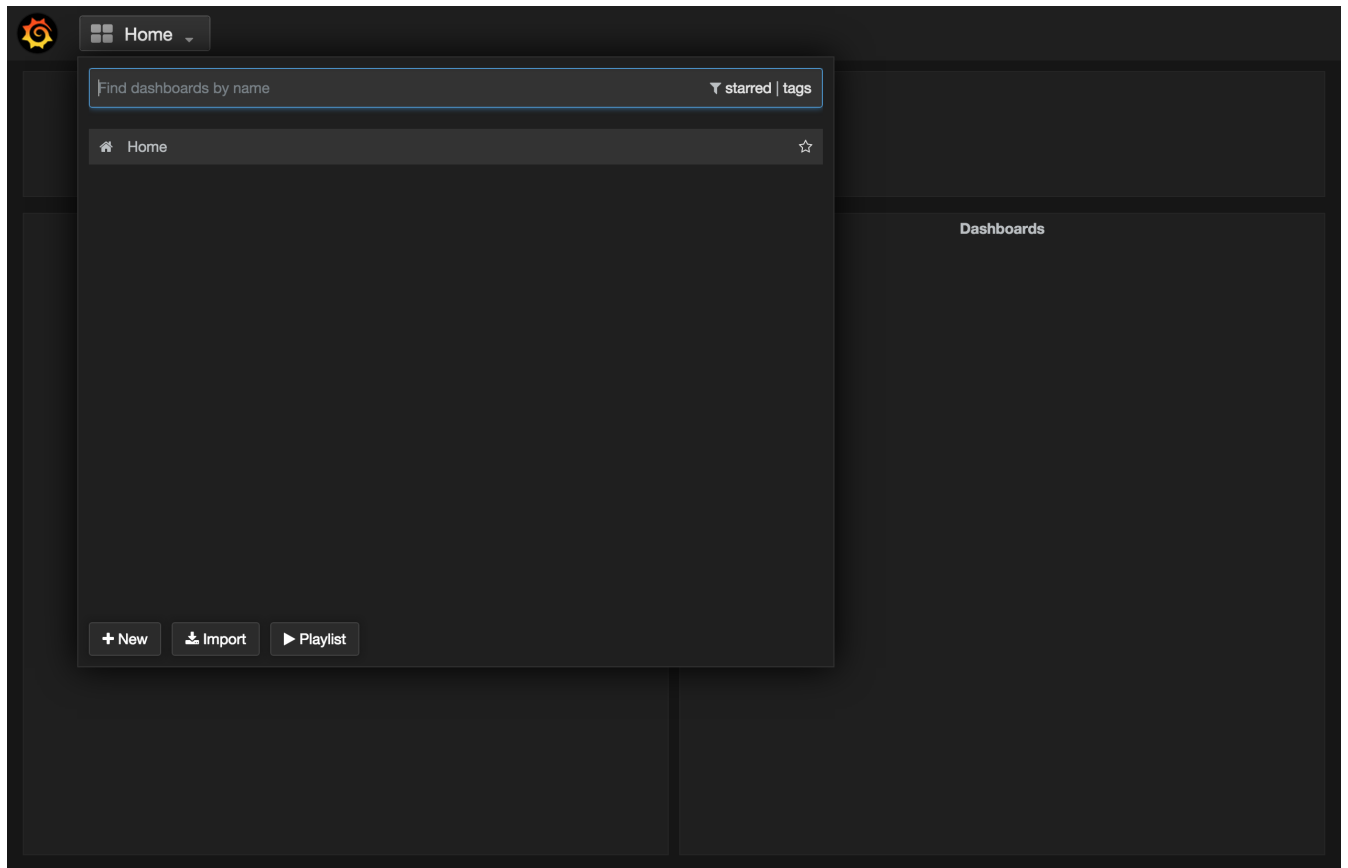
The above command is create grafana container and link it to influxdb. So this container can access the influxdb container. The environment variables that we set are influxdb information for create datasource in grafana.

Go to url <http://localhost:3300>, you will see the login page. Use username admin and password admin to login. This is the default account which come with docker image.

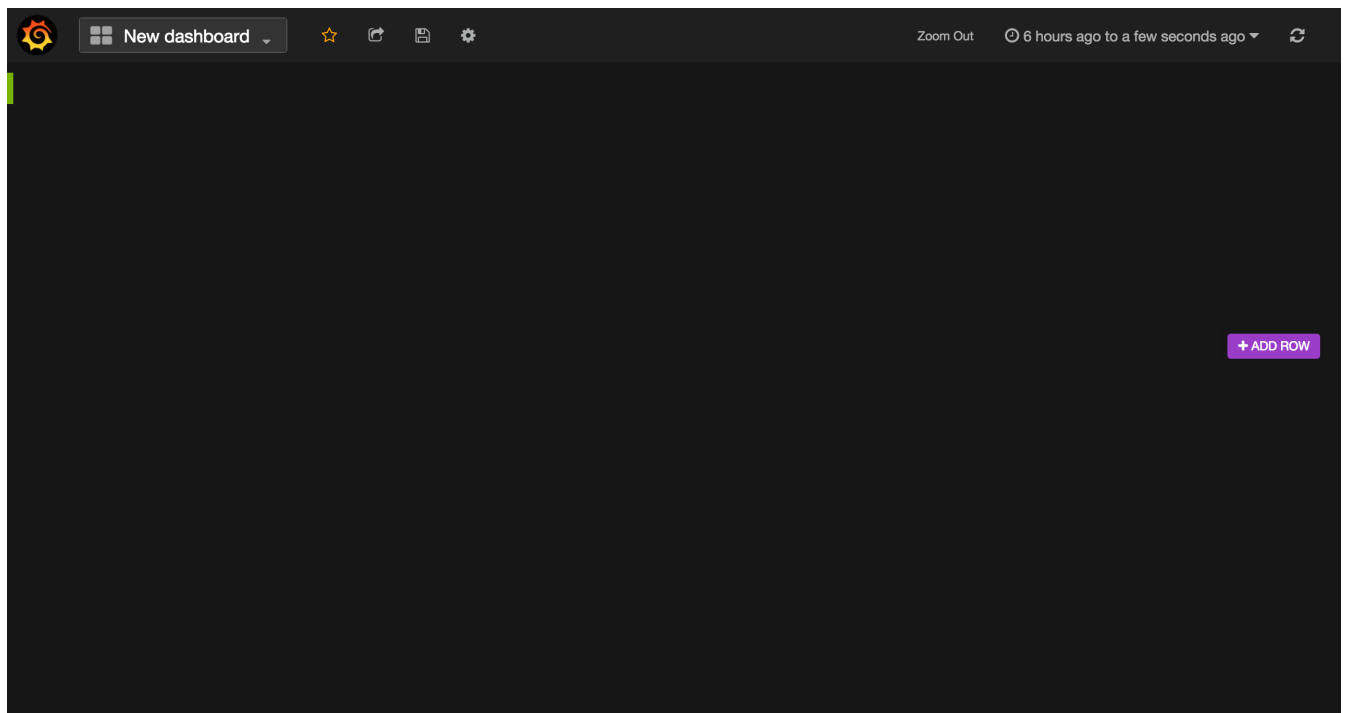
1.9. Monitor

After we login to Grafana, we will add graphs for counter request and response-time that we created go application before. The first page the you will see after login is Dashboard page

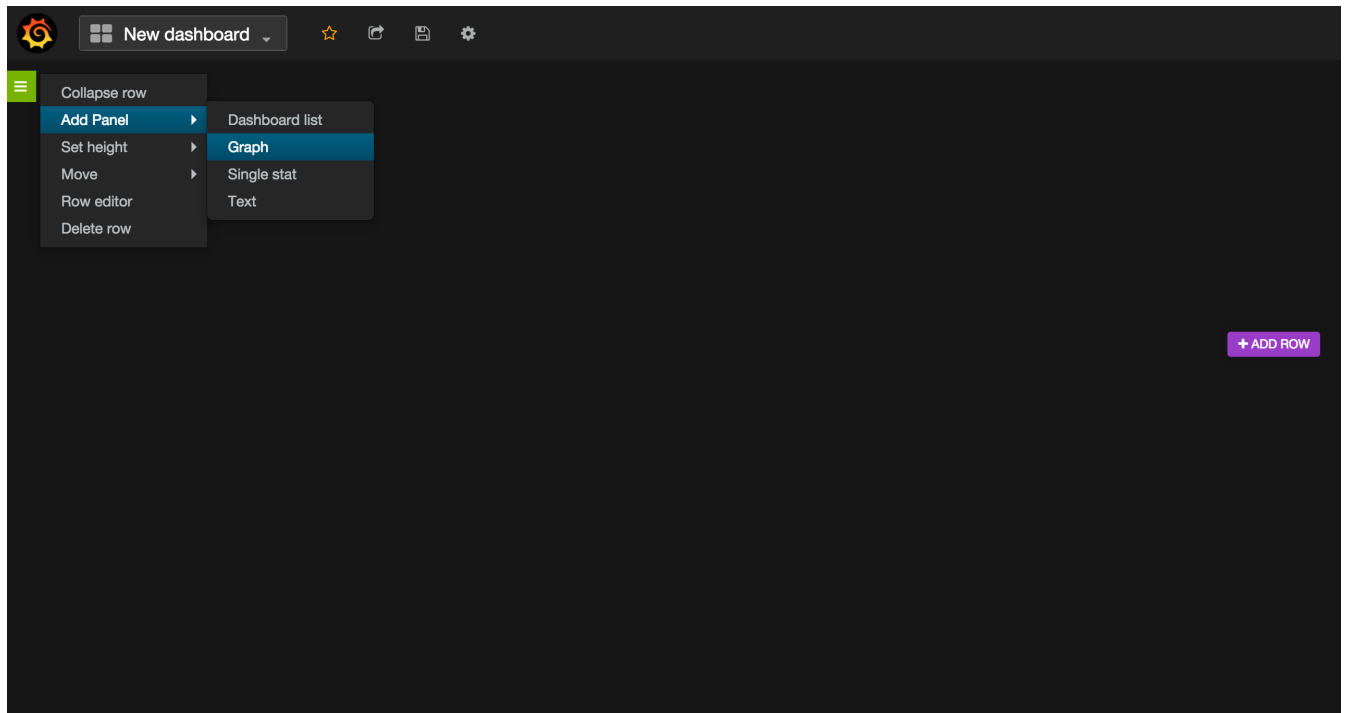




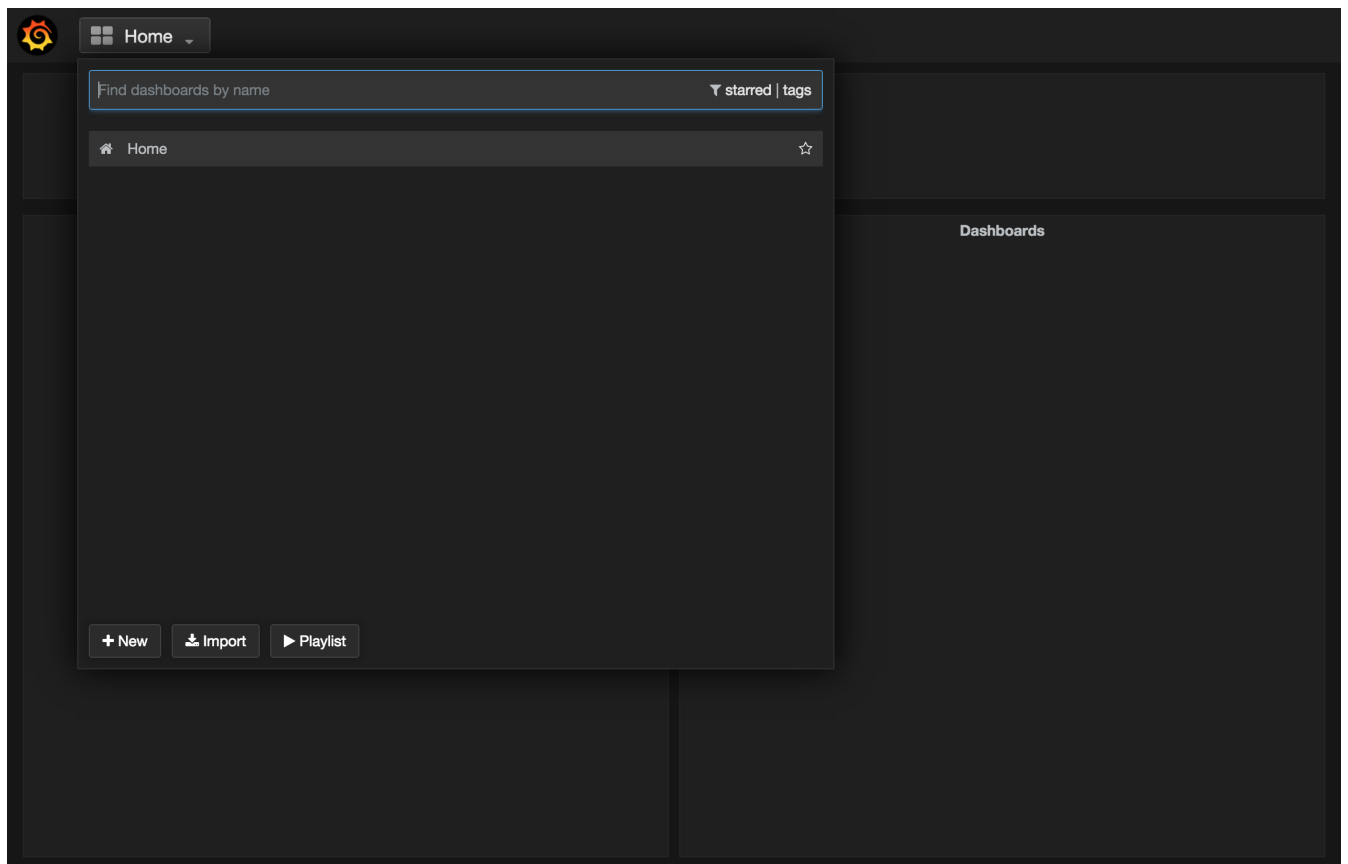
Click Home button on the top, then click New (this will create a new dashboard)



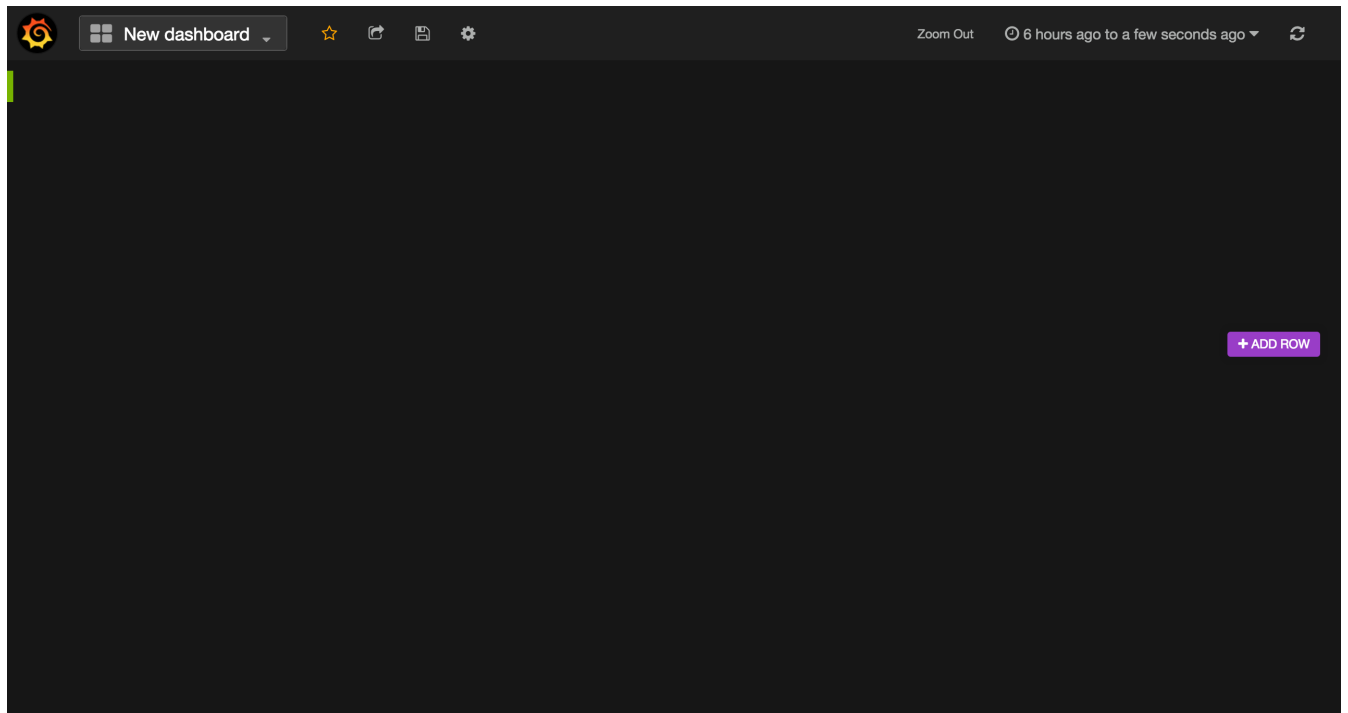
Now you are in side the new dashboard, on the left side you will see a small brown ractangle



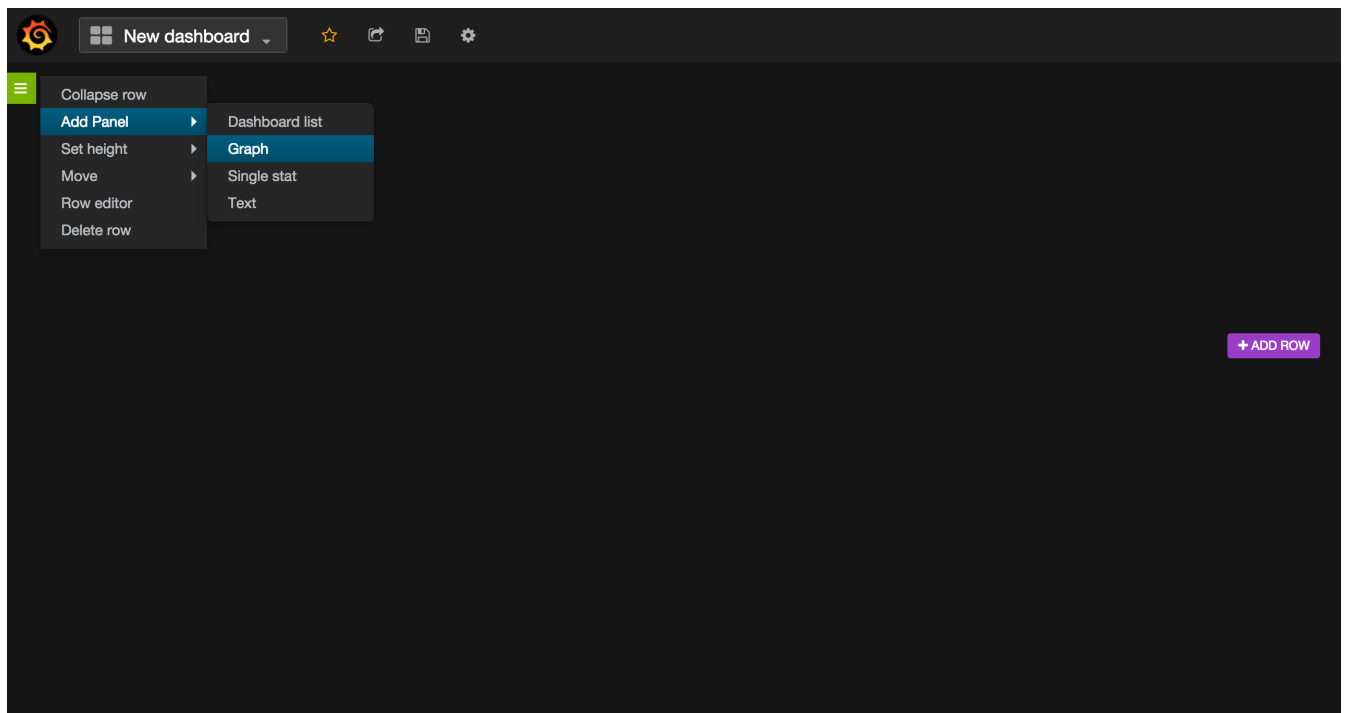
Click that rectangle and Select Add Panel → Graph



When you created, don't forget save the dashboard (disk icon on the top menu). Click Home button on the top, then click New (this will create a new dashboard)

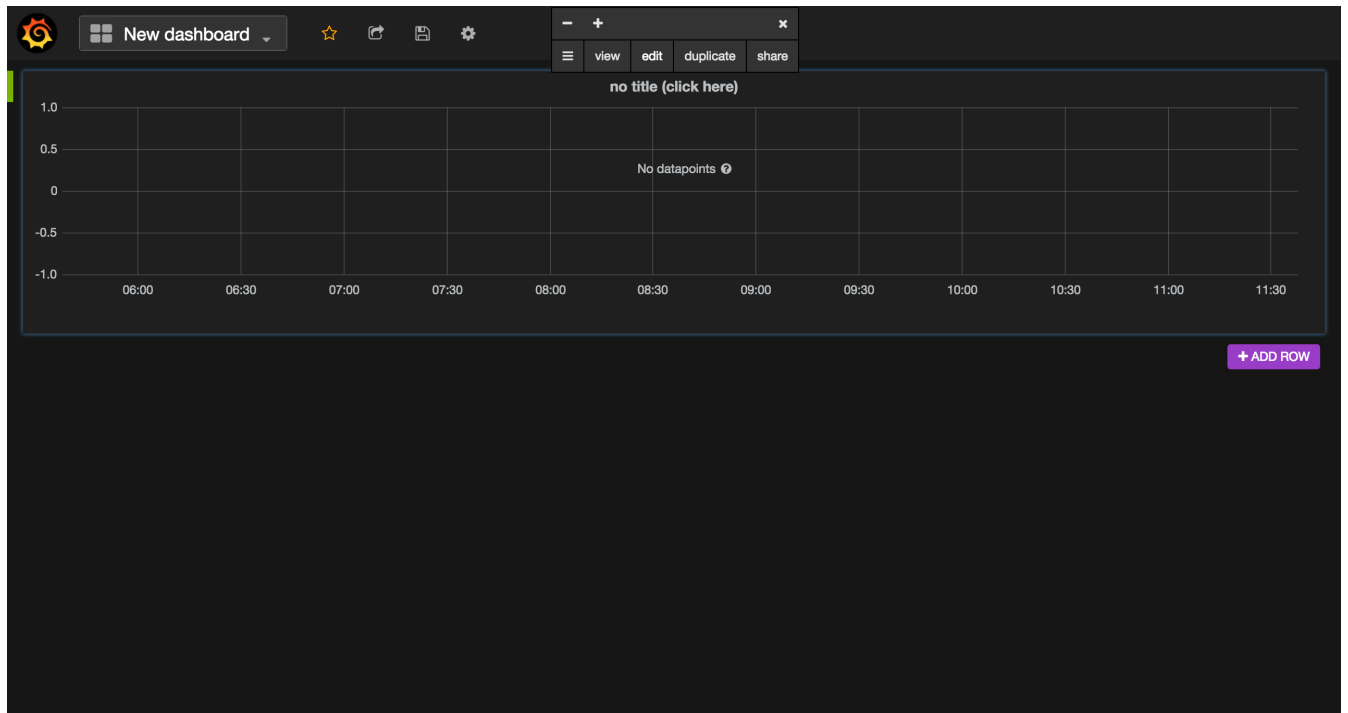


Now you are inside the new dashboard, on the left side you will see a small brown rectangle

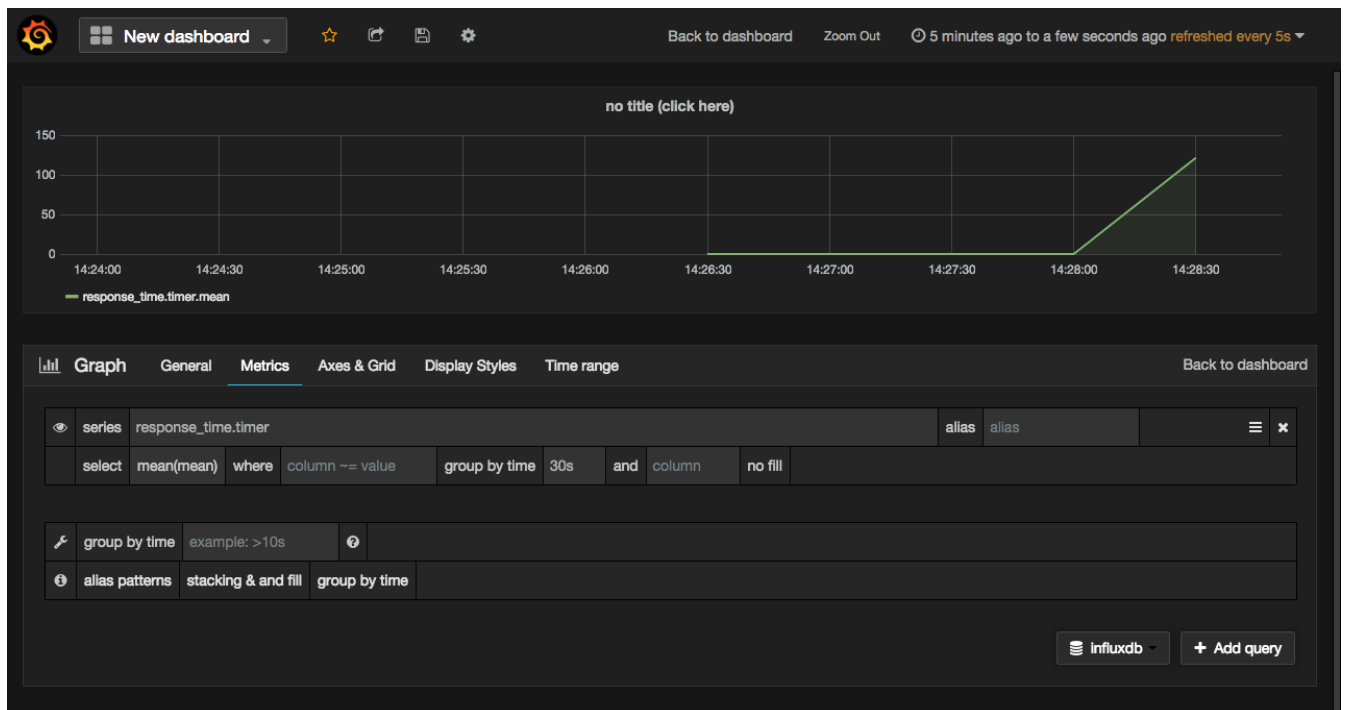


Click that rectangle and Select Add Panel → Graph When you created, don't forget save the dashboard (disk icon on the top menu)

Performance

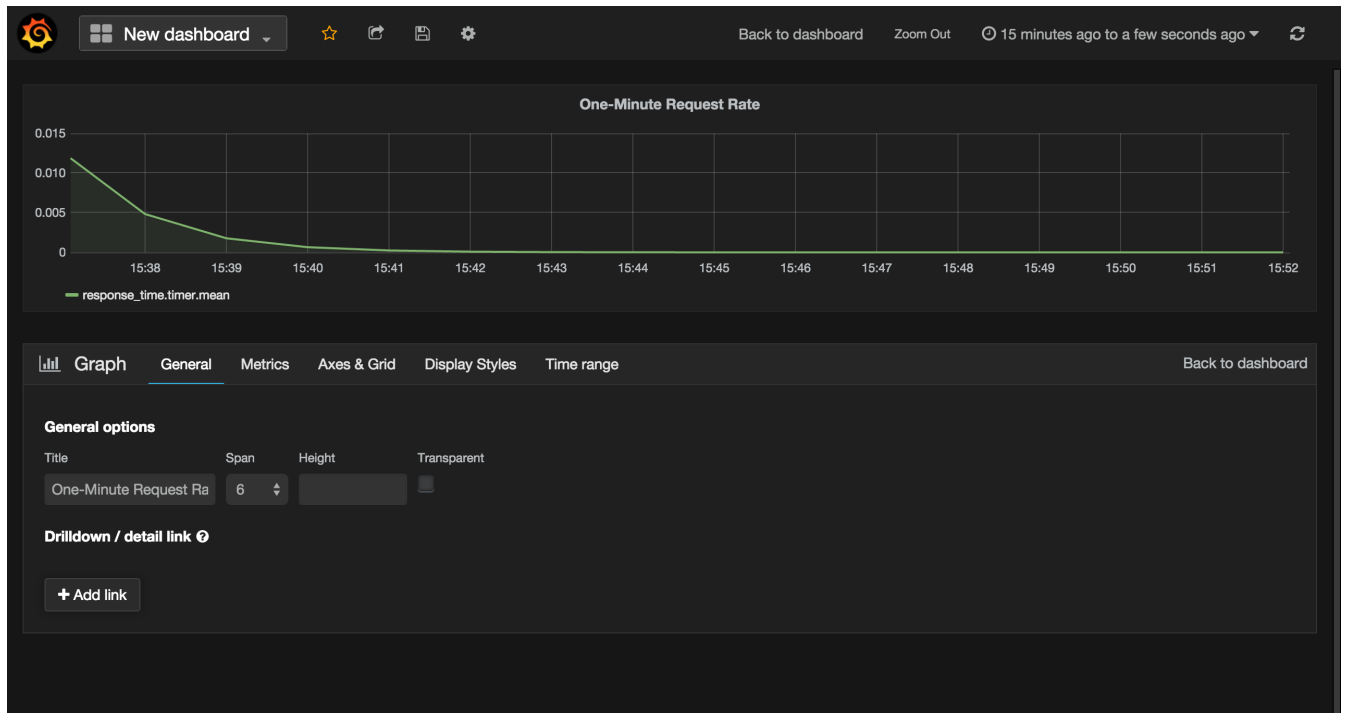


On middle top of graph, click `no title (click here)` then click `edit` for edit graph

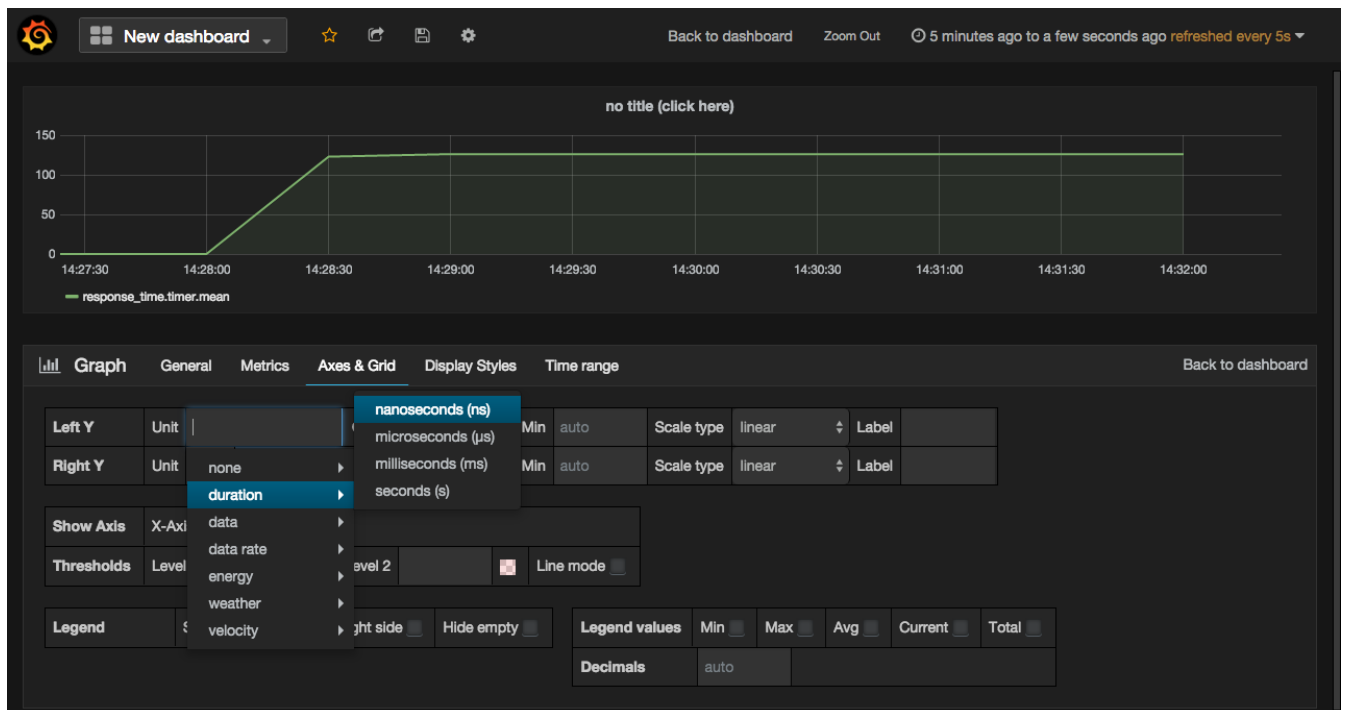


On Metrics tab, you will see the series this series come from datasource. (you can see influxdb series by query `list series` in influxdb query page) Input `response_time.timer` series then you can select fields of series to display on graph. This example input `response_time.timer` series and select mean of mean group time 30s

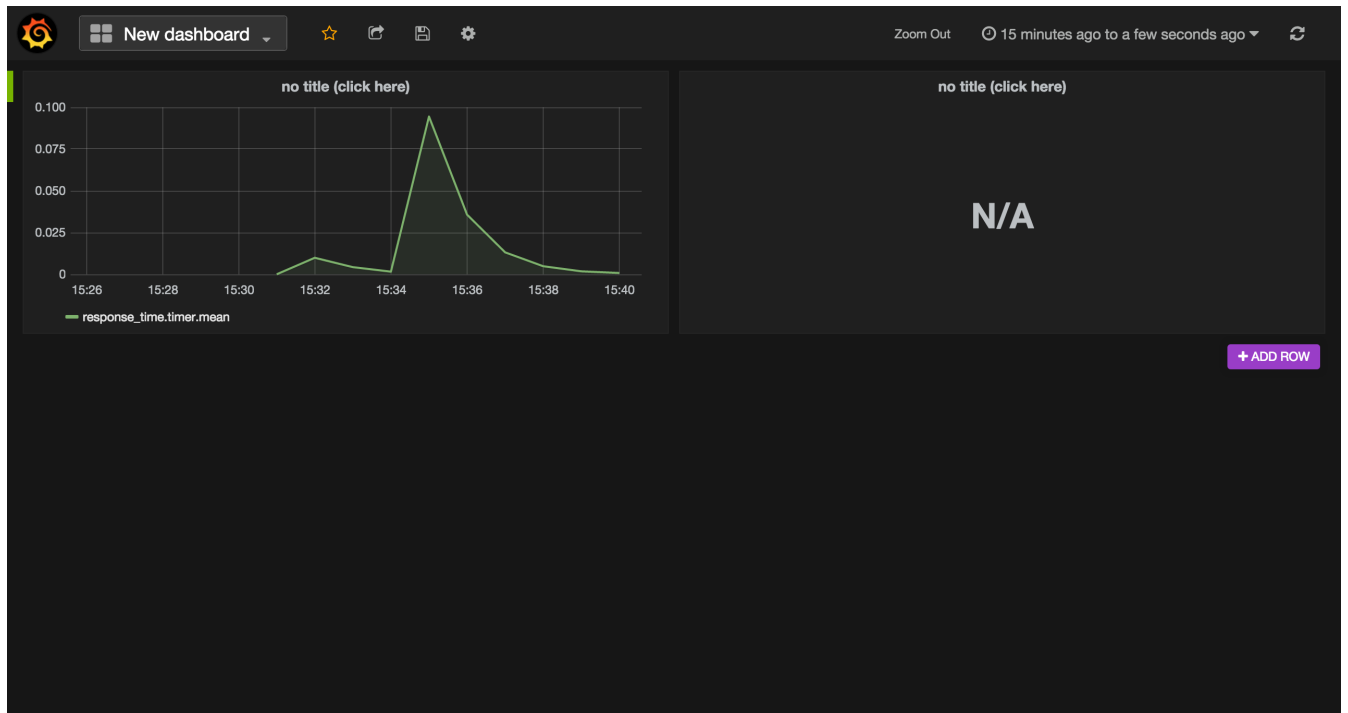
Performance



You can change the title of this graph by go to General tab and change it in title input form



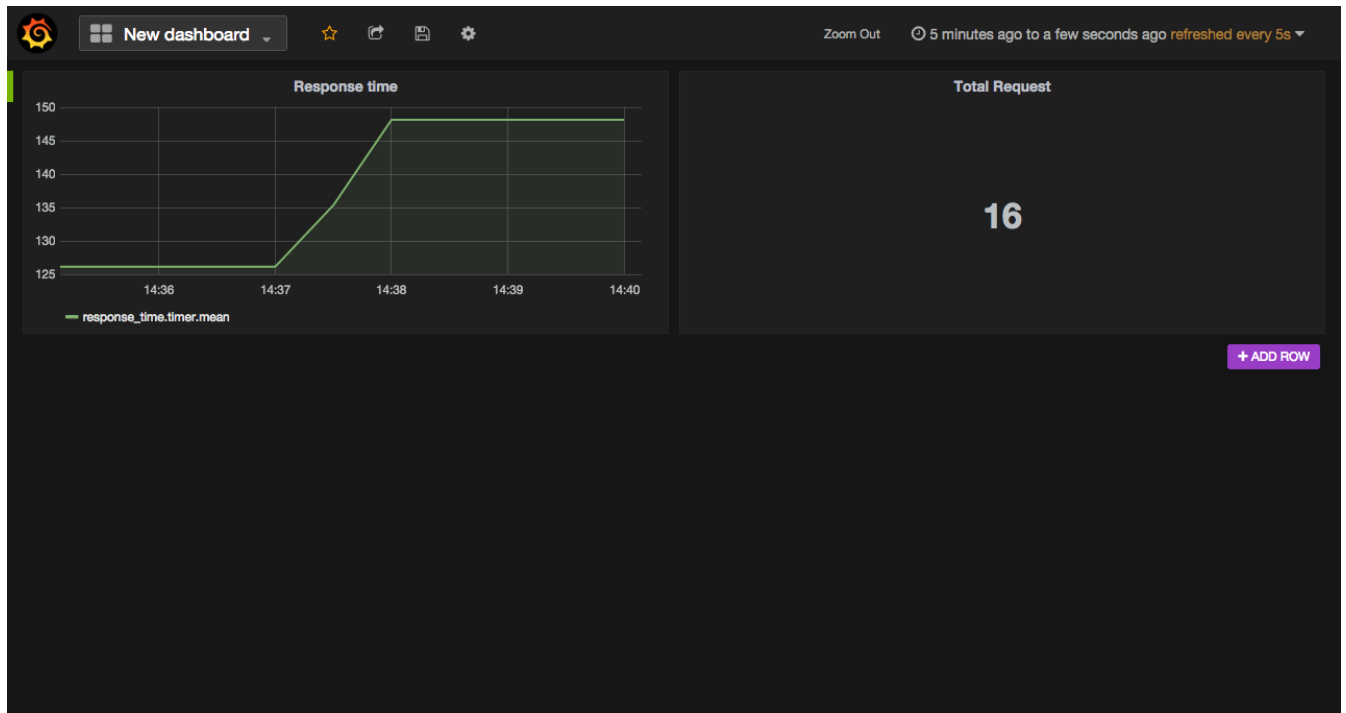
Let's change unit on Y-Axis, click on Axes & Grid tab then click short on Left Y row choose duration → nanoseconds (ns) Then click save dashboard on the top menu (disk icon on the top menu)



Next we will add counter of request into this dashboard Click that rectangle and Select Add Panel → Single stat

This screenshot shows the 'Singlestat' configuration interface in Grafana. The top section displays the value '13'. Below this is a configuration area with tabs for 'General', 'Metrics', 'Options', and 'Time range'. The 'Metrics' tab is currently selected. It contains a table for defining the data source and query. The table has columns for 'series', 'select', 'where', 'group by time', 'alias', and 'no fill'. The 'series' is set to 'count_request.count' and the 'select' is set to 'last(count)'. Below the table, there are sections for 'group by time' (set to 'example: >10s') and 'alias patterns' (set to 'stacking & and fill'). At the bottom right, there are buttons for 'Influxdb' and '+ Add query'.

Again click the top middle of single state no title (click here) then click edit Now we are inside of single state, This state we will add request counter. The series is `count_request.count` select last of count



Then click save dashboard on the top menu (disk icon on the top menu) Result

1.10. Stacking Metrics

1.11. Distributed Tracing with ZipKin

In this section, we will take a look at distributed tracing. Usually tracing a single program allows us to see how an action performed on it translates into calls to each components of the system and how long those calls take. But with microservices architecture, your program may only constitute a single part of a complex system and an action performed at one end may translate into many service calls across multiple machines.

Let's first look at how we might manually trace a simple Go program:

A simple Go program.

```

package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println(outer())
}

func outer() int {
    time.Sleep(1 * time.Second)
    return inner1() + inner2()
}

func inner1() int {
    time.Sleep(1 * time.Second)
    return 1
}

func inner2() int {
    time.Sleep(1 * time.Second)
    return 1
}

```

From the program we can deduce that running the `main()` function will result in a call to the `outer()` function, taking at least 1 second and that the function itself also calls the two other functions, namely `inner1()` and `inner2()` each taking at least a second.

A trace diagram from the above program might look like this:

Figure 1.1. An example trace diagram.



From the diagram we can see the breakdown of the calls in visual form. Having a diagram like this allows us to quickly and easily identify the bottleneck in the system during optimization as well as quickly identify any potential problems.

1.11.1. ZipKin

ZipKin (<http://twitter.github.io/zipkin/>) is a distributed tracing system designed specifically for this scenario and is itself a tracing server that accepts trace "Spans" sent from multiple locations. Let's first take a look at how a ZipKin span looks like.

Sample ZipKin span in JSON format.

```
{
  "trace_id": 3871090523923428400, //
  "name": "main.firstCall",          //
  "id": 3871090523923428400,        //
  "parent_id": null,
  "annotations": [                  //
    {
      "timestamp": 1433850777559706,
      "value": "cs", //
      "host": {
        "ipv4": 0,
        "port": 8080,
        "service_name": "go-zipkin-testclient"
      },
      "duration": null
    },
    {
      "timestamp": 1433850777564406,
      "value": "cr",
      "host": {
        "ipv4": 0,
        "port": 8080,
        "service_name": "go-zipkin-testclient"
      },
      "duration": null
    }
  ],
  "binary_annotations": [],
  "debug": true
}
```

Since ZipKin is designed for distributed tracing, we may have data coming from different sources, the `traceId` here is an identifier that will help us identify Spans that originate from the same action so we can group them together and analyze the parts.

ZipKin spans also have a `name` which is usually the qualified name of the method under trace or the name of the network operation being performed.

Each individual ZipKin spans also have its own `id` and a `parent_id`. These two values help identify the relationship between different ZipKin spans. For example, a trace coming from the `main()` function in our initial example could be a parent to a span identifying the `outer()` call and likewise the trace span from `outer()` would have 2 children being the traces from `inner1()` and `inner2()` call.

ZipKin spans usually also contains two annotations with it, the timestamps at the start of the operation and another at the end so we can use this value to deduce the time the service took. For a client-side operation, the pair of values are named `"cs"` which stands for "client send" and `"cr"` which stands for "client receive". On the server-side the pair of values are named `"sr"` and `"ss"` instead referring to "server receive" and "server send" respectively.

1.11.2. Setting up a ZipKin server

Running Zipkin, we have set up following Zipkin services.

- `Scribe Collector` service for communicate with client to send data
- `Web ui` service for query our span and see in graphic

- Query service for web ui can query data in different data store
- Cassandra data store for collect data from Scribe Collector

This is quite a lot for setup for testing and development. The easiest way we setup is combine them into one docker container and run it with one docker run. Another solution is docker-zipkin [<https://github.com/itszero/docker-zipkin>], packed all services with shell script.

Before install please ensure that your machine have 8 gigabit ram or more. First we have to clone docker-zipkin repository into your machine

```
$ git clone git@github.com:itszero/docker-zipkin.git
```

In deploy folder you will see 2 shell scripts.

1. build.sh run docker build zipkin services
2. deploy.sh run docker run zipkin services

Run build.sh for build docker images. You have to wait for a while for complete this.

```
$ ./deploy/build.sh
```

After we finished building docker image. Let's check the result in `docker images` command, the result will look like this.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
itszero/zipkin-cassandra	latest	fe45e32f270c	2 minutes ago	467 MB
itszero/zipkin-collector	latest	0444124a7ede	2 minutes ago	936.2 MB
itszero/zipkin-web	latest	a8aed6955ab0	2 minutes ago	936.2 MB
itszero/zipkin-query	latest	f106fbe382ba	2 minutes ago	936.2 MB
itszero/zipkin-base	latest	3302a4ac3c59	2 minutes ago	936.2 MB

Right now we got all zipkin services images. Let's run all of them with `deploy.sh`. Before run `deploy` we have config a bit inside `deploy.sh`. In line 5 of `deploy.sh` you will see the `ROOT_URL` config. Change it to your url or ip address for access zipkin web ui. in this example will change it to `ip 127.0.0.1`. You change port with `PUBLIC_PORT` default is 8080.

```
PUBLIC_PORT="8080"
ROOT_URL="http://127.0.0.1:$PUBLIC_PORT"
```

Save and run `deploy.sh`. You will see all services run at the same time.

```
$ deploy.sh
```

Let's check all services is running correctly by `docker ps`.

```
$ docker ps
```

1.11.3. Tracing a Go program.

Now that we have ZipKin setup, let's send some tracing data to it. For this we will use the `spacemonkeygo/monitor` library on GitHub. Let's add some imports to our simple program first:

```
import (
    "fmt"
    "time"

    "golang.org/x/net/context" //
    "gopkg.in/spacemonkeygo/monitor.v1" //
    "gopkg.in/spacemonkeygo/monitor.v1/trace/gen-go/zipkin" //
)
```

The library makes use of the experimental HTTP context package `golang.org/x/net/context` for sharing values between multiple middlewares and misc HTTP calls.

This is the library itself, imported via `gopkg.in` service so we have version locking while the master branch on GitHub changes.

We will need to supply some ZipKin-specific configuration as well so we need to import this generated code package.

Now that we have all the components imported, let's initialize the tracing library:

```
trace.Configure(1, true, &zipkin.Endpoint{
    Ipv4:      127*0x01000000 + 1,
    Port:      8080,
    ServiceName: "go-zipkin-testclient",
})

if c, e := trace.NewScribeCollector("0.0.0.0:9410"); e != nil {
    panic(e)
} else {
    trace.RegisterTraceCollector(c)
}
```

The `Configure()` calls setup the tracing library with information about our current machine and running program. This line specifies the IP address of the service with an integer value that represents the address 127.0.0.1

The trace library delivers ZipKin spans embedded inside Twitter's Scribe logging protocol so we are creating a new spans collector here that can talk the Scribe protocol and which will send it to our ZipKin server running at address 0.0.0.0 on port 9410. The `RegisterTraceCollector()` call registers the collector as our default collector for all subsequent `trace.Trace()` calls.

And then for each call, we can now call the `trace.Trace()` method to log the times taken during each call. The function returns another function for us to invoke at the end of the tracing method. This method also requires that we pass in a `context.Context`. Methods constituting the same chain of operation should also share the same context instance. For the very first call we can use the `context.Background()` method to obtain a new, empty context.

Let's try adding tracing code for the first method call now:

```
func outer() int {
    ctx := context.Background()
    done := trace.Trace(ctx)

    time.Sleep(1 * time.Second)
    result := inner1() + inner2()

    done(nil)
    return result
}
```

Note that the `done()` function returned from the initial `trace.Trace()` call above also lets us pass in a pointer to an error variable as well. We can pass in a pointer to our error return value variable should we have one and if the method `panic()` for any reason, the tracing call will pick it up and sets the error return value for us.

This pattern however, can be more briefly achieved in Go using the `defer` keyword.

```
func outer() int {
    ctx := context.Background()
    defer trace.Trace(&ctx)(nil)

    time.Sleep(1 * time.Second)
    return inner1() + inner2()
}
```

Likewise for the `inner1()` and `inner2()` methods we can call the `trace.Trace()` method to begin the trace and invoke the result with `defer` to wrap up the call. We will also need to modify the method so that we can pass the `context.Context` instance through as well.

```
// return inner1(ctx) + inner2(ctx)

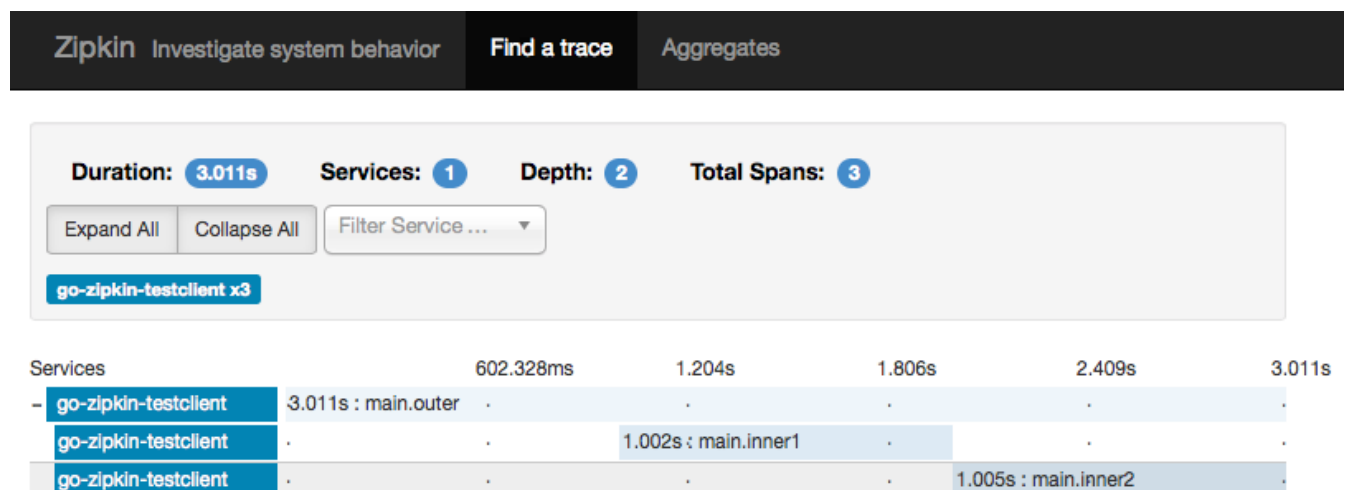
func inner1(ctx context.Context) int {
    defer trace.Trace(&ctx)(nil)
    time.Sleep(1 * time.Second)
    return 1
}

func inner2(ctx context.Context) int {
    defer trace.Trace(&ctx)(nil)
    time.Sleep(1 * time.Second)
    return 1
}
```

Try running the program a few times to generate some spans. Using the ZipKin web UI you can browse and see the span data being sent to ZipKin as well as visualize the call tree.

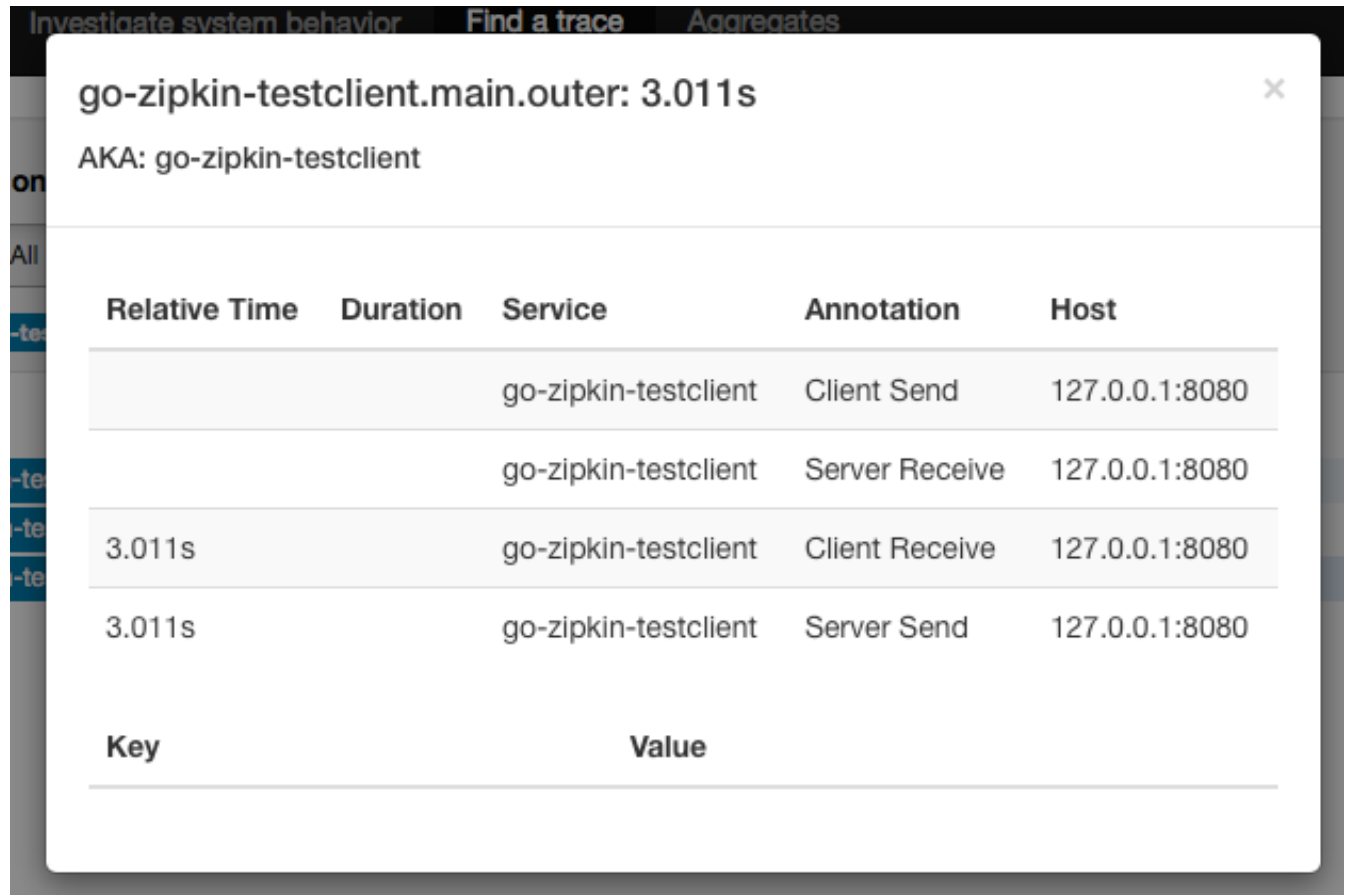
From the web UI, use the service drop-down box to select the service we have named in our code. Clicking on "Find Traces" should produce a list of recent traces being sent to the server.

Figure 1.2. Finding traces on ZipKin web UI.



You can see the call trace graph similar to what we've drawn ourselves earlier from the UI. You can also click on each individual call to see further breakdown and details about the calls:

Figure 1.3. Span inspection



1.11.4. Tracing across services.

To provide a complete tracing picture across services there needs to be a way for each of the services to communicate and share the current tracing session. If all your services talk over HTTP, the tracing library we have used earlier already provides this.

Let's look at how to make a simple HTTP service with tracing using the `spacemonkeygo/monitor` library. First let's make a simple HTTP route:

A simple Go HTTP service.

```
package main

import (
    "net/http"
)

func HelloHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello world #2"))
}

func main() {
    http.ListenAndServe(":3002", http.HandlerFunc(HelloHandler))
}
```

To be able to receive the trace session from another service, we will need to wrap this handler inside a `trace.TraceHandler` and to be able to pass along a `context.Context` from the client, we'll need to also wrap that in `trace.ContextWrapper` as well. Our HTTP Handler will also now needs to accept a `context.Context` that is coming from the previous trace(s), if any.

Tracing added to the HTTP server.

```
// import "golang.org/x/net/context"
// import "gopkg.in/spacemonkeygo/monitor.v1/trace"
// import "gopkg.in/spacemonkeygo/monitor.v1/trace/gen-go/zipkin"

func HelloHandler(ctx context.Context, w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello world #2"))
}

func main() {
    handler := trace.ContextWrapper(trace.TraceHandler(
        trace.ContextHTTPHandlerFunc(HelloHandler)))
    http.ListenAndServe(":3002", handler)
}
```

Just like normal Go programs, we will also need to `Configure()` the trace so that it knows where to send the data just like we did previously. We will omit the code here since it is basically the same code with a different service name.

Now our server will now be able to trace itself as well as add spans to any existing traces being passed over from other services. The `context.Context` instance that was passed to our HTTP server can also be used in further tracing as well. For example, we can add a mock SQL call and add a step to the trace as well:

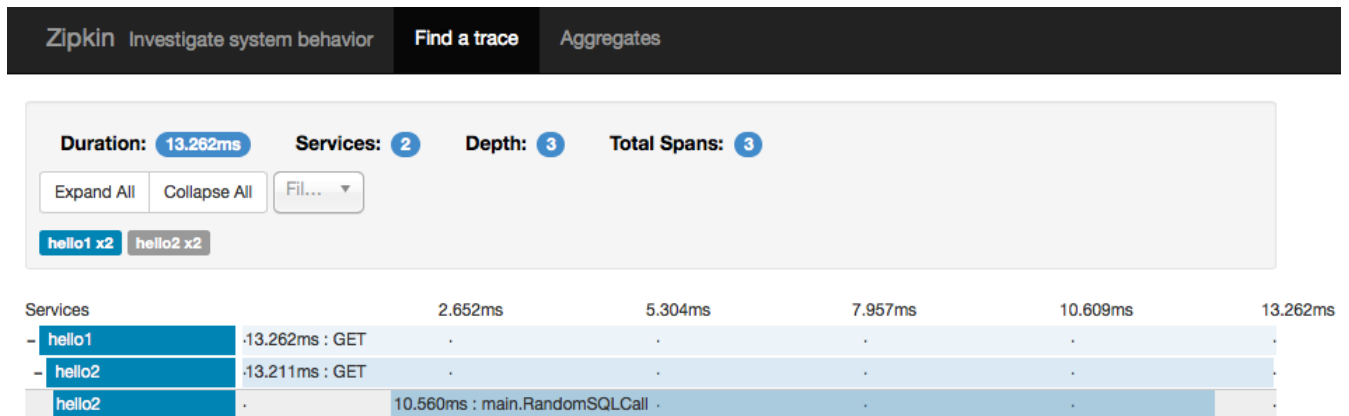
```
func HelloHandler(ctx context.Context, w http.ResponseWriter, r *http.Request) {
    RandomSQLCall(ctx)
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello world #2"))
}

func RandomSQLCall(ctx context.Context) {
    defer trace.Trace(&ctx)(nil)
    time.Sleep(time.Duration(rand.Int()%10000) * time.Microsecond)
}
```

If you have more methods that you think you will need tracing information from, simply pass along the `context.Context` instance to each method and call `defer trace.Trace()()` on it.

Below is an example of a trace on the web UI that goes from a `hello1` service to `hello2` over a network call and then to the `RandomSQLCall()` method:

Figure 1.4. Trace spanning more than one services



As you can see, with ZipKin you can trace your architecture from end-to-end in one place without having to correlate data from disparate services yourself.

1.12. Role of Caches, Queues and Databases

We are just giving a brief overview here as each one has a special chapter.

1.12.1. Overview of why they are important

1.12.2. Delay everything Queues

1.13. Go Profiling

Go already comes with profiling tools. The best place for learning pprof tool is <http://blog.golang.org/profiling-go-programs>

1.14. Load tests

We do load tests for checking the performance in our system, and how many we can handle incoming request. There are many tools can help you do load test and show a report. In this section we will pick load tests tools which written in Go.

This is a example code for running load test, It's just Hello world application.

```
package main

import (
    "net/http"
)

func IndexHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hello world"))
}

func main() {
    http.HandleFunc("/", IndexHandler)

    http.ListenAndServe(":3000", nil)
}
```

1.14.1. Boom

Boom is similar to Apache bench (ab) tool but written in go.

We can install Boom with `go get`

```
go get github.com/rakyll/boom
```

This is an example command we will run on `http://localhost:3000`

```
boom -n 1000 -c 50 -m GET http://localhost:3000
```

This command run 1000 requests and 50 concurrents per second. The result will be like this.

```
Summary:
Total: 1.1616 secs.
Slowest: 1.0122 secs.
Fastest: 0.0015 secs.
Average: 0.0108 secs.
Requests/sec: 860.8770
Total Data Received: 11000 bytes.
Response Size per Request: 11 bytes.

Status code distribution:
[200] 1000 responses

Response time histogram:
0.002 [1] |
0.103 [995] | #####
0.204 [0] |
0.305 [0] |
0.406 [1] |
0.507 [0] |
0.608 [0] |
0.709 [0] |
0.810 [0] |
0.911 [0] |
1.012 [3] |

Latency distribution:
10% in 0.0054 secs.
25% in 0.0063 secs.
50% in 0.0072 secs.
75% in 0.0082 secs.
90% in 0.0101 secs.
95% in 0.0108 secs.
99% in 0.0128 secs.
```

The report in Boom look easily and nice histogram. You can see the how many response http status code.

1.14.2. Vegeta

Vegeta is another one written in Go. Support multiple target and config file.

We can install Vegeta with `go get`

```
go get github.com/tssenart/vegeta
go install github.com/tssenart/vegeta
```

Here is our load test command

```
echo "GET http://localhost:3000" | vegeta attack | vegeta report
```

We pipe string url and send to `vegeta attack`, after attacked we using `vegeta report` to display the result. And this is the example result.

```

Requests [total]      500
Duration [total, attack, wait] 9.981421067s, 9.980964392s, 456.675µs
Latencies [mean, 50, 95, 99, max] 532.152µs, 401.337µs, 1.042179ms, 34.251801ms, 34.251801ms
Bytes In [total, mean] 5500, 11.00
Bytes Out [total, mean] 0, 0.00
Success [ratio] 100.00%
Status Codes [code:count] 200:500
Error Set:

```

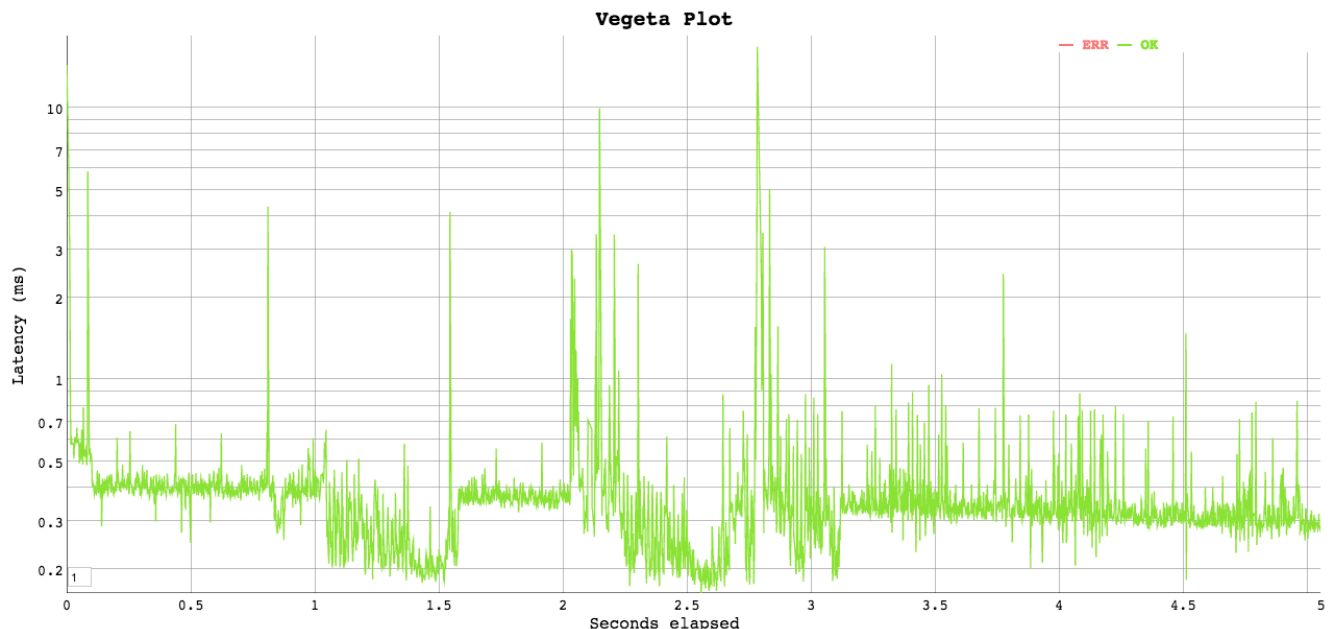
If you want the report in another format like json or graph (Dygraphs [<http://dygraphs.com>]). you just add `-reporter` flag and `-output` flag

```

echo "GET http://localhost:3000" | vegeta attack | vegeta report -reporter json -output result.json
echo "GET http://localhost:3000" | vegeta attack | vegeta report -reporter plot -output result.html

```

This is the example command that report in json format and output to `result.json` Another example is reporter in graph the the result is html format.



[Download as PNG](#)

Open graph result, you will see graph like this.

1.14.3. Load test with Jenkins and jMeter Report

In this section we will talk about how to make an automate load test in Jenkins and show the result with jMeter Report. Jenkins has plugin (Performance Plugin [<https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin>]) which read and display jMeter report. This example will use vegeta tool for load test and convert it to jMeter result format for display report in Jenkins. The goal of this scenario for auto detect performance failure.

Let's try

Install vegeta tool into Jenkins machine and make it can run inside Jenkins project

```
go get github.com/tsenart/vegeta  
go install github.com/tsenart/vegeta  
mv $GOPATH/bin/vegeta /usr/local/bin/vegeta
```

Make a tool for convert vegeta to jMeter. This is an example to convert

```

package main

import (
    "bufio"
    "bytes"
    "encoding/json"
    "encoding/xml"
    "flag"
    "io/ioutil"
    "os"
    "path/filepath"
    "strings"
    "time"
)

type TestResults struct {
    XMLName xml.Name `xml:"testResults"`
    Version string    `xml:"version,attr"`
    Samples []Sample `xml:"httpSample"`
}

type Sample struct {
    Label      string `xml:"lb,attr"`
    Timestamp  int64  `xml:"ts,attr"`
    Success    bool   `xml:"s,attr"`
    Elapsed    int64  `xml:"t,attr"`
    ResponseCode int    `xml:"rc,attr"`
}

type VegetaResult struct {
    Code      int    `json:"code"`
    Timestamp time.Time `json:"timestamp"`
    Latency   int64  `json:"latency"`
}

func WriteJMeter(filename string, vegetaResults []VegetaResult) {
    _, label := filepath.Split(filename)
    index := strings.LastIndex(label, ".")
    label = label[:index]

    result := &TestResults{
        Version: "1.2",
        Samples: make([]Sample, len(vegetaResults)),
    }

    for i := 0; i < len(vegetaResults); i++ {
        result.Samples[i].Label = label
        result.Samples[i].Timestamp = vegetaResults[i].Timestamp.UTC().UnixNano()
        result.Samples[i].Elapsed = vegetaResults[i].Latency / int64(time.Millisecond)
        result.Samples[i].ResponseCode = vegetaResults[i].Code

        if vegetaResults[i].Code > 199 && vegetaResults[i].Code < 300 {
            result.Samples[i].Success = true
        }
    }

    buffer := &bytes.Buffer{}
    buffer.WriteString(xml.Header)

    encoder := xml.NewEncoder(buffer)
    encoder.Indent("", "    ")
    if err := encoder.Encode(result); err != nil {
        panic(err)
    }
}

```


Build it and make it can run inside Jenkins project

```
cd $GOPATH/vegetatojmeter
go build
mv vegetatojmeter /usr/local/bin/vegetatojmeter
```

Let's go to Jenkins and download Performance Plugin [<https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin>] After installed you can make a report after build in each project. Create a new project name load-test and freestyle project mode

Item name

☒ **Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

☐ **Build a maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

☐ **Build multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

☐ **Monitor an external job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).

☐ **Copy existing item**
Copy from

OK

Add Execute shell build step. Put load test and converter command.

Jenkins > load-test > configuration

☐ Build when a change is pushed to GitHub
☐ GitHub Pull Request Builder
☐ Poll SCM

Build Environment

☐ Send files or execute commands over SSH before the build starts
☐ Send files or execute commands over SSH after the build runs
☐ Assign unique TCP ports to avoid collisions
☐ Looks
☐ SSH Agent
☐ Run an Android emulator during build

Build

Execute shell

Command

[See the list of available environment variables](#)

Delete

Add build step

Post-build Actions

Add post-build action

Save Apply

Help us localize this page

Page generated: May 27, 2015 12:44:53 AM [REST API](#) Jenkins ver. 1.582

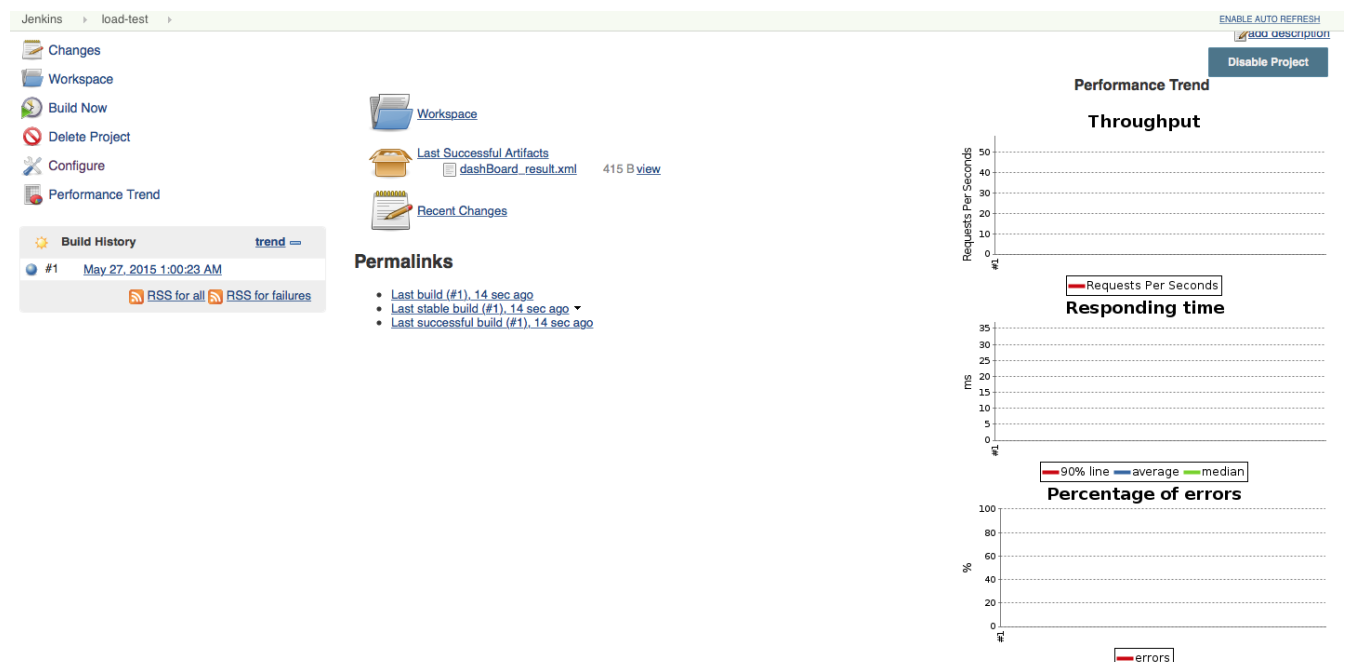
```
echo "GET http://www.example.com" | vegeta attack | vegeta dump -dumper json -output result.json
vegetatojmeter -vegeta result.json -jmeter result.jtl
```

The above script will load test on <http://www.example.com> and dump all requests into result.json. After that run vegetatojmeter to convert result.json from vegeta to result.jtl which is jMeter format

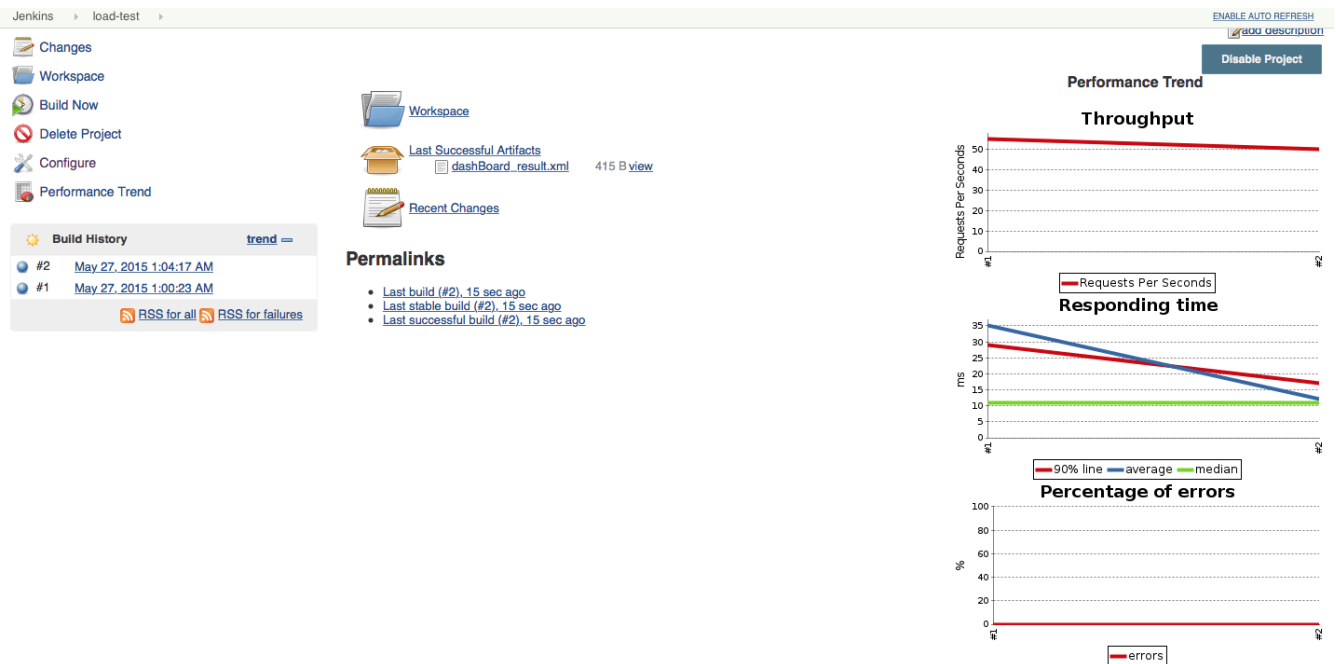
On post build Actions Add Public Performance test result report. This action come from the Performance plugin. The Add a new report JMeter and report files is result.jtl In this example will use Error Threshold mode. So if you have more than one error, the result is unstable If error more than 3, the result will be failed. The unstable should be less than failed.

The screenshot shows the Jenkins configuration page for the 'Publish Performance test result report' plugin. The breadcrumb trail is 'Jenkins > load-test-ebook > configuration'. The page title is 'Publish Performance test result report' with a sub-header 'Performance report'. Under the 'JMeter' section, the 'Report files' field contains 'result.jtl'. There is a 'Delete' button next to it. Below this is an 'Add a new report' dropdown menu. The 'Select mode:' section has two radio buttons: 'Relative Threshold' (unselected) and 'Error Threshold' (selected). Under 'Use Error thresholds on single build:', there are two input fields: 'Unstable' with the value '1' and 'Failed' with the value '3'. The 'Use Relative thresholds for build comparison:' section has two columns for 'Unstable % Range' and 'Failed % Range', each with a '-' and a '+' input field, all containing '0.0'. There are two radio buttons for comparison: 'Compare with previous Build' (selected) and 'Compare with Build number' (unselected) with a value of '0'. The 'Compare based on' dropdown is set to 'Average Response Time'. The 'Performance display' section has two checked checkboxes: 'Performance Per Test Case Mode' and 'Show Throughput Chart'. There are 'Save', 'Apply', and 'Delete' buttons at the bottom.

Then save the setting and try build the job. After success the 3 graph will be displayed on the right hand side. You can see that all of these graphs X-Axis is the build number, right now we have only one build.



Build again to see the different You can see that all graphs has changed



1.15. General coding tips

1.15.1. Garbage collection pointers

###Maybe make a concurrency chapter ===== Mutexes vs Channels

Go does provide traditional locking mechanisms in the `sync` package. Most locking issues can be solved using either channels or traditional locks. It's depend on you, or you can try which way has gain performance for you.

1.15.1.1. Mutex

```
package main

import (
    "math/rand"
    "sync"
)

type MapMutex struct {
    mutex *sync.Mutex
    data  map[int]int
}

func NewMapMutex() *MapMutex {
    return &MapMutex{
        mutex: &sync.Mutex{},
        data:  make(map[int]int),
    }
}

func (m *MapMutex) Read() int {
    key := rand.Intn(5)

    m.mutex.Lock()
    val := m.data[key]
    m.mutex.Unlock()

    return val
}

func (m *MapMutex) Write() {
    key := rand.Intn(5)
    val := rand.Intn(100)

    m.mutex.Lock()
    m.data[key] = val
    m.mutex.Unlock()
}
```

1.15.1.2. Channel

```
package main

import (
    "math/rand"
)

type MapChannel struct {
    lock    chan struct{}
    unlock  chan struct{}
    data    map[int]int
}

func NewMapChannel() *MapChannel {
    m := &MapChannel{
        lock:    make(chan struct{}),
        unlock:  make(chan struct{}),
        data:    make(map[int]int),
    }

    go func() {
        for {
            select {
            case <-m.lock:
                m.unlock <- struct{}{}
            }
        }
    }()

    return m
}

func (m *MapChannel) Read() int {
    key := rand.Intn(5)
    m.lock <- struct{}{}
    val := m.data[key]
    <-m.unlock

    return val
}

func (m *MapChannel) Write() {
    key := rand.Intn(5)
    val := rand.Intn(100)

    m.lock <- struct{}{}
    m.data[key] = val
    <-m.unlock
}
```

Let's see the performance for this code in both ways.

1.15.1.3. Benchmark

```
package main

import (
    "testing"
)

func BenchmarkMutex(b *testing.B) {
    m := NewMapMutex()

    for i := 0; i < 10; i++ {
        m.Write()
    }

    b.ReportAllocs()
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        m.Read()
    }
}

func BenchmarkChannel(b *testing.B) {
    m := NewMapChannel()

    for i := 0; i < 10; i++ {
        m.Write()
    }

    b.ReportAllocs()
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        m.Read()
    }
}
```

We run multi-core to see the different of performance.

```
go test -v -bench . -cpu=1,2,3,4
```

Here benchmark result we got. It's look like the mutex can do it better than channel.

Example Result

PASS				
BenchmarkMutex	20000000	74.2 ns/op	0 B/op	0 allocs/op
BenchmarkMutex-2	20000000	75.8 ns/op	0 B/op	0 allocs/op
BenchmarkMutex-3	20000000	75.3 ns/op	0 B/op	0 allocs/op
BenchmarkMutex-4	20000000	73.5 ns/op	0 B/op	0 allocs/op
BenchmarkChannel	3000000	570 ns/op	0 B/op	0 allocs/op
BenchmarkChannel-2	1000000	1536 ns/op	0 B/op	0 allocs/op
BenchmarkChannel-3	1000000	1339 ns/op	0 B/op	0 allocs/op
BenchmarkChannel-4	1000000	1172 ns/op	0 B/op	0 allocs/op

1.15.2. Concatenation string performance

In Go there are many ways to do concatenation string, But we do not know which one has a good performance. This section we will show you how to concatenate string in different ways and show the performance in each.

1.15.2.1. Use the concatenate operator

Each time you append to the end of a string, it must create a new string and copy the contents of both the existing string and the appended string into it.

```
a := "hello"
a += "world"
```

1.15.2.2. Use strings.Join

strings.Join is a standard library which didn't use + operator to concat string but using byte slice to make to string

```
import "strings"

a := strings.Join([]string{"hello", "world"}, " ")
```

1.15.2.3. Use bytes.Buffer

bytes.Buffer is a standard library which concat via io.Writer, Write byte of string into buffer.

```
import "bytes"

var buffer bytes.Buffer

buffer.WriteString("hello")
buffer.WriteString("world")
```

1.15.2.4. Performance

We will benchmark all methods and see which method is the best performance.

This is example of benchmark code. Concat 10, 1000, 10000 string in each method.

```

package concat

import (
    "bytes"
    "strings"
    "testing"
)

func ConcatOperator(b *testing.B, numStr int) {
    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        str := "a"
        for j := 0; j < numStr; j++ {
            str += "a"
        }
    }
}

func ConcatJoin(b *testing.B, numStr int) {
    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        strArr := []string{"a"}
        for j := 0; j < numStr; j++ {
            strArr = append(strArr, "a")
        }

        strings.Join(strArr, "")
    }
}

func ConcatBuffer(b *testing.B, numStr int) {
    b.ReportAllocs()

    for i := 0; i < b.N; i++ {
        buf := bytes.NewBufferString("a")
        for j := 0; j < numStr; j++ {
            buf.WriteString("a")
        }
    }
}

func BenchmarkOperator10(b *testing.B) {
    ConcatOperator(b, 10)
}

func BenchmarkJoin10(b *testing.B) {
    ConcatJoin(b, 10)
}

func BenchmarkBuffer10(b *testing.B) {
    ConcatBuffer(b, 10)
}

func BenchmarkOperator1000(b *testing.B) {
    ConcatOperator(b, 1000)
}

func BenchmarkJoin1000(b *testing.B) {
    ConcatJoin(b, 1000)
}

func BenchmarkBuffer1000(b *testing.B) {

```


The result of benchmark

PASS					
BenchmarkOperator10	2000000	769 ns/op	80 B/op	10 allocs/op	
BenchmarkJoin10	1000000	1571 ns/op	512 B/op	6 allocs/op	
BenchmarkBuffer10	2000000	756 ns/op	152 B/op	3 allocs/op	
BenchmarkOperator1000	2000	1121344 ns/op	531376 B/op	1000 allocs/op	
BenchmarkJoin1000	20000	68577 ns/op	34784 B/op	12 allocs/op	
BenchmarkBuffer1000	50000	21397 ns/op	2440 B/op	9 allocs/op	
BenchmarkOperator10000	20	62076805 ns/op	53014704 B/op	10000 allocs/op	
BenchmarkJoin10000	2000	834716 ns/op	848096 B/op	21 allocs/op	
BenchmarkBuffer10000	10000	195268 ns/op	39304 B/op	13 allocs/op	

You can see the result that operator allocs is the number of string. Because it new allocate new string everytime when append the string. The fastest is bytes.Buffer method. In small concat we can use operator method instead of buffer method, the performance is not different.