triggerFS Documentation

Release 1.0.0 (beta)

Hasan Pekdemir

May 14, 2018

Contents

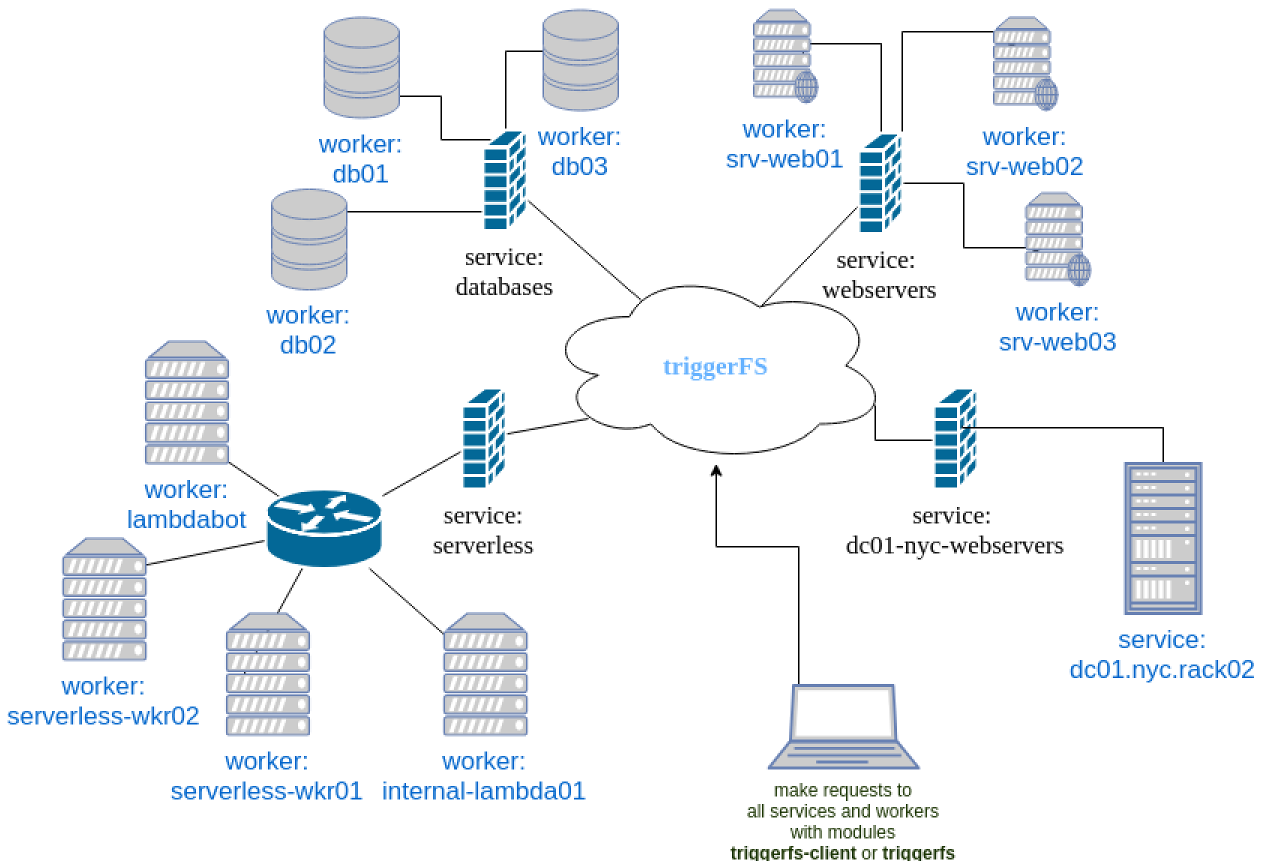
Imagine you could deploy your own customized services onto any server and map the API requests to it into regular files on your filesystem. Imagine you could build a network of workers which you can do socket communication with via files. Imagine you have a server out there which can make use of tons of plugins written in go and be able to do any kind of job and make requests to them in a service-oriented way. Imagine how basically all applications able to write to a file could magically call all of your services and trigger something.

This is triggerFS. A `triggerfs-worker` running on a server, waiting for requests and ready to process them with the help of *plugins*. Now from anywhere you can make those requests either by executing the `triggerfs-client` or by writing to regular files on your filesystem mounted anywhere with the help of the `triggerfs` module.

Check out our Use Cases page to get a short overview what triggerFS is capable of.

CHAPTER 1

Diagram



CHAPTER 2

Why triggerFS?

TriggerFS is here to help you with the right tool. We don't want to invent anything new, we want to take already existing parts and put them together to build a system which can deliver the experience many people are looking for when:

- there is a new problem to be solved
- building distributed systems can actually be tricky
- suddenly automating things becomes relevant
- communication across boundaries is necessary but firewalls are in your way
- tasks need to be outsourced to other systems
- tasks need to be scheduled
- periodic tasks need to be run
- geo-replication is a topic
- a "serverless" design is taken into consideration
- lambda functions on own servers are needed
- fire&forget-style events could be needed
- wanting a central place where everything can be run (triggered) from

2.1 Benefits

triggerFS offers you one more way to access your server or services on that server. If ssh is the door to get into your server, think about triggerFS as a door to access various, selected services on that server.

triggerFS' `fs` module also helps to enable M2M (machine-to-machine) communication on embedded devices. Since writing to files is a cheap operation, embedded devices with limited resources can do socket communication easily with triggerFS.

By inviting other triggerFS users into your team you can collaborate with each other and share resources within the team. For example: Different universities can come together and share their servers to do data crunching or data analysis.

By joining other teams you can share your services team-wide with other users of that team. This way you can make your service accessable only for a set of users.

By making your service public you can offer your service to the whole world. People can then access your services either with the triggerFS `client` module or the HTTP/broker gateway via HTTP to make requests to your service.

So let's dive into it.

2.1.1 Introduction

Attention: triggerFS has been launched and is now available with all features enabled. This means that all signups are made with the free-tier with **no limitations** and **all features enabled**. Enjoy!

Welcome to triggerFS

triggerFS is a distributed, realtime message passing and trigger system. triggerFS enables you to build distributed systems and do realtime messaging in a service-oriented fashion.

It is made up of four modules:

- worker
- cli
- client
- fs

Features

- firewall-friendly (only outbound connections being made by workers)
- build lambda functions on your own servers
- build a network of workers and services
- make use of various plugins
- fast, reliable and service-oriented networking
- high-speed, low latency and asynchronous messaging
- realtime stdout output
- cluster-enabled services
- different message passing algorithms on services: roundrobin, serial, mirror (parallel)
- messaging via regular files with the `fs` module (triggers)
- the `fs` module is available on every device (distributed, synced FUSE filesystem)
- write your own plugin
- invite others to your team and share resources with each other
- join other teams and share resources with each other

- make services public so everybody can use them

How it works

You deploy the worker module on your servers and make a request with the client module. The worker will take that request, pass it to a plugin and send the output back to the client. By specifying the plugin on the client-side you can control what plugin shall be used by the worker. You can then go ahead and group all workers by binding them to services.

These services act like a proxy to your workers and enable you to do the client requests in a load-balanced, serial or parallelized way. Services guarantee high-availability by doing a fail-over to other available workers in a group if one or more workers should fail within that group.

Here is a small list of things a plugin can do:

- compute something
- filter, grep, grok, sort or manipulate data
- write to a file
- log to a file
- forward a message
- read logs
- collect metrics (eg. send metrics to graphite)
- just echo back the received message
- monitor the server the worker is running on
- execute something on a remote machine via ssh
- run a chef/puppet/fabric recipe or ansible playbook
- run a bash script
- send an SMS via twillio
- send a text message to telegram
- flip a switch (0/1 flipflops)
- start/stop/restart a service
- kill a process
- chain requests by writing to another trigger file (this is fun)
- ...

To learn more about Plugins please have a look at Plugins.

Roadmap

- marketplace api for the plugin ecosystem
- marketplace web UI
- marketplace integration into `cli` module
- streaming services for the `worker` module (long-running plugins/services)
- listening feature for the `client` module (for streaming services)

- service broadcasting feature
- HTTP/broker gateway for making requests via HTTP
- periodic tasks via HTTP/db
- team mailboxes in `cli` for notifications from broker and triggerfs.io
- log tables for storing output of plugins. (history of stdouts)
- encrypted communication (messaging) channels (no content encryption)
- and much more…

2.1.2 Modules

worker

A worker is or has a service and receives messages (requests) from one or more clients in a safe and concurrent way and executes them by using a plugin. Deploy a worker to a bunch of servers and connect them together, build a network of services for high-speed messaging, to distribute workload or create clusters or groups of workers.

Every worker has an identity and a token to authenticate against the broker. A worker can be attached to a service. If more than one worker is attached to the same service, the service acts as a load-balancer.

You can specify other algorithms beside load-balacing. Eg. serial (distribution of tasks to workers in order) or mirror (parallelized execution of a task on all workers behind that service).

Note: After successfully authenticated, a worker's identity is referred to as a `service`. We do not distinguish between them. However, a service you define in the `cli` is a sort of reverse proxy or alias for a worker (a real service). The identity of a worker is really only a string acting as an identifier.

Just remember, that once a worker is online, it's identity becomes a service. Hence the flag `-service` in the client module. You can pass both a real service or an identity of a worker to it.

cli

The cli module is an interactive cli-based management console for managing everything on your triggerFS platform. With the cli module you can:

- create new users
- create new workers
- create new services
- make services public (public services feature)
- select the message dispatching algorithm (loadbalancing and more)
- create triggers
- make requests to your services in realtime
- and much more

The triggerFS cli module is the central place for managing, orchestrating and configuring your triggerFS environment.

Suppose we wanted to do the following: If we know that there will be a service called demo with a command plugin and three workers behind that service (announcing that service), we could create a trigger named command, attach it to the demo service, add the demo service to all of our three workers and set a timeout of - let's say - 10s. Now we have created a trigger which is represented as a file in our fs module.

client

A client sends messages (requests) to a service and gets back a response. The client can specify which service it wants to talk to and what plugin shall be used. The client module is simple but powerful.

fs

The fs module is a module for mapping the above mentioned triggers to files with the help of FUSE. It was built to go a step further than just sending messages back and forth. It enables machine-to-machine communication. Mounting files is cheap and doing socket communication by using files makes this module attractive to embedded devices or small computers like the Raspberry Pi™.

Create a directory and define a trigger in that directory in your cli. Now, if you mount the fs module to a place on your filesystem, you end up with a file in that directory within that mountpoint. Every write to that file (with the content being the data written to that file) will send a request to the workers behind the above defined service with all the predefined set of rules we configured ealier. The fs module aims to make triggerFS app-friendly in such a way that other applications can use files as their way to send a message to your services.

For example: A trigger file could be defined in such a way that the result would be a logging of the request being sent to a service. Now we could tell Nginx to log into our trigger file instead of /var/log/nginx/*. Now everytime Nginx wants to log someting, it makes a syscall (write) to our file which would result in a message being sent. Our service would then write it to eg. a central NFS server of the company which is located on the machine where the worker is running.

Another example would be a raspberry pi which collects weather data and sends it to a central server (service) by writing into the trigger-files it mounted on its filesystem. Either scripted or syscalled. A simple echo 'somedata 31F;10°;3.2' > /mnt/triggerfs/weatherstation/rpi/station1 is enough to send your data.

What we just did is, we triggered an action by writing to a file. Hence the name trigger.

Note: You can't create regular files in your mountpoint. The `fs` module only supports trigger-files. The only allowed operations are:

- `mkdir` to create directories
- `mv` to rename trigger-files
- `chmod` to set unix permissions on trigger-files

There is one more module called `broker`. This is the broker we maintain and operate in the cloud (the service behind triggerfs.io). The client/worker communication happens to be routed via the broker. The broker is the main coordinator for every message. It takes the request from the client and dispatches it to the services accordingly.

2.1.3 Security

Communication Flow

Beside our RESTful HTTP (JSON) API for database access, we use ZeroMQ for the communication between the client/worker and the broker. Every authenticated request to our API is done by using a JSON Web Token (JWT).

The communication/networking between clients and workers (services) are as follows:

- client <==> broker <==> service (worker or service)

The central broker in the cloud (we, the triggerFS team) is responsible for routing the messages back and forth. A client cannot reach a worker without the broker and vise versa.

The broker exclusively uses the JWT of the client and/or worker if it has to make some operations on behalf of either part. This means that the JWT is also being sent when a message is sent. It is part of the message.

Since a JWT in triggerFS does not include sensitive data (only metadata) it is acceptable to send a JWT over the wire. However, in future releases we want to implement channel encryption on top of the SSL/TLS HTTP API calls, so that even the zmq channels we use to communicate are also encrypted in the future.

Database

Our database is powered by postgreSQL and here is a listing of what will be stored in our db:

- users with their identity and secret (we use pgcrypto and bcrypt the password/secret before it is inserted into the db)
- workers with their identities and tokens (we use uuid_v4 for the token of a worker)
- teams with their configuration settings
- services with their configuration settings
- triggers with their configuration settings

This is everything which is stored in our db. The only sensitive data is the identity/secret of a user and identity/token of a worker and we make sure to use cryptography to secure those.

In future there will be a log table for storing the output of a plugin into the db. Which will possibly hold sensitive data. We will think about how to store those in a secure way.

Maintenances

If triggerfs.io schedules maintenances and/or the broker has to be shut down, all workers/services will automatically get notified and will reconnect as soon as the broker is up and running again.

In the future we will also notify the team with a notification message to their mailboxes (future feature), which they can access within their `cli`.

2.1.4 Pricing

triggerFS will have a three-tier plan. Here is an overview of the pricing plan:

	Free	Basic	Advanced
	1 team	1 team	2 teams
	2 workers	25 workers	100 workers
	1 service	5 services	20 services
	3 triggers	26 triggers	51 triggers
	2 users/team	9 users/team	25 users/team
	Unlimited access to marketplace		
	Join other teams		
	public services	All features enabled	
Price	free	$x/month	$x/month

- worker, service and trigger limits are per team

- the free tier will always be free

Note: We are in the beta-testing phase. We can't exactly tell the price, but we will update it once we know how we want to charge our customers. Our goal is to launch this application in a beta-testing stage so we can estimate which resources will cost us how much. Based on that calculation we will try to offer a fair price to our customers.

If you have been using this application for a while, we would like to hear your feedback. You can reach us at feedback@triggerfs.io. Thank you.

2.1.5 Target Group

We think that devops and system administrators will love to use triggerFS due to the way it simplifies building tools such as automation systems and communication of services.

We see DCs (data centers) in general also as a target group. For example: A triggerfs-worker as a top-of-the-rack (tor) worker which is responsible for the systems in a rack to handle deployments, automation, triggering of jobs, etc. is one of the scenarios triggerFS can fit into.

Systemadministrators can use triggerFS for maintenance purposes or devops engineers can build whole clusters for various deployment scenarios.

In the end, it will be the massive amount of plugins which will enable triggerFS to become something useful for any possibly imaginable task.

Of course everybody is welcome to try out triggerFS (there is a free-tier subscription. Go try it out!)

2.1.6 All-In-One-Installer

Note: Signup in less than a minute. Download all modules directly with the installer and set up your first network in less than 4 minutes.

For a complete setup of your triggerFS you need to sign up and create a team. Let's go!

2.1.7 Pre-Requirements

Despite of triggerFS being a SaaS application it comes without a website or SPA (Single Page Application) like most SaaS do. Instead, the whole application is based on cli. So it was only right to continue with this philosophy and implement the Sign-Up process into a cli installer. That is why triggerFS comes with a cli based installer.

The cli installer makes use of dialog. So make sure you have dialog installed if it does not come with your distribution by default. Also you need wget and curl.

```
apt-get install dialog wget curl
```

to install dialog on debian-based distributions.

The installer will notify you anyway if at least one of the above packages is missing.

2.1.8 The Installer

One-liner

The installer can be started with a one-liner directly from the triggerfs.io server:

```
curl https://install.triggerfs.io | bash
```

The script behind https://install.triggerfs.io is the same as the `install.sh` script in the official triggerFS packages repository at https://github.com/triggerfsio/packages.

Manual

Download install.sh from the triggerFS packages repository or clone the whole repository and run the `install.sh` script:

```
git clone https://github.com/triggerfsio/packages.git
cd packages
./install.sh
```

Signup

Signing up is really simple and takes less than a minute. Just choose "Signup" from the menu and follow the instructions of the installer. You will be notified with an email if you have successfully signed up.

Team creation

You can choose to create a team after you have successfully signed up and activated your account with the activation code which was sent to you via mail. You can also create a team in your cli module. See Configuration for more information on how to use the cli.

Download modules

To get all binaries (modules: worker, cli, client and fs) choose "Download modules" and specify a directory which must exist on your filesystem. The binaries come as statically linked. This means you don't need any dependencies on your system to be able to run the modules.

However, you need *libfuse2* for the modules `cli` and `fs` since both of them use FUSE to mount trigger-files onto your filesystem. Install it:

```
apt-get install libfuse2
```

2.1.9 Configuration File

Important thing to know is that all triggerFS modules use the same configuration file for its configuration directives. The configuration file is a toml file called `triggerfs.toml` and is searched in $HOME by default.

Get the skeleton configuration toml file from https://github.com/triggerfsio/packages:

```
wget https://raw.githubusercontent.com/triggerfsio/packages/master/triggerfs.toml
```

or copy triggerfs.toml from the git repository we previously cloned:

```
cd packages
cp triggerfs.toml ~
```

Edit configuration file

Replace your credentials in the main section of the configuration file:

```
### MAIN SECTION
[main]
# team name for login
team = "myawesometeam"
# identity for login
identity = "myidentity"
# password for login
secret = "password"
### MAIN SECTION END
```

We will mention the configuration file a few more times in the Configuration section of this documentation.

Wasn't that easy? Now you are ready to go. Go into your `triggerfs-cli` now and start configuring your first worker.

2.1.10 Get Started

Note: Here we will show how you can start with triggerFS in less than 4 minutes. We will create a *worker*, attach it to the *service*, make a request with the `triggerfs-client` and do the same request with the `triggerfs` module. We will use the *command* plugin in this tutorial.

We will start with the cli module since we have nothing configured to run the other modules. Everything configuration happens in the cli.

We will first start by creating a worker, fire up the worker, create a service and a trigger.

Note: The `cli` module's syntax is similar to Juniper's cli. The syntax and commands available in the cli are pretty much self-explanatory and its help menu will have information on what each command is doing. You can access the help menu by invoking `help` in both `run` and `configure` modes.

The cli also has autocompletion of words. So use <TAB><TAB> (like you would do in your shell) to type fast and complete worker identities, service names, etc.

In this configuration example we assume that we are logged in as the user "hp" in the team "team1". The prompt of the cli will indicate this.

Workers

First we will start with creating a worker so we can run it on a server.

Create a new worker

```
(team1) hp@example.com$ configure# set workers identity demo01
(team1) hp@example.com$ configure#
(team1) hp@example.com$ configure# show workers
  id | identity |      description       |    state     |                token         ␣
↪          |   owner   |             created             |             updated
+----+----------+------------------------+--------------+-----------------------------
↪---------+-----------+--------------------------------+-----------------------------
↪-+
   6 | demo01   |                        | DISCONNECTED | 9b1bd278-e645-4eec-bffb-
↪0587ad84534e | hp        | Mon, 30 Apr 2018 18:45:30 UTC | Mon, 30 Apr 2018␣
↪18:45:30 UTC
+----+----------+------------------------+--------------+-----------------------------
↪---------+-----------+--------------------------------+-----------------------------
↪-+
(1 rows)

Time: 545.092072ms
(team1) hp@example.com$ configure#
```

Update configuration file

Now we take the identity and token of the newly created worker and put it into our toml configuration file.

The naming of the subsection is simple: workers.$team.$identity. Then specify the token under that subsection with a key named “token”.

```
...
[workers]
# path to plugins folder
pluginspath = "/home/hp/gocode/src/github.com/triggerfsio/plugins/plugins"

# subsections of [workers] section identified by identities prefixed with "workers."
[workers.team1.demo01]
token = "9b1bd278-e645-4eec-bffb-0587ad84534e"
...
```

Hint: Workers only need the [workers] section in the config file. No need for credentials in the [main] section

Services

Until now the worker is already available under the service name “demo01”. Remember that a worker identity automatically becomes a service in triggerFS.

Since we want to make use of a trigger file, we need to go a step further and create a real service so we can attach our worker to it.

Note: trigger-files only work with real services. Workers cannot be attached to a trigger file.

Create a new service

```
(team1) hp@example.com$ configure# set services name demoservice
(team1) hp@example.com$ configure# show services
  id |    name     | description | timeout | algorithm  | triggers | workers |␣
↪visibility | owner |            created             |            updated
+----+-------------+-------------+---------+------------+----------+---------+--------
↪----+-------+--------------------------------+-------------------------------+
   2 | demoservice |             |         | roundrobin |          |         | local ␣
↪    | me     | Mon, 30 Apr 2018 18:49:02 UTC | Mon, 30 Apr 2018 18:49:02 UTC
+----+-------------+-------------+---------+------------+----------+---------+--------
↪----+-------+--------------------------------+-------------------------------+
(1 rows)

Time: 47.334023ms
```

The service is now available only to you (visibility is local by default).

Attach workers

```
(team1) hp@example.com$ configure# set services demoservice workers add demo01
Notice: this service has algorithm roundrobin. Roundrobin is handled by the broker.␣
↪Any newly added worker to this service should reannounce its services.
(team1) hp@example.com$ configure# show services
  id |    name     | description | timeout | algorithm  | triggers | workers |␣
↪visibility | owner |            created             |            updated
+----+-------------+-------------+---------+------------+----------+---------+--------
↪----+-------+--------------------------------+-------------------------------+
   2 | demoservice |             |         | roundrobin |          | demo01  | local ␣
↪    | me     | Mon, 30 Apr 2018 18:49:02 UTC | Mon, 30 Apr 2018 18:49:02 UTC
+----+-------------+-------------+---------+------------+----------+---------+--------
↪----+-------+--------------------------------+-------------------------------+
(1 rows)

Time: 51.229509ms
(team1) hp@example.com$ configure#
```

Note: Notice how the cli is telling us to let any newly added workers to reannounce their services. Since the worker we have just created never ran, it will automatically announce `demoservice` on startup.

Start the worker

Let’s start the worker on any server. Remember to deploy the toml configuration file to the server.

```
hp@localpc $ ./triggerfs-worker -identity demo01 -debug true
2018/04/30 21:03:01 I: connecting to broker at tcp://triggerfs.io:5555...
2018/04/30 21:03:01 I: trying to connect recv socket to broker
2018/04/30 21:03:01 I: trying to connect send socket to broker
2018/04/30 21:03:02 I: received BROKER_ACCEPTED from broker for worker socket -␣
↪Connected
2018/04/30 21:03:02 I: Found service demoservice. Announcing it to broker.
2018/04/30 21:03:05 I: received HEARTBEAT for worker socket from broker.
```

```
2018/04/30 21:03:07 I: received HEARTBEAT for worker socket from broker.
2018/04/30 21:03:10 I: received HEARTBEAT for worker socket from broker.
```

Notice how the worker has recognized that it is attached to the service `demoservice` and announced it to the broker. Ready to listen on it. A quick look in the cli tells us that the worker is online:

```
(team1) hp@example.com$ configure
(team1) hp@example.com$ configure# show workers
  id | identity |      description        |    state     |                 token        ␣
↪          |   owner   |             created             |             updated
+----+----------+------------------------+--------------+------------------------------
↪---------+-----------+-------------------------------+------------------------------
↪-+
   6 | demo01   |                        | ONLINE       | 9b1bd278-e645-4eec-bffb-
↪0587ad84534e | hp         | Mon, 30 Apr 2018 18:45:30 UTC | Mon, 30 Apr 2018␣
↪19:03:01 UTC
+----+----------+------------------------+--------------+------------------------------
↪---------+-----------+-------------------------------+------------------------------
↪-+
(4 rows)

Time: 33.375014ms
(team1) hp@example.com$ configure#
```

Attention: Important: Remember to have the command plugin installed on the server where the worker is running. Have a look at *Plugins* for more information.

Trigger

Create new directory

All trigger-files must be within a directory under root (/). So first, create a directory if you haven't already:

```
(team1) hp@example.com$ configure# ^D
(team1) hp@example.com$ file
>> Starting a new interactive shell
hp@localpc /tmp/triggerfs-client403096611  $ mkdir newtrigger
hp@localpc /tmp/triggerfs-client403096611  $ <CTRL+D>
(team1) hp@example.com$
```

Create new trigger

```
(team1) hp@example.com$ configure
(team1) hp@example.com$ configure# set trigger name /newtrigger/demotrigger
(team1) hp@example.com$ configure#
```

Configure trigger

Since a trigger is just a set of definitions to what shall happen if the trigger-file is being written, we need to define them first:

```
(team1) hp@example.com$ configure# set trigger demotrigger plugin command/command
(team1) hp@example.com$ configure# set trigger demotrigger service attach demoservice
(team1) hp@example.com$ configure#
```

Now we've defined that this trigger shall route the messages to the service called `demoservice` (where the worker demo01 sits behind and listens) and that the plugin command/command should be used. Note that `command/command` is the actual path to the directory where the plugin (binary) is located. Since the pluginspath in the configuration file is configured as `/home/hp/gocode/src/github.com/triggerfsio/plugins/plugins` it looks for a binary in `/home/hp/gocode/src/github.com/triggerfsio/plugins/plugins/command/` named `command`.

```
(team1) hp@example.com$ configure# show triggers
   id |     name      | description |      plugin       | hits |    owner    | visibility |␣
↪           created            |             updated
 +----+-------------+-------------+-----------------+------+-----------+------------+-
↪------------------------------+-------------------------------+
    2 | demotrigger |             | command/command |    0 | hp          | local      |␣
↪Mon, 30 Apr 2018 18:53:54 UTC | Mon, 30 Apr 2018 18:53:54 UTC
 +----+-------------+-------------+-----------------+------+-----------+------------+-
↪------------------------------+-------------------------------+
(1 rows)

Time: 207.095606ms
(team1) hp@example.com$ configure#
```

If we go back into our filesystem where the fs module has mounted our triggerFS filesystem, we will see that a new file is located under the `newtrigger` directory:

```
(team1) hp@example.com$ configure# ^D
(team1) hp@example.com$ file
>> Starting a new interactive shell
hp@localpc /tmp/triggerfs-client403096611  $ ll newtrigger/
total 512
-rw-r--r-- 1 hp hp 0 Apr 30 20:53 demotrigger
hp@localpc /tmp/triggerfs-client403096611  $ <CTRL+D>
(team1) hp@example.com$
```

Client

Make a request

Let's make a request to our new service with the client module. We will define our service, the plugin to be used and a timeout for the request. Our command will be `uptime` to get the uptime of the server:

```
hp@localpc $ ./triggerfs-client -service demoservice -plugin command/command -timeout␣
↪10s -command uptime
2018/04/30 21:14:06 Sending message to service demoservice (roundrobin)
[command/command@demo01]  21:14:07 up  8:26,  7 users,  load average: 0.61, 0.60, 0.52

Exit code: 0
Total messages: 1
Time ran: 907.914622ms

hp@localpc $
```

The response came from the server with the worker running on called `demo01` and the plugin `command/command` and the output of the `uptime` command.

Hint: The client module also reads stdin, so you can skip the `-command` flag and echo uptime piped to the client:

```
hp@localpc $ echo uptime | ./triggerfs-client -service demoservice -plugin command/
↪command -timeout 10s
2018/04/30 21:14:06 Sending message to service demoservice (roundrobin)
[command/command@demo01]  21:14:07 up  8:26,  7 users,  load average: 0.61, 0.60, 0.52

Exit code: 0
Total messages: 1
Time ran: 907.914622ms

hp@localpc $
```

FS

Now, since we have set up a trigger we can also use the fs module to write to a real file.

Run

First we start the module which will always run in the foreground (there is no background mode currently):

```
hp@localpc $ ./triggerfs
triggerfs (v1.0.0)

*********************************************************************************
*** Welcome to triggerFS. A realtime messaging and distributed trigger system. ***
*********************************************************************************

2018/04/30 21:20:33 === triggerfs module started ===
2018/04/30 21:20:33 No JWT provided. Authenticating with login credentials in config.
2018/04/30 21:20:34 Successful login. JWT is eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
2018/04/30 21:20:34 Successful login.

2018/04/30 21:20:34 Started triggerfs.
2018/04/30 21:20:34 Serving filesystem in ./mountpoint
2018/04/30 21:20:34 Log file is ./triggerfs.log
2018/04/30 21:20:34 Ready and running in foreground...
```

The mountpoint in this case is the directory called `mountpoint` in $PWD (set in the configuration toml file as ./mountpoint).

Execute (write to trigger-file)

Now in another terminal we can go into that directory and write to the trigger-file:

```
hp@localpc $ ll mountpoint/
total 512
drwxrwxr-x 1 hp hp 0 Apr 30 20:52 newtrigger/
hp@localpc $ ll mountpoint/newtrigger/
```

```
total 512
-rw-r--r-- 1 hp hp 0 Apr 30 20:53 demotrigger
hp@localpc $ echo uptime > mountpoint/newtrigger/demotrigger
hp@localpc $
```

Since we cannot write into stdout in FUSE (except we have read from a file) the output (response) of this request will be displayed in the terminal where the fs module is running in foreground.

If a logfile was specified in the configuration file for the `[triggerfs]` section then the response can be found there as well.

A look at the trigger in our cli will show that it got one hit:

```
(team1) hp@example.com$ configure# show triggers
  id |    name     | description |     plugin      | hits |   owner    | visibility | ␣
↪          created             |            updated
+----+-------------+-------------+-----------------+------+-----------+------------+--
↪------------------------------+--------------------------------+
   2 | demotrigger |             | command/command |    1 | hp        | local      |␣
↪Mon, 30 Apr 2018 18:53:54 UTC | Mon, 30 Apr 2018 19:22:52 UTC
(1 rows)

Time: 177.43962ms
(team1) hp@example.com$ configure#
```

in the `hits` column.

This was one configuration flow in its simplest form for a complete setup of a trigger.

We have created a worker, bound it to a service, created a trigger with a specified set of rules and executed a request in both ways with the client and the fs module.

Plugins

Plugins are binaries written and compiled in go. Since triggerFS uses zmq for its socket communication, you need to install libzmq3.

This is only necessary if you want to build the plugin yourself. The triggerFS marketplace later will have binaries pre-built (statically linked, so you don't need any dependencies) for you (beside the available source code of the plugin).

The triggerFS core plugins are available at https://github.com/triggerfsio/plugins.

Go get them with `go get`:

```
go get github.com/triggerfsio/plugins
```

Note: The core plugins currently come without a pre-built binary. We will save this feature for later when we have launched the marketplace.

For now, you **have to** build the core plugins yourself. If you are experienced with docker, you can also make use of the golang docker file which comes with golang already installed. In this example we will assume that you have installed golang and libzmq3-dev on your machine.

Dependencies

If you want to build a plugin you need to have installed libzmq3 and its developer files. Install it with:

```
apt-get install libzmq3-dev
```

Build

Now switch to the folder of the plugin you want to build:

```
hp@localpc ~ $ cd gocode/src/github.com/triggerfsio/plugins/
hp@localpc ~/gocode/src/github.com/triggerfsio/plugins $ cd plugins/command/
hp@localpc ~/gocode/src/github.com/triggerfsio/plugins/plugins/command $ go build␣
↪command.go
hp@localpc ~/gocode/src/github.com/triggerfsio/plugins/plugins/command $ ll
total 8.2M
-rwxrwxr-x 1 hp hp 8.2M Apr 30 21:33 command*
-rw-rw-r-- 1 hp hp 2.3K Apr  4 00:58 command.go
hp@localpc ~/gocode/src/github.com/triggerfsio/plugins/plugins/command $
```

Now you can point your plugins folder to this directory (in your toml configuration file under section `[workers]`):

```
### WORKERS SECTION
[workers]
# path to plugins folder
pluginspath = "/home/hp/gocode/src/github.com/triggerfsio/plugins/plugins"

# subsections of [workers] section identified by identities prefixed with "workers."
[workers.team1.demo01]
token = '9b1bd278-e645-4eec-bffb-0587ad84534e'
...
```

The command binary will be ready for use now.

2.1.11 What is a Plugin

A plugin is a tiny binary (written and compiled in go) which does one thing and does it well. A plugin could be anything:

- compute something
- filter, grep, grok, sort or manipulate data
- write to a file
- log to a file
- forward a message
- read logs
- collect metrics (eg. send metrics to graphite)
- just echo back the received message
- monitor the server the worker is running on
- execute something on a remote machine via ssh

- run a chef/puppet/fabric recipe or ansible playbook
- run a bash script
- send an SMS via twillio
- send a text message to telegram
- flip a switch (0/1 flipflops)
- start/stop/restart a service
- kill a process
- chain requests by writing to another trigger file (this is fun)
- . . .

you name it. As you can see the job of a plugin is to make our life easier and serve us with mostly things that can be automated or are needed on-demand. You can write your own plugins or search for existing plugins on the marketplace (more about that below).

2.1.12 Core Plugins

The core plugins are the official plugins written and maintained by the triggerFS team. They are well tested and known to be bug-free and working.

We will include more and more plugins into the core plugins repository, so we can have a nice set of plugins to work with.

We hope that the rest will be done by the community with their great ideas and great plugins shared to the public.

Please refer to Configuration#Plugins to see how to get and configure the triggerFS core plugins.

2.1.13 Custom Plugins

In this section you will learn how to write your own plugins and build them so your workers can use them.

Plugins are written in go. You will need golang (https://golang.org) to be able to write plugins.

Writing plugins should be as easy as possible. That's why we offer you a skeleton that you can copy and paste into your editor and start writing your plugin from there.

Let's start with copying the skeleton plugin code available in the official triggerFS plugins repository at https://github.com/triggerfsio/plugins.

If you haven't installed go please do so first.

1. Go get the repository with `go get`:

```
go get github.com/triggerfsio/plugins
```

2. Copy the skeleton plugin code to a new project (directory) and open it in your favorite editor:

```
cd $GOPATH/src/github.com/triggerfsio/plugins
cp skeleton/plugin.go ~/mynewplugin/main.go
```

3. Write your own function in the `Init()` method of the plugin.

Let's say we want to write the payload to a file and return the message "file has been written." back to the client. We also want to notify the client in realtime that we are about to write to that file.

```
// Init implements the triggerfs plugin Interface
func (ap *AwesomePlugin) Init(message *plugins.Message, resp *plugins.Response) error
↪{
        // open a channel for realtime communication back to the client.
        err := ap.Plugin.Open(message.Socket)
        if err != nil {
                return err
        }
        // IMPORTANT: remember to defer close the channel or you will get timeouts!
        defer ap.Plugin.Close()

        data := message.Command[0]
        filepath := message.Args["filepath"]

        var output string
        // now implement your plugin here
        ap.Plugin.Send(fmt.Sprintf("Writing payload: %s to file %s now", data,␣
↪filepath))
        err = ioutil.WriteFile(filepath, []byte(data+"\n"), 0644)
        if err != nil {
                log.Printf("failed to write to file: %s\n", err)
                ap.Plugin.Send("failed to write to file.")
                output = "failed to write to file."
                return err
        }

        output = "file has been written."

        // and finally set the exitcode and a final message on resp
        resp.ExitCode = 0
        resp.Output = []string{output}

        return nil
}
```

We have defined data and filepath here. data is the command being sent by the client with the `-command` flag or stdin and filepath is the plugin argument specified by the client. We then notify the client that we will write now. Then we do the actual writing to the file. And finally we return from the function after we set our response output (that the file has been written).

4. Build your plugin with go. Optionally give it an output name `myplugin`:

```
cd ~/mynewplugin
go build main.go -o myplugin
```

5. Set your pluginspath in the configuration toml file:

```
pluginspath = "/home/hp/mynewplugin"
```

Note: You must use absolute paths in your toml file. $HOME/mynewplugin would not work here.

6. Make a request to the service and specify your own plugin:

```
$ ./triggerfs-client -service hp01 -plugin myplugin -timeout 3s -command "hello world
↪" -args filepath=/tmp/myfile.txt
2018/05/02 00:52:48 Sending message to service hp01
```

```
[myplugin@hp01] Writing payload: hello world to file /tmp/myfile.txt now
[myplugin@hp01] file has been written.

Exit code: 0
Total messages: 2
Time ran: 118.079784ms

$
```

If we cat the file /tmp/myfile.txt we see the following:

```
$ cat /tmp/myfile.txt
hello world
$
```

Congratulations! You have just written your first triggerFS plugin. Of course this one was really simple. A plugin can vary from simple to super-complex stuff. That's why plugins enable you to do so many things.

2.1.14 Introduction

You can do a lot with triggerFS. That's why we want to give you some examples and use cases. Hopefully you will find enough information here to use triggerFS the way it fits your needs.

We do have a screencast where we show a complete set up of two workers with one service in front of it and task execution with the help of the *command* core plugin.

2.1.15 Media

Use Cases

2.1.16 Scenario 1 - Logging

You have installed and set up Graylog as a log-aggregator and want to log to it with the GELF format whenever somebody hits your nginx webserver. You define a path in your nginx configuration to log to a trigger-file in your mountpoint anywhere on your server where the webserver is runing.

You use a plugin which takes data, packs it into a GELF format and sends it to the graylog server. Now whenever nginx logs a hit to your website, a message with the log content will be logged to graylog.

2.1.17 Scenario 2 - Embedded devices

You have a Raspberry Pi running somewhere outdoors which measures the outside temperature. You define a trigger which sends a message to a specific service with the appropriate plugin. The plugin takes the temperature and stores it into your MySQL database. Now everytime your Raspberry Pi measures the temperature it writes to a trigger-file mounted on its filesystem and the temperature ends up in the MySQL database.

2.1.18 Scenario 3 - Lambda functions

You want to use a lambda function in a serverless way. Your application code reaches a point where it has to send an SMS to a user. You make use of the client or the HTTP/broker gateway to send a request to your service and use a plugin which uses twilio as an SMS provider. You provide your Twilio api key and api secret with the `-args` flag of the client module and a string. The call returns with either a success or an error and your application code continues accordingly.

2.1.19 Scenario 4 - A call back service

You have set up asterisk as your PBX server. You have set up a service which uses a plugin that registers users by their phone numbers and whenever somebody requests your service you call them back with your PBX server.

2.1.20 Scenario 5 - Chaining requests

You want to relay messages from one server to another. You set up a service which uses a plugin that takes a request and writes it directly to a trigger-file mounted on the server where the worker is running. That trigger-file is defined in such a way that every write to that file will trigger a request to another service, and so on. Now whenever someone sends a message to server A it ends up being received on server B.

The list goes on and on. The opportunities to use triggerFS are countless. Feel free to come up with your own ideas on how to use triggerFS. And also feel free to tell us your story on how you use triggerFS in your environment. Drop us an email at feedback@triggerfs.io.

Examples

- Mass System Upgrades Given we have 100 servers with Ubuntu/Debian, we deploy one worker on each of them and bind them to multiple services (eg. divided by location/rack/dc). We use the *command* plugin to upgrade the systems:

  ```
  ./triggerfs-client -service dc1 -timeout 1h -plugin command -command 'apt-get␣
  ↪update && apt-get upgrade'
  ```

- Telegram Bot Given we have a telegram group with a bot inside, we deploy one worker on a server/vps and mount our trigger files on the same server with the `triggerfs` module. We write a telegram plugin to listen on that group and whenever a message is sent to the bot we write into a trigger-file which would result in sending a message to a certain service. We send back a message to the group that the file has been written.

- Suppose Twilio would be using triggerFS. Given that the Twilio team offers a public service to registered users for sending SMS: We have an account at Twilio and want to send an SMS to a friend.

  We want to use this public service and invoke the following command:

  ```
  ./triggerfs-client -service twilio/api -plugin sms -command 'Hey buddy, can we␣
  ↪meet at 3pm today?' -args recipient=0123456789 -args apikey=$MYAPIKEY_BASH_ENV -
  ↪args apisecret=$MYAPISECRET_BASH_ENV
  ```

  Note the service syntax for public services here. We specify the team name and the service name separated by a slash.

2.1.21 Home For Plugins

triggerFS gives you the platform for high-speed realtime messaging. But that is only a small fraction of what this is all about, really. What really powers triggerFS is the gigantic amount of plugins you can use to do any kind of task. These plugins or the combination of several of them is what makes triggerFS a pleasure to work with.

Those plugins need a place where they can be listed to the public, so everyone can benefit and download and use them. That is our vision of building the triggerFS marketplace. Anyone who can write go (https://golang.org/) can also write plugins. Check out our core plugins so you can get a feeling of how a plugin works. Our plugin skeleton has 57 lines of code (loc). Add your own function to it and you have just created your first own plugin.

Plugins are small pieces of code, doing one thing and doing it well. We want to build as many plugins as possible for any kind of job. You can host them on github and link your repository in the triggerFS marketplace, making it accessible within the triggerFS app. Our goal is to provide you with tons of plugins with the help of the community in the near future.

Note: The marketplace is in development right now and will be launched soon.

2.1.22 How The Marketplace Works

It is really simple. If you have written a plugin and you think it's worth to be used by everyone you can share it with others on the triggerFS marketplace:

- upload your plugin to github
- provide a meta json file and a README.md
- go to the marketplace and add a new plugin
- provide the URL to your github repository of that plugin
- we will fetch the informations and metadata to your repository
- we will display your plugin on the marketplace
- we will test (a real test) your plugin by building it first and then run an internal backend worker with that plugin
- we will then make a request to that worker with the "invocation" example you give in the meta json
- if everything is ok we will build a statically linked binary (no dependencies needed, such as zmq) and offer a download of the binary
- people still can choose to use the source code and build it by themselves

2.1.23 Community

We hope that any time soon the community will offer more and more plugins, because hopefully everyone will write plugins and will love to share them. We want to inspire the community with own work of the core plugins. Whenever we have time to build a plugin, we write and test it so we can put them into the core plugins repository.

2.1.24 A Free Marketplace

The marketplace is mainly free to use. At least this is what we think the marketplace should look like. However, we think it can be beneficial for companies to offer their own plugins (think of proprietary plugins) for money. Or other users who spent a big amount of time on a plugin and want to monetize their idea/work. That's why we are thinking about giving the opportunity to people to place their plugin and charge users. How to charge and when is still in theory. For now we don't spend too much time on this idea. We can't say for sure if and when this idea will be introduced to the triggerFS marketplace.