Kapow! Documentation

Release 0.7.0

BBVA Innovation Labs

Mar 12, 2021

Contents

CHAPTER 1

The Project

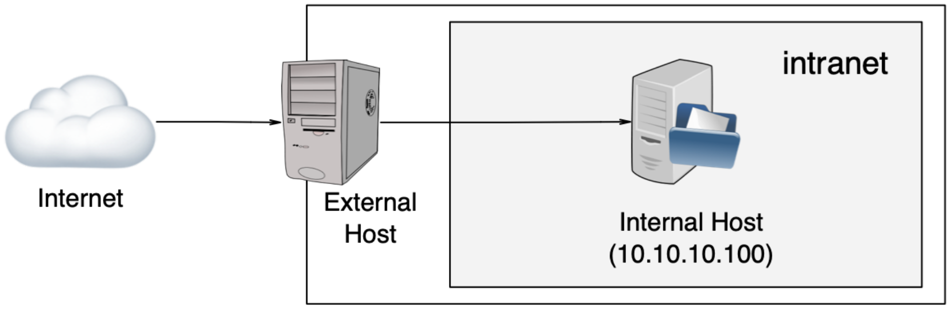
This section will introduce you to *Kapow!* basics.

1.1 Quick Start Guide

We'll explain a simple example to help you understand what *Kapow!* can do and why it is so awesome :-).

1.1.1 Scenario

In this example we'll consider that our scenario is a corporate network like this:



Our organization has an external host that acts as a bridge between our intranet an the public Internet.

Goal

Our team must be able to check if the **Internal Host** is alive on an ongoing basis.

1.1.2 Limitations and Constraints

1. We don’t want to grant access to the **External Host** to anybody.
2. We don’t want to manage VPNs or any similar solutions to access **Internal Host** from the Internet.
3. We want to limit the actions that a user can perform in our intranet while it is checking if **Internal Host** is alive.
4. We want to use the most standard mechanism. Easy to use and automate.
5. We don’t have a budget to invest in a custom solution.

1.1.3 The Desired Solution

After analyzing the problem and with our goal in mind, we conclude that it is enough to use a simple **`ping`** to **Internal Host**.

So, the next step is to analyze how to perform the **`ping`**.

Accessing via SSH to External Host

If we choose this option, then, for every person that needs to check the status of **Internal Host** we need to create a user in the **External Host** and grant them `ssh` access.

Verdict

This is **not a good idea**, because:

1. We’d need to manage users (violates a constraint).
2. We’d need to grant users access to a host (violates a constraint).
3. We would not be able to control what options the user could provide to **`ping`** (violates a constraint).

Develop and Deploy a Custom Solution

OK, this approach could be the best choice for our organization, but:

1. We’d need to start a new project, develop, test, manage and maintain it.
2. We’d need to wait for for the development to be production ready.
3. We’d need a budget. Even if we have developers in our organization, their time it’s not free.

Verdict

This is **not a good idea**, because:

1. We’d need to spend money (violates a constraint).
2. We’d need to spend time (and time is money, see reason #1).

Using *Kapow!* (spoiler: it's the winner!)

OK, let's analyze *Kapow!* and check if it is compatible with our constraints:

1. *Kapow!* is Open Source, so it's also **free as in beer**.
2. By using *Kapow!* we don't need to code our own solution, so **we don't have to waste time**.
3. By using *Kapow!* we can run any command in the **External Host**, limiting the command parameters, so **it's safe**.
4. By using *Kapow!* we can launch any system command as an `HTTP API` easily, so **we don't need to grant login access to External Host to anybody**.

Verdict

Kapow! is the **best** choice, because it satisfies all of our requirements.

1.1.4 Using *Kapow!*

In order to get our *example scenario* working we need to follow the steps below.

Install *Kapow!*

Follow the *installation instructions*.

Write an Init Program `ping-route`

Kapow! uses init programs/scripts where the endpoints you want to expose are defined.

For each endpoint, you can decide which commands get executed.

For our example we need a file like this:

```
$ chmod +x ping-route
$ cat ping-route
#!/usr/bin/env sh
kapow route add /ping -c 'ping -c 1 10.10.10.100 | kapow set /response/body'
```

Let's dissect this beast piece by piece:

1. `#!/usr/bin/env sh` - shebang line so that the kernel knows which interpreter to use
2. `kapow route add /ping` - adds a new `HTTP API` endpoint at `/ping` path in the *Kapow!* server. You have to use the `GET` method to invoke the endpoint.
3. `-c` - after this parameter, we write the system command that *Kapow!* will run each time the endpoint is invoked.
4. `ping -c 1 10.10.10.100` - sends one `ICMP ping packet` to the **Internal Host**.
5. `| kapow set /response/body` - writes the output of **`ping`** to the body of the response, so you can see it.

Launch the Service

At this point, we only need to launch **`kapow`** with our `ping-route`:

```
$ kapow server ping-route
```

Kapow! can expose the user interface through HTTPS, to do this provide the corresponding key and certificates chain paths at startup:

```
$ kapow server --keyfile path/to/keyfile --certfile path/to/certfile ping-route
```

Consume the Service

Now we can call our newly created endpoint by using our favorite `HTTP` client. In this example we're using **`curl`**:

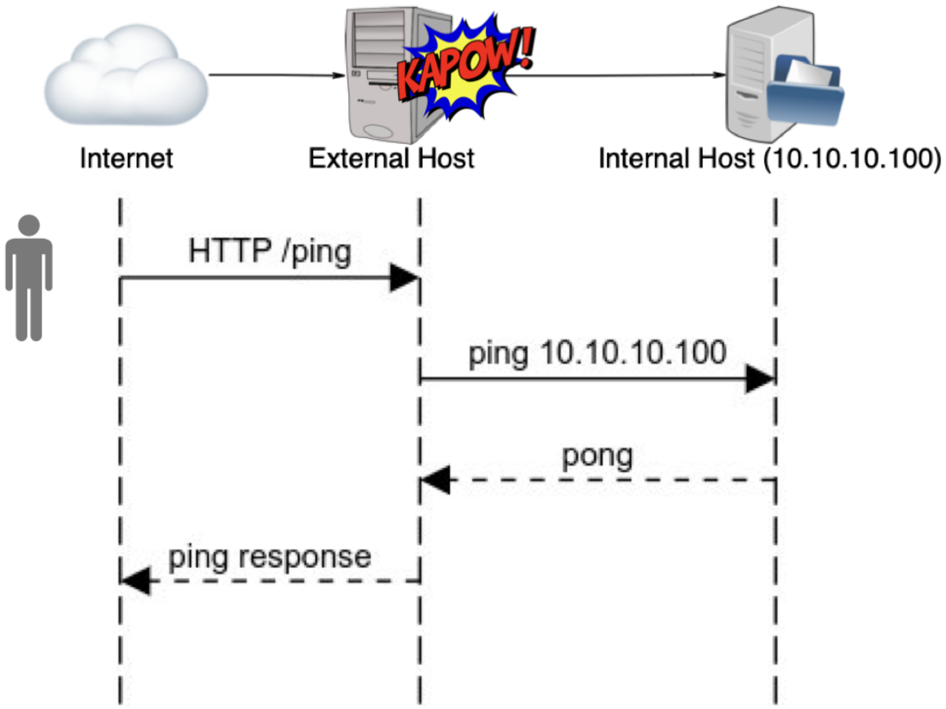
```
$ curl http://external.host/ping
PING 10.10.100 (10.10.100): 56 data bytes
64 bytes from 10.10.100: icmp_seq=0 ttl=55 time=1.425 ms
```

et voilà !, if you're using HTTPS don't forget to provide the CA certificate if needed:

```
$ curl --cacert path/to/CAfile https://external.host/ping
PING 10.10.100 (10.10.100): 56 data bytes
64 bytes from 10.10.100: icmp_seq=0 ttl=55 time=1.425 ms
```

Under the Hood

To understand what's happening under the hood with *Kapow!* let's see the following diagram:



As you can see, *Kapow!* provides the necessary *mojo* to turn a **system command** into an `HTTP API`.

1.2 Security Concerns Working with Parameters

Special care has to be taken when using parameters provided by the user when composing command line invocations.

Sanitizing user input is not a new problem, but in the case of *Kapow!*, we have to take into account also the way that the shell parses its arguments, as well as the way the command itself interprets them, in order to get it right.

> **Warning:** It is **imperative** that the user input is sanitized properly if we are going feed it as a parameter to a command line program.

1.2.1 Parameter Injection Attacks

When you resolve variable values be careful to tokenize correctly by using double quotes. Otherwise you could be vulnerable to **parameter injection attacks**.

This example is VULNERABLE to parameter injection

In this example, an attacker can inject arbitrary parameters to **ls**.

```
$ cat command-injection
#!/usr/bin/env sh
kapow route add '/vulnerable/{value}' - <<-'EOF'
     ls $(kapow get /request/matches/value) | kapow set /response/body
EOF
```

Exploiting using **curl**:

```
$ curl http://localhost:8080/vulnerable/-lai%20hello
```

This example is NOT VULNERABLE to parameter injection

Note how we add double quotes when we recover *value* data from the request:

```
$ cat command-injection
#!/usr/bin/env sh
kapow route add '/not-vulnerable/{value}' - <<-'EOF'
     ls -- "$(kapow get /request/matches/value)" | kapow set /response/body
EOF
```

> **Warning:** Quotes around parameters only protect against the injection of additional arguments, but not against turning a non-option into option or vice-versa. Note that for many commands we can leverage double-dash to signal the end of the options. See the “Security Concern” section on the docs.

1.2.2 Parameter Mangling Attacks

In order to understand what we mean by parameter mangling let’s consider the following route:

```
#!/bin/sh
kapow route add /find -c <<-'EOF'
       BASEPATH=$(kapow get /request/params/path)
       find "$BASEPATH" | kapow set /response/body
EOF
```

The expected use for this endpoint is something like this:

```
$ curl http://kapow-host/find?path=/tmp
/tmp
/tmp/.X0-lock
/tmp/.Test-unix
/tmp/.font-unix
/tmp/.XIM-unix
/tmp/.ICE-unix
/tmp/.X11-unix
/tmp/.X11-unix/X0
```

Let's suppose that a malicious attacker gets access to this service and makes this request:

```
$ curl http://kapow-host/find?path=-delete
```

Let's see what happens:

The command that will eventually be executed by **bash** is:

```
find -delete | kapow set /response/body
```

This will *silently delete all the files below the current directory*, no questions asked. Probably not what you expected.

This happens because **find** has the last word on how to interpret its arguments. For **find**, the argument `-delete` is not a path.

Let's see how we can handle this particular case:

```
#!/bin/sh
kapow route add /find -c <<-'EOF'
	USERINPUT=$(kapow get /request/params/path)
	BASEPATH=$(dirname -- "$USERINPUT")/$(basename -- "$USERINPUT")
	find "$BASEPATH" | kapow set /response/body
EOF
```

Note: Since this is critical for keeping your *Kapow!* services secure, we are working on a way to make this more transparent and safe, while at the same time keeping it *Kapowy*.

1.3 Installing *Kapow!*

Kapow! has a reference implementation in `Go` that is under active development right now. If you want to start using *Kapow!* you can choose from several options.

1.3.1 Download and Install a Binary

Binaries for several platforms are available from the releases section, visit the latest release page and download the binary corresponding to the platfom and architecture you want to install *Kapow!* in.

GNU/Linux

Install the downloaded binary using the following command as a privileged user.

```
# install kapow_0.7.0_linux_amd64 /usr/local/bin/kapow
```

macOS

Install the downloaded binary using the following command as a privileged user.

```
# install kapow_0.7.0_darwin_amd64 /usr/local/bin/kapow
```

Windows®

Native windows installers are provided for both 32 and 64 bit versions.

Alternatively, you can download the provided zipfile, expand it, and move `kapow.exe` to a directory of your choice; then update the system `PATH` variable to include that directory. Note that the zipfile also includes the LICENSE.

1.3.2 Install the package with `go get`

If you already have installed and configured the **`go`** runtime in the host where you want to run *Kapow!*, simply run:

```
$ go get -v github.com/BBVA/kapow
```

Note that *Kapow!* leverages Go modules, so you can target specific releases:

```
$ GO111MODULE=on go get -v github.com/BBVA/kapow@v0.7.0
go: finding github.com v0.7.0
go: finding github.com/BBVA v0.7.0
go: finding github.com/BBVA/kapow v0.7.0
go: downloading github.com/BBVA/kapow v0.7.0
go: extracting github.com/BBVA/kapow v0.7.0
github.com/google/shlex
github.com/google/uuid
github.com/spf13/pflag
github.com/BBVA/kapow/internal/server/httperror
github.com/BBVA/kapow/internal/http
github.com/BBVA/kapow/internal/server/model
github.com/BBVA/kapow/internal/client
github.com/gorilla/mux
github.com/BBVA/kapow/internal/server/user/spawn
github.com/BBVA/kapow/internal/server/data
github.com/BBVA/kapow/internal/server/user/mux
github.com/BBVA/kapow/internal/server/user
github.com/BBVA/kapow/internal/server/control
github.com/spf13/cobra
github.com/BBVA/kapow/internal/server
github.com/BBVA/kapow/internal/cmd
github.com/BBVA/kapow
```

1.3.3 Include *Kapow!* in your Container Image

If you want to include *Kapow!* in a `Docker` image, you can add the binary directly from the releases section. Below is an example `Dockerfile` that includes *Kapow!*.

```
FROM debian:stable-slim

ADD https://github.com/BBVA/kapow/releases/download/v0.7.0/kapow_0.7.0_linux_amd64 /usr/bin/kapow

RUN chmod 755 /usr/bin/kapow

ENTRYPOINT ["/usr/bin/kapow"]
```

If the container is intended for running the server and you want to dinamically configure it, remember to include a `--control-bind` param with an external bind address (e.g., `0.0.0.0`) and to map all the needed ports in order to get access to the control interface.

After building the image you can run the container with:

```
$ docker run --rm -i -p 8080:8080 -v $(pwd)/whatever-route:/opt/whatever-route kapow:latest server /opt/whatever-route
```

With the `-v` parameter we map a local file into the container's filesystem so we can use it to configure our *Kapow!* server on startup.

CHAPTER 2

Kapow! Tutorial

This tutorial will help you get more familiar with *Kapow!*.

It tells you the story of a junior ops person landing a new job in a small company. Teaming up with an experienced senior, they'll face many challenges of increasing difficulty. With *Kapow!* at their side, they will be able to pass all the hurdles.

You just need to follow the steps and execute the code shown in the tutorial to learn *the Kapow! way*.

Enjoy the ride!

2.1 Your First Day at Work

Senior

Welcome to *ACME Inc.* This is your first day here, right?

Junior

Hi! Yes! And I am eager to start working. What will be my first task?

Senior

First, let me help you get acquainted with our infrastructure.

Junior

OK.

Senior

We have two Linux boxes that provide services to our employees.

1. The Corporate Server: Provides email, database and web services.
2. The Backup Server: It is used to store backup of the important company data.

Junior

That's it? OK, just like Google, then.

Senior

Smartass…

Junior

(chuckles nervously).

Senior

Well, I think is time for you to start with your first task. It just so happens that we received another request to backup the database from the projects team.

2.2 Let's Backup that Database!

Junior

A Backup? Don't you have this kind of things already automated?

Senior

Well, is not that simple. We of course have periodic backups. But, our project team ask us for a backup every time a project is finished.

I've already prepared a script to do the task. Before executing it in production, why don't you `download it` and test it in your laptop?

```
$ curl --output backup_db.sh https://raw.githubusercontent.com/BBVA/kapow/
↪master/docs/source/tutorial/materials/backup_db.sh
$ chmod u+x backup_db.sh
```

Junior

(after a few minutes)

OK, done! I just run it and I got this output:

```
$ ./backup_db.sh
Backup done!
Your log file is at /tmp/backup_db.log
```

Senior

That's right. That script performed the backup and stored it into the **Backup Server** and appended some information into the backup log file at `/tmp/backup_db.log`.

Now you can **`ssh`** into the **Corporate Server** and make the real backup.

Junior

Wait, wait… how long have you been doing this?

Senior

This procedure was already here when I arrived.

Junior

And why don't they do it themselves? I mean, what do you contribute to the process?

Senior

I am the only one allowed to **ssh** into the **Corporate Server**, for obvious reasons.

Junior

Why do you need to **ssh** in the first place? Couldn't it be done without **ssh**?

Senior

Actually, it could be done with a promising new tool I've just found... *Kapow!*

Is a tool that allows you to publish scripts as `HTTP` services. If we use it here we can give them the ability to do the backup whenever they want.

Junior

Sounds like less work for me. I like it!

Senior

OK then, let's it try on your laptop first.

First of all, you have to follow the *installation instructions*.

Junior

I've just installed it in my laptop, but I don't understand how all of this is going to work.

Senior

Don't worry, it is pretty easy. Basically we will provide an `HTTP` endpoint managed by *Kapow!* at the **Corporate Server**; when the project team wants to perform a backup they only need to call the endpoint and *Kapow!* will call the backup script.

Junior

It seems pretty easy. How can I create the endpoint?

Senior

First you have to start a fresh server. Please run this in your laptop:

```
$ kapow server
```

> **Warning:** It is important that you run this command in the same directory in which you downloaded `backup_db.sh`.

Junior

Done! But it doesn't seem to do anything...

Senior

Now you have the port `8080` open, but no endpoints have been defined yet. To define our endpoint you have to run this in another terminal:

```
$ kapow route add -X PUT /db/backup -e ./backup_db.sh
```

This will create an endpoint accessible via `http://localhost:8080/db/backup`. This endpoint has to be invoked with the `PUT` method to prevent accidental calls.

Junior

Cool! Do we need to do all this stuff every time we start the **Corporate Server**?

Senior

Not at all. The creators of *Kapow!* have thought of everything. You can put all your route definitions on init programs, which can be shell scripts, and pass them to the server on startup.

It should look something like:

```
$ cat backup-route
#!/usr/bin/env sh
kapow route add -X PUT /db/backup -e ./backup_db.sh
```

And then you can start *Kapow!* with it:

```
$ kapow server backup-route
```

Junior

Great! Now it says:

```
$ kapow server backup-route
2019/11/26 11:40:01 Running init program: "backup-route"
{"id":"19bb4ac7-1039-11ea-aa00-106530610c4d","method":"PUT","url_pattern":"/
↪db/backup","entrypoint":"./backup_db.sh","command":"","index":0}
2019/11/26 11:40:01 Done running init program: "backup-route"
```

I understand that this is proof that we have the endpoint available.

Senior

That appears to be the case, but we better check it.

Call it with **`curl`**:

```
$ curl -X PUT http://localhost:8080/db/backup
```

Junior

Yay! I can see the log file at `/tmp/backup_db.log`

Senior

That's great. I am going to install all this in the **Corporate Server** and forget about the old procedure.

That enough for your first day! Go home now and get some rest.

2.3 What have we done?

Senior

Hey, I come from seeing our project team mates. They're delighted with their new toy, but they miss something.

I forgot to tell you that after the backup is run they need to review the log file to check that everything went OK.

Junior

Makes sense. Do you think that *Kapow!* can help with this? I have the feeling that this is the right way to go about it…

Senior

Sure! Let's take a look at the documentation to see how we can tweak the logic of the request.

Junior

Got it! There're a *lot of resources to work with*. I see that we can write to the response. Do you think this will work for us?

Senior

Yeah, the team is used to **cat** the log file contents to see what happened in the last execution:

```
$ cat /tmp/backup_db.log
```

I've made it easy for you. Are you up to it?

Junior

Let me try add this to our init program:

```
#!/usr/bin/env sh
kapow route add /db/backup_logs -c 'cat /tmp/backup_db.log | kapow set /
↪response/body'
```

Senior

Looks good to me, clean and simple, and it is a very good idea to use `GET` here as it won't change anything in the server. Let's restart *Kapow!* and try it.

Junior

Wooow! I get back the content of the file. If they liked the first one they're going to loooove this.

Senior

Agreed. And with this, I think we are done for the day…

2.4 We need to filter

Senior

Hiya! How're you doing this morning? I've got a new challenge from our grateful mates.

As time goes on from the last log rotation, the size of the log file gets bigger and bigger. Furthermore, they want to limit the output of the file to pick only some records, and only from the end of the file. We need to do something to help them as they are wasting a lot of time reviewing the output.

Junior

I have a feeling that this is going to entail some serious *bash-foo*. What do you think?

Senior

Sure! But in addition to some good shell plumbing we're going to squeeze *Kapow!*'s superpowers a litle bit more to get a really good solution.

Can you take a look at *Kapow!*'s documentation to see if something can be done?

Junior

I've read in the documentation that there is a way to get access to the data coming in the request. Do you think we can use this to let them choose how to do the filtering?

Senior

Sounds great! How have we lived without *Kapow!* all this time?

As they requested, we can offer them a parameter to filter the registers they want to pick, and another parameter to limit the output size in lines.

Junior

Sounds about right. Now we have to make some modifications to our last endpoint definition to add this new feature. Let's get cracking!

Senior

Well, we got it again, this is exactly what they need:

```
kapow route add /db/backup_logs -c 'grep -- "$(kapow get /request/params/
↪filter)" /tmp/backup_db.log \
  | tail -n "$(kapow get /request/params/lines)" \
  | kapow set /response/body'
```

It looks a bit weird, but we'll have time to revise the style later. Please make some tests on your laptop before we publish it on the **Corporate Server**. Remember to send them an example URL with the parameters they can use to filter and limit the amount of lines they get.

Junior

OK, should look like this, doesn't it?

```
$ curl 'http://localhost:8080/db/backup_logs?filter=rows%20inserted&lines=200
↪'
```

Senior

Exactly. Another great day helping the company advance. Let's go grab a beer to celebrate!

2.5 I Need my Report!

Junior

Good morning!

You look very busy, what's going on?

Senior

I am finishing the capacity planning report. Let me just mail it. . . Done!

Today I am going to teach you how to write this report so we can split the workload.

Junior

Oh. That sounds. . . fun. OK, tell me about this report.

Senior

Here at ACME Inc. we take capacity planning seriously. It is important that our employees always have the best resources to accomplish their job.

We prepare a report with some statistics about the load of our servers. This way we know when we have to buy another one.

Junior

I see this company scales up just like Google. . .

Senior

Smartass. . .

Junior

(chuckles)

Senior

We have a procedure:

1. `ssh` into the machine.
2. Execute the following commands copying its output for later filling in the report:
    - `hostname` and `date`: To include in the report.
    - `free -m`: To know if we have to buy more RAM.
    - `uptime`: To see the load of the system.
    - `df -h`: Just in case we need another hard disk drive.
3. Copy all this in an email and send it to *Susan*, the operations manager.

Junior

And why can't *Susan* `ssh` into the server herself to see all of this?

Senior

She doesn't have time for this. She is a manager, and she is very busy!

Junior

Well, I guess we can make a *Kapow!* endpoint to let her see all this information from the browser. This way she doesn't need to waste any time asking us.

I started to write it already:

```
kapow route add /capacityreport -c 'hostname | kapow set /response/body;␣
↪date | kapow set /response/body; free -m | kapow set /response/body;␣
↪uptime | kapow set /response/body; df -h | kapow set /response/body'
```

Senior

Not good enough!

First of all, that code is not readable. And the output would be something like:

```
corporate-server
Tue 26 Nov 2019 01:03:44 PM CET
              total        used        free      shared  buff/cache   ␣
↪available
Mem:          31967        2286       23473         729        6207       ␣
↪28505
Swap:             0           0           0
 13:03:44 up  5:57,  1 user,  load average: 0.76, 0.63, 0.45
Filesystem           Size  Used Avail Use% Mounted on
dev                   16G     0   16G   0% /dev
run                   16G  1.7M   16G   1% /run
```

Which is also very difficult to read!

What *Susan* is used to see is more like this:

```
Hostname:
... the output of `hostname` ...
================================================================================
Date:
... the output of `date` ...
================================================================================
Memory:
... the output of `free -m` ...
================================================================================
... and so on ...
```

Junior

All right, what about this?

```
kapow route add /capacityreport -c 'hostname | kapow set /response/body;␣
↪echo␣
↪================================================================================␣
↪| kapow set /response/body; ...'
```

Senior

That fixes the issue for *Susan*, but makes it worse for us.

What about a HEREDOC to help us make the code more readable?

Junior

A *HEREwhat*?

Senior

A HEREDOC or **here document** is the method Unix shells use to express multi-line literals.

They look like this:

```
$ cat <<-'EOF'
        you can put
        more than one line
        here
EOF
```

The shell will put the data between the first `EOF` and the second `EOF` as the `stdin` of the **cat** process.

Junior

OK, I understand. That's cool, by the way.

So, if I want to use this with *Kapow!*, I have to make it read the script from `stdin`. To do this I know that I have to put a `-` at the end.

Let me try:

```
kapow route add /capacityreport - <<-'EOF'
        hostname | kapow set /response/body
        echo␣
↪================================================================================␣
↪| kapow set /response/body
        date | kapow set /response/body
        echo␣
↪================================================================================␣
↪| kapow set /response/body
```

(continues on next page)

(continued from previous page)

```
        free -m | kapow set /response/body
        echo␣
↪================================================================================␣
↪| kapow set /response/body
        uptime | kapow set /response/body
        echo␣
↪================================================================================␣
↪| kapow set /response/body
        df -h | kapow set /response/body
        echo␣
↪================================================================================␣
↪| kapow set /response/body
EOF
```

Senior

That would work. Nevertheless I am not yet satisfied.

What about all the repeated `kapow set /response/body` statements? Do you think we could do any better?

Junior

Maybe we can redirect all output to a file and use the file as the input of `kapow set /response/body`.

Senior

There is a better way. You can make use of another neat **bash** feature: command grouping.

Command grouping allows you to execute several commands treating the group as one single command.

You can use this way:

```
{ command1; command2; } | command3
```

Junior

What about this:

```
kapow route add /capacityreport - <<-'EOF'
        {
                hostname
                echo␣
↪================================================================================
                date
                echo␣
↪================================================================================
                free -m
                echo␣
↪================================================================================
                uptime
                echo␣
↪================================================================================
                df -h
                echo␣
↪================================================================================
        } | kapow set /response/body
EOF
```

Senior

Nice! Now I am not worried about maintaining that script. Good job!

Junior

You know me. Whatever it takes to avoid writing reports ;-)

(both chuckle).

2.6 Sharing the Stats

Junior

Good morning!

Senior

Just about time. . . We are in trouble!

The report stuff was a complete success, so much so that now *Susan* has hired a frontend developer to create a custom dashboard to see the stats in real time.

Now we have to provide the backend for the solution.

Junior

And what's the problem?

Senior

We are not developers! What are we doing writing a backend?

Junior

Just chill out. Can't be that difficult. . . What do they need, exactly?

Senior

We have to provide a new endpoint to serve the same data but in `JSON` format.

Junior

So, we have half of the work done already!

What about this?

```
kapow route add /capacitystats - <<-'EOF'
        echo "{\"memory\": \"`free -m`\"}"  | kapow set /response/body
EOF
```

Senior

For starters, that's not valid `JSON`. The output would be something like:

```
$ echo "{\"memory\": \"`free -m`\"}"
{"memory": "              total        used        free      shared  buff/
↪cache    available
Mem:          31967       3121       21680        980        7166      ␣
↪27418
Swap:             0           0           0"}
```

You can't add new lines inside a `JSON` string that way, you have to escape the new line characters as `\n`.

Junior

Are you sure?

Senior

See for yourself.

```
$ echo "{\"memory\": \"`free -m`\"}" | jq
parse error: Invalid string: control characters from U+0000 through U+001F␣
↪must be escaped at line 3, column 44
```

Junior

jq? What is that command?

Senior

jq is a wonderful tool for working with `JSON` data from the command line. With **jq** you can extract data from a `JSON` document and it also allows you to generate a well-formed `JSON` document.

Junior

Let's use it, then!

How can we generate a `JSON` document with **jq**?

Senior

To generate a document we use the `-n` option:

```
$ jq -n '{"mykey": "myvalue"}'
{
  "mykey": "myvalue"
}
```

Junior

That does not seem very useful. The output is just the same.

Senior

Bear with me, it gets better. You can add variables to the `JSON` and **jq** will escape them for you.

```
$ jq -n --arg myvar "$(echo -n myvalue)" '{"mykey": $myvar}'
{
  "mykey": "myvalue"
}
```

Junior

Sweet! That's just what I need.

(hacks away for a few minutes).

What do you think of this?

```
$ jq -n --arg host "$(hostname)" --arg date "$(date)" --arg memory "$(free -
↪m)" --arg load "$(uptime)" --arg disk "$(df -h)" '{"hostname": $host, "date
↪": $date, "memory": $memory, "load": $load, "disk": $disk}'
{
  "hostname": "junior-host",
  "date": "Tue 26 Nov 2019 05:27:24 PM CET",
  "memory": "              total        used        free      shared  buff/
↪cache   available\nMem:          31967        3114       21744         913↪
↪      7109       27492\nSwap:             0           0           0",
```

(continues on next page)

(continued from previous page)

```
  "load": " 17:27:24 up 10:21,  1 user,  load average: 0.20, 0.26, 0.27",
  "disk": "Filesystem          Size  Used Avail Use% Mounted on\ndev                ␣
↪         16G      0   16G   0% /dev"
}
```

Senior

That is the output we have to produce, right. But the code is far from readable. And you also forgot about adding the endpoint.

Can we do any better?

Junior

That's easy:

```
kapow route add /capacitystats - <<-'EOF'
        jq -n \
           --arg hostname "$(hostname)" \
           --arg date "$(date)" \
           --arg memory "$(free -m)" \
           --arg load "$(uptime)" \
           --arg disk "$(df -h)" \
           '{"hostname": $hostname, "date": $date, "memory": $memory, "load
↪": $load, "disk": $disk}' \
        | kapow set /response/body
EOF
```

What do you think?

Senior

I'm afraid you forgot an important detail.

Junior

I don't think so! the `JSON` is well-formed and it contains all the required data. And the code is quite readable.

Senior

You are right, but you are not using `HTTP` correctly. You have to set the `Content-Type` header to let your client know the format of the data you are outputting.

Junior

Oh, I see. Let me try again:

```
kapow route add /capacitystats - <<-'EOF'
        jq -n \
           --arg hostname "$(hostname)" \
           --arg date "$(date)" \
           --arg memory "$(free -m)" \
           --arg load "$(uptime)" \
           --arg disk "$(df -h)" \
           '{"hostname": $hostname, "date": $date, "memory": $memory, "load
↪": $load, "disk": $disk}' \
        | kapow set /response/body
        echo application/json | kapow set /response/headers/Content-Type
EOF
```

Senior

Better. Just a couple of details.

1. You have to set the headers **before** writing to the body. This is because the body can be so big that *Kapow!* is forced to start sending it out.
2. In cases where you want to set a small piece of data (like the header), it is better not to use `stdin`. *Kapow!* provides a secondary syntax for these cases:

```
$ kapow set <resource> <value>
```

Junior

Something like this?

```
kapow route add /capacitystats - <<-'EOF'
        kapow set /response/headers/Content-Type application/json
        jq -n \
           --arg hostname "$(hostname)" \
           --arg date "$(date)" \
           --arg memory "$(free -m)" \
           --arg load "$(uptime)" \
           --arg disk "$(df -h)" \
           '{"hostname": $hostname, "date": $date, "memory": $memory, "load
↪": $load, "disk": $disk}' \
        | kapow set /response/body
EOF
```

Senior

That's perfect! Now, let's upload this to the **Corporate Server** and tell the frontend developer about it.

2.7 Securing the server

Senior

Hi. . . I hope you rested last night!

Come on, I need your help here!

Junior

Good morning! What's the matter? Sounds worrying

Senior

We forgot to take the most basic security measures when deploying our services. Every body at the company can access the services and the information is transferred in clear text.

Junior

Oh! Damn, you're right! You think we can do anything to solve this mess?

Senior

Yes, I'm pretty sure that those smart guys have thought on that when building Kapow! Have a look at the *documentation*.

Junior

Got it! They did it, here're the instictions to start a server with HTTPS support.

It's amazing! It says we can even use mTLS to control access, really promising.

Senior

Ok, ok… First thigs first. We need to get a server certificate to start working with HTTPS. Fortunately we can ask for one to the CA we use for the other servers. Let's pick up one for development, they're quick to get.

Junior

Yeah! I'll change the startup script to configure HTTPS:

```
$ kapow server --keyfile /etc/kapow/tls/keyfile    \
               --certfile /etc/kapow/tls/certfile  \
               /etc/kapow/awesome-route
```

It's easy, please copy the private key file and certificate chain to `/etc/kapow/tls` and we can restart.

Senior

Great! it's working, communications are secured. Let's say everybody to change from http to https.

Junior

Ok, did it. What are the steps to follow to limit access by using mTLS?

Senior

Besides configuring the server we need to provide the users with their own client certificates and private keys so they can configure their browsers and the application server.

Junior

Yes, please give me the CA certificate that will issue our client certificates and I'll change the startup script again

```
$ kapow server --keyfile /etc/kapow/tls/keyfile            \
               --certfile /etc/kapow/tls/certfile          \
               --clientauth true                           \
               --clientcafile /etc/kapow/tls/clientCAfile  \
               /etc/kapow/awesome-route
```

Done!

Senior

Ok, let's communicate the changes to all the affected teams before we restart

Junior

Oh God, After all we're starting to look like Google

(chuckles)

CHAPTER 3

Examples

3.1 Working with Init Scripts

3.1.1 Starting *Kapow!* using an init script

An init program, which can be just a shell script, allows you to make calls to the `kapow route` command.

```
$ kapow server example-init-program
```

With the `example-init-program`:

```
$ cat example-init-program
#!/usr/bin/env sh
#
# This is a simple example of an init program
#
echo '[*] Starting my init program'

# We add 2 Kapow! routes
kapow route add /my/route -c 'echo hello world | kapow set /response/body'
kapow route add -X POST /echo -c 'kapow get /request/body | kapow set /response/body'
```

Note: *Kapow!* can be fully configured using just init scripts

3.1.2 Writing Multiline Routes

If you need to write more complex actions, you can leverage multiline routes:

```
$ cat multiline-route
#!/usr/bin/env sh
```

(continues on next page)

(continued from previous page)

```
kapow route add /log_and_stuff - <<-'EOF'
     echo this is a quite long sentence and other stuff | tee log.txt | kapow set /response/body
     cat log.txt | kapow set /response/body
EOF
```

Warning: Be aware of the **"-"** at the end of the `kapow route add` command. It tells `kapow route add` to read commands from `stdin`.

Warning: If you want to learn more about multiline usage, see: Here Doc

3.1.3 Keeping Things Tidy

Sometimes things grow, and keeping things tidy is the only way to mantain the whole thing.

You can distribute your endpoints in several init programs. And you can keep the whole thing documented in one html file, served with *Kapow!*.

```
$ cat index-route
#!/usr/bin/env sh
kapow route add / - <<-'EOF'
    cat howto.html | kapow set /response/body
EOF

source ./info_stuff
source ./other_endpoints
```

You can import other shell script libraries with `source`.

3.1.4 Debugging Init Programs/Scripts

Since *Kapow!* redirects the standard output and the standard error of the init program given on server startup to its own, you can leverage `set -x` to see the commands that are being executed, and use that for debugging.

To support debugging user request executions, the server subcommand has a `--debug` option flag that prompts *Kapow!* to redirect both the script's standard output and standard error to *Kapow!*'s standard output, so you can leverage `set -x` the same way as with init programs.

```
$ cat withdebug-route
#!/usr/bin/env sh
kapow route add / - <<-'EOF'
    set -x
    echo "This will be seen in the log"
    echo "Hi HTTP" | kapow set /response/body
EOF

$ kapow server --debug withdebug-route
```

3.2 Managing Routes

3.2.1 Adding New Routes

Warning: Be aware that if you register more than one route with exactly the same path, only the first route added will be used.

GET route

Defining a route:

```
$ kapow route add /my/route -c 'echo hello world | kapow set /response/body'
```

Calling route:

```
$ curl http://localhost:8080/my/route
hello world
```

POST route

Defining a route:

```
$ kapow route add -X POST /echo -c 'kapow get /request/body | kapow set /response/body'
```

Calling a route:

```
$ curl -d 'hello world' -X POST http://localhost:8080/echo
hello world
```

Capturing Parts of the URL

Defining a route:

```
$ kapow route add '/echo/{message}' -c 'kapow get /request/matches/message | kapow set /response/body'
```

Calling a route:

```
$ curl http://localhost:8080/echo/hello%20world
hello world
```

3.2.2 Listing Routes

You can list the active routes in the *Kapow!* server.

```
$ kapow route list
[{"id":"20c98328-0b82-11ea-90a8-784f434dfbe2","method":"GET","url_pattern":"/echo/{message}","entrypoint":"/bin/sh -c","command":"kapow get /request/matches/message | kapow set /response/body"}]
```

Or, if you want human-readable output, you can use **jq**:

```
$ kapow route list | jq
[
  {
    "id": "20c98328-0b82-11ea-90a8-784f434dfbe2",
    "method": "GET",
    "url_pattern": "/echo/{message}",
    "entrypoint": "/bin/sh -c",
    "command": "kapow get /request/matches/message | kapow set /response/body",
  }
]
```

Note: *Kapow!* has a *HTTPS Control Interface*, bound by default to `localhost:8081`.

3.2.3 Deleting Routes

You need the ID of a route to delete it. Running the command used in the *listing routes example*, you can obtain the ID of the route, and then delete it by typing:

```
$ kapow route remove 20c98328-0b82-11ea-90a8-784f434dfbe2
```

3.3 Handling HTTP Requests

3.3.1 Add or Modify an HTTP Header

You may want to add some extra HTTP header to the response.

In this example we'll be adding the header `X-Content-Type-Options` to the response.

```
$ cat sniff-route
#!/usr/bin/env sh
kapow route add /sec-hello-world - <<-'EOF'
     kapow set /response/headers/X-Content-Type-Options nosniff
     kapow set /response/headers/Content-Type text/plain

     echo this will be interpreted as plain text | kapow set /response/body
EOF

$ kapow server nosniff-route
```

Testing with **curl**:

```
$ curl -v http://localhost:8080/sec-hello-world
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /sec-hello-word HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
```

(continues on next page)

(continued from previous page)

```
< HTTP/1.1 200 OK
< X-Content-Type-Options: nosniff
< Date: Wed, 20 Nov 2019 10:56:46 GMT
< Content-Length: 24
< Content-Type: text/plain
<
this will be interpreted as plain text
```

> **Warning:** Please be aware that if you don't explicitly specify the value of the `Content-Type` header, *Kapow!* will guess it, effectively negating the effect of the `X-Content-Type-Options` header.

Note: You can read more about the `X-Content-Type-Options: nosniff` header here.

3.3.2 Upload Files

Example #1

Uploading a file using *Kapow!* is very simple:

```
$ cat upload-route
#!/usr/bin/env sh
kapow route add -X POST /upload-file - <<-'EOF'
     kapow get /request/files/data/content | kapow set /response/body
EOF
```

```
$ cat results.json
{"hello": "world"}
$ curl       -X POST -H 'Content-Type: multipart/form-data' -F data=@results.json http://localhost:8080/upload-file
{"hello": "world"}
```

Example #2

In this example we reply the line count of the file received in the request:

```
$ cat count-file-lines
#!/usr/bin/env sh
kapow route add -X POST /count-file-lines - <<-'EOF'

     # Get sent file
     FNAME=$(kapow get /request/files/myfile/filename)

     # Counting file lines
     LCOUNT=$(kapow get /request/files/myfile/content | wc -l)

     kapow set /response/status 200

     echo "$FNAME has $LCOUNT lines" | kapow set /response/body
EOF
```

```
$ cat file.txt
hello
World
$ curl -F myfile=@file.txt http://localhost:8080/count-file-lines
file.txt has        2 lines
```

3.3.3 Sending HTTP error codes

You can specify custom status code for `HTTP` response:

```
$ cat error-route
#!/usr/bin/env sh
kapow route add /error - <<-'EOF'
     kapow set /response/status 401
     echo -n '401 error' | kapow set /response/body
EOF
```

Testing with **`curl`**:

```
$ curl -v http://localhost:8080/error
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /error HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Date: Wed, 20 Nov 2019 14:06:44 GMT
< Content-Length: 10
< Content-Type: text/plain; charset=utf-8
<
401 error
```

3.3.4 How to redirect using HTTP

In this example we'll redirect our users to `Google`:

```
$ cat redirect
#!/usr/bin/env sh
kapow route add /redirect - <<-'EOF'
     kapow set /response/headers/Location https://google.com
     kapow set /response/status 301
EOF
```

```
$ curl -v http://localhost:8080/redirect
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /redirect HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
```

(continues on next page)

(continued from previous page)

```
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently
< Location: http://google.com
< Date: Wed, 20 Nov 2019 11:39:24 GMT
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

3.3.5 Manage Cookies

If you track down some user state, *Kapow!* allows you manage Request/Response Cookies.

In the next example we'll set a cookie:

```
$ cat cookie
#!/usr/bin/env sh
kapow route add /setcookie - <<-'EOF'
     CURRENT_STATUS=$(kapow get /request/cookies/kapow-status)

     if [ -z "$CURRENT_STATUS" ]; then
             kapow set /response/cookies/Kapow-Status 'Kapow Cookie Set'
     fi

     echo -n OK | kapow set /response/body
EOF
```

Calling with **curl**:

```
$ curl -v http://localhost:8080/setcookie
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /setcookie HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Set-Cookie: Kapow-Status="Kapow Cookie Set"
< Date: Fri, 22 Nov 2019 10:44:42 GMT
< Content-Length: 3
< Content-Type: text/plain; charset=utf-8
<
OK
* Connection #0 to host localhost left intact
```

3.4 Using JSON

3.4.1 Modify JSON by Using Shell Commands

Note: Nowadays Web services are JSON-based, so making your script JSON aware is probably a good choice. In order to be able to extract data from a JSON document as well as composing JSON documents from a script, you can leverage jq.

Example #1

In this example our *Kapow!* service will receive a JSON value with an incorrect date, then our init program will fix it and return the correct value to the user.

```
$ cat fix_date
#!/usr/bin/env sh
kapow route add -X POST /fix-date - <<-'EOF'
     kapow set /response/headers/Content-Type application/json
     kapow get /request/body | jq --arg newdate "$(date +'%Y-%m-%d_%H-%M-%S')" '.
↪incorrectDate=$newdate' | kapow set /response/body
EOF
```

Call the service with **curl**:

```
$ curl -X POST http://localhost:8080/fix-date -H 'Content-Type: application/json' -d '
↪{"incorrectDate": "no way, Jose"}'
{
   "incorrectDate": "2019-11-22_10-42-06"
}
```

Example #2

In this example we extract the name field from the incoming JSON document in order to generate a two-attribute JSON response.

```
$ cat echo-attribute
#!/usr/bin/env sh
kapow route add -X POST /echo-attribute - <<-'EOF'
     JSON_WHO=$(kapow get /request/body | jq -r .name)

     kapow set /response/headers/Content-Type application/json
     kapow set /response/status 200

     jq --arg greet Hello --arg value "${JSON_WHO:-World}" --null-input '{ greet:
↪$greet, to: $value }' | kapow set /response/body
EOF
```

Call the service with **curl**:

```
$ curl -X POST http://localhost:8080/echo-attribute -H 'Content-Type: application/json
↪' -d '{"name": "MyName"}'
{
  "greet": "Hello",
  "to": "MyName"
}
```

3.5 Shell Tricks

3.5.1 How to Execute Two Processes in Parallel

We want to **ping** two machines parallel. *Kapow!* can get IP addresses from query params:

```
$ cat parallel-route
#!/usr/bin/env sh
kapow route add '/parallel/{ip1}/{ip2}' - <<-'EOF'
     ping -c 1 -- "$(kapow get /request/matches/ip1)" | kapow set /response/body &
     ping -c 1 -- "$(kapow get /request/matches/ip2)" | kapow set /response/body &
     wait
EOF
```

Calling with **curl**:

```
 $ curl -v http://localhost:8080/parallel/10.0.0.1/10.10.10.1
```

3.5.2 Script debugging

Bash provides the `set -x` builtin command that "After expanding each simple command, for command, case command, select command, or arithmetic for command, display the expanded value of PS4, followed by the command and its expanded arguments or associated word list". This feature can be used to help debugging the init programs and, together the `--debug` option in the server sub-command, the scripts executed in user requests.

3.6 Using HTTPS and mTLS with *Kapow!*

Kapow! can be accesed over HTTPS and use mutual TLS for authentication. Right now there are two possibilities to configure HTTPS/mTLS on a running server.

Note: In the following sections we refer to the host running the *Kapow!* server as `kapow:8080`.

For testing purposes you can generate a self-signed certificate with the following command:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
```

3.6.1 Using *Kapow!* built in capabilities

In this section we present the option flags that the *Kapow!* server command provides in order to set up a server with HTTPS and/or mTLS.

Enabling HTTPS

When starting the server we can provide the private key and the certificate chain presented to the clients by giving to the server the paths to the corresponding files using the `--keyfile` and `--certfile` option flags in the command line:

```
$ kapow server --keyfile path/to/keyfile --certfile path/to/certfile foobar-route
```

Now *Kapow!* is listening on its default port (8080) accepting requests over HTTPS. You can test it with the following command:

```
$ curl --cacert path/to/CAfile https://localhost:8080/endpoint
```

Where `path/to/CAfile` is the path to the file containing the CA certificate that issued *Kapow!*'s certificate.

Enabling mTLS

Once we have *Kapow!* configured to use HTTPS we can, optionally, activate mTLS so we can reject client connections that do not present a valid client certificate. Currently only issuer CA matching is supported, but in a near future we'll be able to filter by DN in order to get more fine grained authentication.

In order to activate mTLS we have to provide *Kapow!* server command with the CA certificate issuing the client certificates we want to accept with the `--clientcafile` option flag and toggle the `--clientauth=true` option flag:

```
$ kapow server --keyfile path/to/keyfile --certfile path/to/certfile --
↪clientauth=true --clientcafile path/to/clientCAfile foobar-route
```

With this configuration *Kapow!* will reject connections that do not present a client certificate or one certificate not issued by the specified CA. You can test it with the following command:

```
$ curl --cacert path/to/CAfile --cert path/to/clientcredentials https://
↪localhost:8080/endpoint
```

Where `path/to/clientcredentials` is the path to the file in PKCS#12 format containing the client certificate and private key. If you have the certificate and the private key in different files you can use the –key option to specify it independently.

3.6.2 *Kapow!* Behind a Reverse Proxy

Although *Kapow!* supports `HTTPS` you can use a reverse proxy to serve a *Kapow!* service via `HTTPS`.

In this section we present a series of reverse proxy configurations that augment the capabilities of *Kapow!*.

Caddy

- **Automatic Let's Encrypt Certificate**

  `Caddy` automatically enables `HTTPS` using `Let's Encrypt` certificates given that some criteria are met.

  ```
  yourpublicdomain.example
  proxy / kapow:8080
  ```

- **Automatic Self-signed Certificate**

  If you want `Caddy` to automatically generate a self-signed certificate for testing you can use the following configuration.

  ```
  yourdomain.example
  proxy / kapow:8080
  tls self_signed
  ```

- **Custom Certificate**

  If you already have a valid certificate for your server use this configuration.

  ```
  yourdomain.example
  proxy / kapow:8080
  tls /path/to/cert.pem /path/to/key.pem
  ```

In order to enable mutual TLS authentication read the Caddy documentation.

HAProxy

With the following configuration you can run `HAProxy` with a custom certificate.

```
frontend myserver.local
    bind *:443 ssl crt /path/to/myserver.local.pem
    mode http
    default_backend nodes

backend nodes
    mode http
    server kapow1 kapow:8080
```

Note: You can produce `myserver.local.pem` from the certificates in previous examples with this command:

```
$ cat /path/to/cert.pem /path/to/key.pem > /path/to/myserver.local.pem
```

In order to enable mutual TLS authentication read the HAProxy documentation.

nginx

With the following configuration you can run `nginx` with a custom certificate.

```
server {
 listen              443 ssl;
 server_name         myserver.local;
 ssl_certificate     /path/to/cert.pem;
 ssl_certificate_key /path/to/key.pem;

 location / {
     proxy_pass http://kapow:8080;
 }
}

In order to enable mutual TLS authentication read the `Nginx documentation`_.
```

CHAPTER 4

Concepts

This section is a contains a reference of all the important *Kapow!* concepts that you should know.

4.1 *Kapow!* HTTP Interfaces

`kapow server` sets up three HTTP server interfaces, each with a distinct and clear purpose.

4.1.1 HTTP User Interface

The `HTTP User Interface` is used to serve final user requests.

By default it binds to address `0.0.0.0` and port `8080`, but that can be changed via the `--bind` flag.

4.1.2 HTTPS Control Interface

The `HTTPS Control Interface` is used by the command `kapow route` to administer the list of system routes.

This interface uses mTLS by default (double-pinned autogenerated certs).

By default it binds to address `127.0.0.1` and port `8081`, but that can be changed via the `--control-bind` flag. If this is the case, consider also `--control-reachable-addr` which will configure the autogenerated certificate to match that address.

4.1.3 HTTP Data Interface

The `HTTP Data Interface` is used by the commands `kapow get` and `kapow set` to exchange the data for a particular request.

By default it binds to address `127.0.0.1` and port `8082`, but that can be changed via the `--data-bind` flag.

4.2 Philosophy

4.2.1 Single Static Binary

- Deployment is then as simple as it gets.
- `Docker`-friendly.

4.2.2 Shell Agnostic

- *Kapow!*, like John Snow, knows nothing, and makes no assumptions about the shell you are using. It only spawns executables.
- You are free to implement a client to the Data API directly if you are so inclined. The spec provides all the necessary details.

4.2.3 Not a Silver Bullet

You should not use *Kapow!* if your project requires complex business logic.

If you try to encode business logic in a shell script, you will **deeply** regret it soon enough.

Kapow! is designed for automating simple stuff.

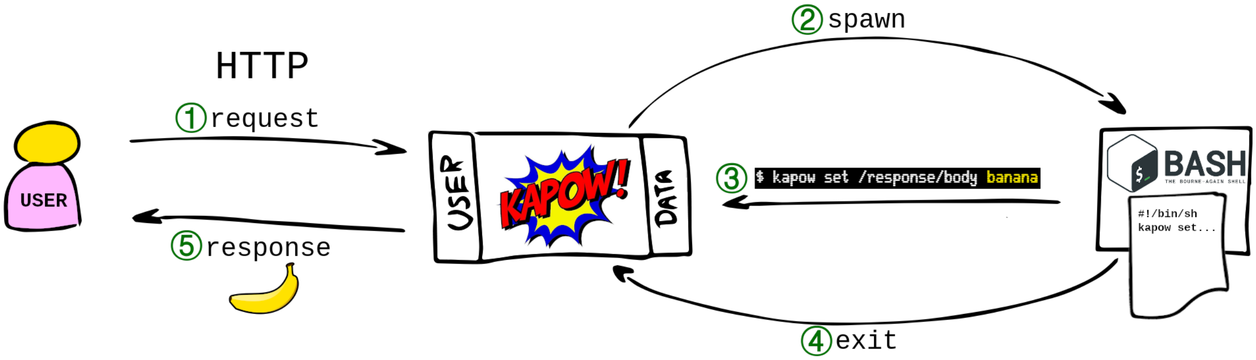
4.2.4 Interoperability over Performance

We want *Kapow!* to be as performant as possible, but not at the cost of flexibility. This is the reason why our *Data API* leverages HTTP instead of a lighter protocol for example.

When we have to choose between making things faster or more interoperable the latter usually wins.

4.3 Request Life Cycle

This section describes the sequence of events happening for each request answered by the *HTTP User Interface*.



```
$ kapow set /response/body banana
```

```
#!/bin/sh
kapow set...
```

4.3.1 1. request

The user makes a request to the *HTTP User Interface*.

- The request is matched against the route table.
- **`kapow`** provides a `HANDLER_ID` to identify this request and don't mix it with other requests that could be running concurrently.

4.3.2 2. spawn

`kapow` spawns the executable specified as entrypoint in the matching route.

The default entrypoint is **`/bin/sh`**; let's focus on this workflow.

The spawned entrypoint is run with the following variables added to its environment:

- `KAPOW_HANDLER_ID`: Containing the `HANDLER_ID`
- `KAPOW_DATA_URL`: With the URL of the *HTTP Data Interface*
- `KAPOW_CONTROL_URL`: With the URL of the *HTTPS Control Interface*

4.3.3 3. `kapow set /response/body banana`

During the lifetime of the shell, the *request and response resources* are available via these commands:

- `kapow get /request/...`
- `kapow set /response/...`

These commands use the aforementioned environment variables to read data from the user request and to write the response. They accept data either as arguments or from `stdin`.

4.3.4 4. exit

The shell dies. Long live the shell!

4.3.5 5. response

`kapow` finalizes the original request. Enjoy your banana now.

4.4 The *Kapow!* Resource Tree

This is the model that *Kapow!* uses to expose the internals of the user request being serviced.

We use this tree to get access to any data that comes in the request, as well as to compose the response.

We access the resource tree easily with the `kapow set` and `kapow get` subcommands.

4.4.1 Overview

```
/
│
├─ request
│  ├──── method                 HTTP Method used (GET, POST)
│  ├──── host                   Host part of the URL
│  ├──── version                HTTP version of the request
│  ├──── path                   Complete URL path (URL-unquoted)
│  ├──── remote                 IP address of client
│  ├──── matches
│  │     └──── <name>           Previously matched URL path parts
│  ├──── params
│  │     └──── <name>           URL parameters (after the "?" symbol)
│  ├──── headers
│  │     └──── <name>           HTTP request headers
│  ├──── cookies
│  │     └──── <name>           HTTP request cookie
│  ├──── form
│  │     └──── <name>           Value of the form field with name <name>
│  ├──── files
│  │     └──── <name>
│  │           └──── filename   Original file name of the file uploaded in the form␣
↪field <name>
│  │           └──── content    The contents of the file uploaded in the form field
↪<name>
│  └──── body                   HTTP request body
│
├─ ssl
│  └──── client
│        └──── i
│              └──── dn         Subject's DN common name coming ina request through␣
↪SSL with mTLS
├─ route
│  └──── id                     Id of the route that matched this request.
│
├─ server
│  └──── log                    Log to server stderr
│        └──── <prefix>         Optional prefix to the message
│
└─ response
  ├──── status                  HTTP status code
  ├──── headers
  │     └──── <name>            HTTP response headers
  ├──── cookies
  │     └──── <name>            HTTP request cookie
  └──── body                    Response body
```

4.4.2 Resources

`/request/method` Resource

The HTTP method of the incoming request.

Sample Usage

If the user runs:

```
$ curl -X POST http://kapow.example:8080
```

then, when handling the request:

```
$ kapow get /request/method
POST
```

`/request/host` Resource

The `Host` header as defined in the HTTP/1.1 spec of the incoming request.

Sample Usage

If the user runs:

```
$ curl http://kapow.example:8080
```

then, when handling the request:

```
$ kapow get /request/host
kapow.example
```

`/request/version` Resource

The HTTP version of the incoming request.

Sample Usage

If the user runs:

```
$ curl --http1.0 http://kapow.example:8080
```

then, when handling the request:

```
$ kapow get /request/version
HTTP/1.0
```

`/request/path` Resource

Contains the path substring of the URL.

Sample Usage

If the user runs:

```
$ curl http://kapow.example:8080/foo/bar?qux=1
```

then, when handling the request:

```
$ kapow get /request/path
/foo/bar
```

`/request/remote` Resource

The IP address of the host making the incoming request.

Sample Usage

If the user runs:

```
$ curl http://kapow.example:8080
```

then, when handling the request:

```
$ kapow get /request/remote
192.168.100.156
```

`/request/matches/<name>` Resource

Contains the part of the URL captured by the pattern `name`.

Sample Usage

For a route defined like this:

```
$ kapow route add /foo/{mymatch}/bar
```

if the user runs:

```
$ curl http://kapow.example:8080/foo/1234/bar
```

then, when handling the request:

```
$ kapow get /request/matches/mymatch
1234
```

`/request/params/<name>` Resource

Contains the value of the URL parameter `name`

Note: In the reference implementation only the first parameter's value can be accessed in the case of multiple values coming in the request.

Sample Usage

If the user runs:

```
$ curl http://kapow.example:8080/foo?myparam=bar
```

then, when handling the request:

```
$ kapow get /request/params/myparam
bar
```

`/request/headers/<name>` Resource

Contains the value of the HTTP header `name` of the incoming request.

Note: In the reference implementation only the first header's value can be accessed in the case of multiple values coming in the request.

Sample Usage

If the user runs:

```
$ curl -H X-My-Header=Bar http://kapow.example:8080/
```

then, when handling the request:

```
$ kapow get /request/headers/X-My-Header
Bar
```

`/request/cookies/<name>` Resource

Contains the value of the HTTP cookie `name` of the incoming request.

Sample Usage

If the user runs:

```
$ curl --cookie MYCOOKIE=Bar http://kapow.example:8080/
```

then, when handling the request:

```
$ kapow get /request/cookies/MYCOOKIE
Bar
```

`/request/form/<name>` Resource

Contains the value of the field `name` of the incoming request.

Note: In the reference implementation there are some caveats:

- Only the first form field's value can be accessed in the case of multiple values coming in the request.
- In order to get access to the form data a correct 'Content-Type' header must be present in the request ('application/x-www-form-urlencoded' or 'multipart/form-data')

Sample Usage

If the user runs:

```
$ curl -F -d myfield=foo http://kapow.example:8080/
```

then, when handling the request:

```
$ kapow get /request/form/myfield
foo
```

`/request/files/<name>/filename` Resource

Contains the name of the file uploaded through the incoming request.

Note: In the reference implementation to get access to the multipart data a correct Content-Type header must be present in the request (multipart/form-data or multipart/mixed).

Sample Usage

If the user runs:

```
$ curl -F 'myfile=@filename.txt' http://kapow.example:8080/
```

then, when handling the request:

```
$ kapow get /request/files/myfile/filename
filename.txt
```

`/request/files/<name>/content` Resource

Contents of the file that is being uploaded in the incoming request.

Note: In the reference implementation to get access to the multipart data a correct Content-Type header must be present in the request (multipart/form-data or multipart/mixed).

Sample Usage

If the user runs:

```
$ curl -F 'myfile=@filename.txt' http://kapow.example:8080/
```

then, when handling the request:

```
$ kapow get /request/files/myfile/content
...filename.txt contents...
```

`/request/body` Resource

Raw contents of the incoming request HTTP body.

Sample Usage

If the user runs:

```
$ curl --data-raw foobar http://kapow.example:8080/
```

then, when handling the request:

```
$ kapow get /request/body
foobar
```

`/ssl/client/i/dn` Resource

The IP address of the host making the incoming request.

Sample Usage

If the user runs:

```
$ curl --cacert path/to/CAfile --cert path/to/clientcredentials http://kapow.
↪example:8080
```

using a client certificate with DN=subject@example.net then, when handling the request:

```
$ kapow get /ssl/client/i/dn
subject@example.net
```

`/route/id` Resource

The ID of the original route that matched this request..

Sample Usage

If the user runs:

```
$ curl http://kapow.example:8080/
```

then, when handling the request:

```
$ kapow get /route/id
ecd5d63f-f28b-11ea-ac55-ec21e5089c1f
```

`/server/log/<prefix>`

Allows logging to Kapow!'s server stderr:

```
$ kapow set /server/log/my_banana_peeler 'banana too slippery!'
```

… meanwhile, in the server…

```
2020/12/18 11:15:49.437642 620846c6-4122-11eb-abeb-002b671b12f9 my_banana_peeler:␣
↪banana too slippery!
```

Note that the '/my_banana_peeler' part is an optional arbitrary prefix, and can be omitted.

`/response/status` Resource

Contains the status code given in the user response.

Note: In the reference implementation there are some caveats:

- The status code value must be between 100 and 999.
- There is no way of writing reason phrase in the status line of the response.

Sample Usage

If during the request handling:

```
$ kapow set /response/status 418
```

then the response will have the status code `418 I am a Teapot`.

`/response/headers/<name>` Resource

Contains the value of the header `name` in the user response.

Note: At this moment header values are only appended, there is no way of reset the values once set.

Sample Usage

If during the request handling:

```
$ kapow set /response/headers/X-My-Header Foo
```

then the response will contain an HTTP header named `X-My-Header` with value `Foo`.

`/response/cookies/<name>` Resource

Contains the value of the cookie `name` that will be set to the user response.

Sample Usage

If during the request handling:

```
$ kapow set /response/cookies/MYCOOKIE Foo
```

then the response will set the cookie `MYCOOKIE` to the user in following requests.

`/response/body` Resource

Contains the value of the response HTTP body.

Sample Usage

If during the request handling:

```
$ kapow set /response/body foobar
```

then the response will contain `foobar` in the body.

4.5 Route Matching

Kapow! maintains a *route* table with a list of routes as provided by the user, and uses it to determine which handler an incoming request should be dispatched to.

Each incoming request is matched against the routes in the route table in strict order. For each route in the route table, the criteria are checked. If the request does not match, the next route in the route list is examined.

4.6 Routes

A *Kapow!* route specifies the matching criteria for an incoming request on the *HTTP User Interface*, and the details to handle it.

Kapow! implements a *route table* where all routes reside.

A route can be set like this:

```
$ kapow route add \
   -X POST \
   '/register/{username}' \
   -e '/bin/bash -c' \
   -c 'touch /var/lib/mydb/"$(kapow get /request/matches/username)"' \
   | jq
{
   "id": "deadbeef-0d09-11ea-b18e-106530610c4d",
   "method": "POST",
   "url_pattern": "/register/{username}",
   "entrypoint": "/bin/bash -c",
   "command": "touch /var/lib/mydb/\"$(kapow get /request/matches/username)\""
}
```

Let's use this example to discuss its elements.

4.6.1 Elements

`id` Route Element

Uniquely identifies each route. It is used for instance by `kapow route remove <route_id>`.

Note: The current implementation of *Kapow!* autogenerates a `UUID` for this field. In the future the user will be able to specify a custom value.

`method` Route Element

Specifies the HTTP method for the route to match the incoming request.

Note that the route shown above will only match a `POST` request.

`url_pattern` Route Element

It matches the `path` component of the `URL` of the incoming request.

It can contain regex placeholders for easily capturing fragments of the path.

In the route shown above, a request with a URL `/register/joe` would match, assigning `joe` to the placeholder `username`.

Kapow! leverages Gorilla Mux for managing routes. For the full story, see https://github.com/gorilla/mux

`entrypoint` Route Element

This sets the executable to be spawned, along with any arguments required.

In the route shown above, the entrypoint that will be run is `/bin/bash -c`, which is an incomplete recipe. It is then completed by the *command element*.

Note: The semantics of this element closely match the `Dockerfile`'s ENTRYPOINT directive.

`command` Route Element

This is an optional last argument to be passed to the *entrypoint*.

In the route shown above, it completes the `entrypoint` to form the final incantation to be executed:

```
/bin/bash -c 'touch /var/lib/mydb/"$(kapow get /request/matches/username)"'
```

Note: The semantics of this element closely match the `Dockerfile`'s CMD directive.

4.6.2 Matching Algorithm

Kapow! leverages Gorilla Mux for this task. Check their documentation for the gory details.

Index