Everybody is freaking out about the rise of the Bitcoin and the potential of the **Blockchain** technologies. The advent of cryptocurrencies, game changing use cases, disruption of established business models by disintermediation, etc.By the time I’m writing this article, there are more than 1300 crypto-currencies listed in [coinmarketcap](https://coinmarketcap.com/all/views/all/).. And a lot more coming with the next ICOs (Internet Coin Offering).

Most certainly, the main enabler of Bitcoin and of many other currencies (although not all of them) is the Blockchain technology.

Although the original paper from Satoshi explains very well the ground idea behind this technology, nothing like **creating your own blockchain** to fully understand how it works, its limitations and potential improvements (aka ***“If you can code it, you certainly understand it”***).

In this post I’d like to share a gentle coding exercise in R (#rstats). Why R? Just because it’s my favorite language… I wouldn’t choose R for a productive, full-fledge block chain implementation, but again, this is not the purpose of this post. This is just a learning exercise hacked quite quickly without any aspiration of ever running this code in a productive environment, and should be understood as such.

**First things first: what is what in a Blockchain**

 A **blockchain** is an immutable chain of sequential records or blocks that are “chained” together using hashes. **Blocks** can be understood as containers, typically of transactions, but it can be extended to documents, etc. We can think of a blockchain as a database where new data is stored in blocks, that are added to an *immutable* chain based on all other existing blocks.  
Blockchain is often referred as a *digital ledger* of transactions performed in a cryptocurrency of choice (Bitcoin or whatever). A **transaction** requires a sender address, a recipient address and a given amount and needs to be assigned to a block.  
The json below shows a typical block with a particular index, an integer representing the proof, a hashed representation of the previous block, which allows for consistency check (more about proof and hashing later) and a timestamp.

{

"index": 2,

"timestamp": 1514108190.2831,

"transactions": [{

"sender": "d4ee26eee15148ee92c6cd394edd964",

"recipient": "23448ee92cd4ee26eee6cd394edd964",

"amount": 15

}, {

"sender": "6eee15148ee92c6cd394edd974d4ee2",

"recipient": "15148ee92cd4ee26eee6cd394edd964",

"amount": 225

}],

"proof": 211,

"previousHash": "afb49032c6c086445a1d420dbaf88e4925681dec0a5b660d528fe399e557bf68"

}

Once we have understood the concept behind the blockchain, let’s build one and make it work   
We are going to need three different files:

* The **class definition file**, where we create the Blockchain class with its components and methods.
* The **API definition file**, where we instantiate the class, register a node and expose the blockchain methods as GET and POST calls for the peers to interact with.
* The **plumber launch file**, to start the server and expose the API methods

**Building a Blockchain**

To represent the Blockchain, we need a list of blocks (the **chain**), a list of **currentTransactions** (waiting to be mined) and a list of mining **nodes**.

Our Blockchain class implements a method to register a new transaction **addTransaction**. Transactions are appended to the list of currentTransactions until a new block is created, which takes care of the newly added transactions that have not been added to any block yet.  
The creation of a new block takes place after calling **nextBlock**. To maintain the consistency, a new block can only be added knowing the hashed value of the previous one as part of a Proof of Work procedure (PoW). Basically, it is just finding a number that satisfies a condition. This number should be easy enough to be verified by anyone in the chain, but difficult enough so that a brute force attack to find it wouldn’t be feasible (too long, too expensive).  
In our example, **proofOfWork** relies on finding a number called *proof* such that if we append this proof to the previous proof and we hash it, the last 2 characters of the resulting string are exactly two zeroes. The way it works in the real Bitcoin chain is quite different, but based on the same concept.

In Bitcoin, the **Proof of Work** algorithm is called Hashcash. Solving this problem requires computational resources (the longer the strings and the more characters to be found within it, the higher the complexity), and miners are rewarded by receiving a coin—in a transaction.

The Blockchain class provides a method to check the consistency over all blocks **validChain**, which iterates over the chain checking if both each block is *properly linked* to the previous one and whether the PoW is preserved.  
The method **registerNode** adds the URL of a mining node to the central nodes register.

Finally, as we need to think of a Blockchain as a distributed system, there is a chance that one node has a different chain to another node. The handleConflicts resolves this conflict by declaring the longest chain the proper one… the one that all nodes take.

list.of.packages <- c("digest", "httr","jsonlite")

new.packages <- list.of.packages[!(list.of.packages %in% installed.packages()[,"Package"])]

if(length(new.packages)) install.packages(new.packages)

require(digest)

require(jsonlite)

require(httr)

Blockchain <- function ()

{

bc = list (

chain = list(),

currentTransactions = list(),

nodes = list()

)

#' Create a new Block in the Blockchain

#'

#' @param proof The proof given by the Proof of Work algorithm

#' @param previousHash Hash of previous Block

#' @return new block generated given the \code{proof} and the \code{previousHash}

#' @examples

#' blockchain = Blockchain()

#' blockchain$nextBlock(previousHash=1, proof=100) # genesis block

bc$nextBlock = function (proof, previousHash=NULL){

previousHash <- ifelse (is.null(previousHash), bc$hashBlock(bc$chain[length(bc$chain)]), previousHash)

block = list('block' = list('index' = length (bc$chain) + 1, 'timestamp' = as.numeric(Sys.time()) , 'transactions' = bc$currentTransactions, 'proof' = proof, 'previousHash' = previousHash))

bc$currentTransactions = NULL

bc$chain <- append(bc$chain, block)

return (block)

}

#' Returns the last block in the Blockchain

#'

#' @examples

#' blockchain$lastBlock()

bc$lastBlock = function () {

bc$chain[length(bc$chain)]

}

#' Register a new transaction in the Blockchain

#'

#' @param sender address of the sender

#' @param recipient address of the recipient

#' @param amount transaction amount

#' @return Index of the Block that will hold this transaction

bc$addTransaction = function (sender, recipient, amount)

{

txn <- list('transaction'= list('sender'=sender,'recipient'=recipient,'amount'=amount))

bc$currentTransactions <- append(bc$currentTransactions, txn)

last.block <- bc$lastBlock()

return(last.block$block$index + 1)

}

#' Hash a block using SHA256

#'

#' @param block

#' @return SHA256 hashed value for \code(block)

#' @examples

bc$hashBlock = function (block) {

require(digest)

digest(block,algo="sha256")

}

#' Find a number p' such that hash(pp') contains leading 4 zeroes, where p is the previous p'

#' p is the previous proof and p' is the new proof

#' @param last\_proof

#' @return SHA256 hashed value for \code(block)

bc$proofOfWork <- function (last\_proof)

{

proof <- 0

while (!bc$validProof(last\_proof, proof))

{

proof <- proof + 1

}

return (proof)

}

#' Find a number p' such that hash(pp') ends with two zeroes, where p is the previous p'

#' p is the previous proof and p' is the new proof

#' @param last\_proof previous proof

#' @param proof proof

#' @return TRUE if correct, FALSE if not

bc$validProof <- function (last\_proof, proof)

{

guess = paste0(last\_proof,proof)

guess\_hash = digest(guess, algo = 'sha256')

return (gsub('.\*(.{2}$)', '\\1',guess\_hash) == "00")

}

#' Checks whether a given blockchain is valid

#'

#' @return TRUE if the chain is valid, FALSE otherwise

bc$validChain <- function (chain)

{

lastBlock <- chain[0]

currentIndex <- 1

while (currentIndex < length(chain))

{

block = chain[currentIndex]

# checking for valid linking

if (block$block$previousHash != bc$hashBlock(lastBlock)) {

return(FALSE)

}

# checking for proof validity

if(!bc$validProof(lastBlock$block$proof, block$block$proof))

{

return (FALSE)

}

lastBlock <- block

currentIndex <- currentIndex +1

}

return(TRUE)

}

#' Add a new node to the list of existing nodes

#'

#' @param address full URL of the node

#' @examples

#' blockchain = Blockchain()

#' blockchain$registerNode('http://192.168.0.5:5000')

bc$registerNode <- function(address)

{

parsed\_url = address

bc$nodes<- append(bc$nodes, parsed\_url)

}

#' Resolve conflicts by replacing the current chain by the longest chain in the network

#'

#' @return TRUE if the chain was replaced, FALSE otherwise

bc$handleConflicts <- function()

{

neighbours <- bc$nodes

new\_chain <- NULL

max\_length = length(bc$chain)

for (i in 1:length(neighbours))

{

chain.node <- GET(paste0(neighbours[i],'/chain'))

node.chain.length <- jsonlite::fromJSON(chain.node)$length

node.chain.chain <- jsonlite::fromJSON(chain.node)$chain if (node.chain.length > max\_length)

{

new\_chain = node.chain.chain

max\_length<-node.chain.length

}

}

if (!is.null(new\_chain))

{

bc$chain <- new\_chain

}

}

# Adding bc to the environment

bc <- list2env(bc)

class(bc) <- "BlockchainClass"

return(bc)

}

**Defining the API Methods**

After defining the Blockchain class, we need to create an instance of it running on a node. First we generate a valid identifier for the node (using *Universally Unique IDentifier*). Then, we add the *genesis block* or first block in the chain, using some default parameters (previousHash=1 and proof=100).  
Everything else takes place when users invoke the **“transaction/new”** method to create new transactions and miners call the **“/mine”** function to trigger the creation of new blocks according to the PoW schema and process the newly created transactions.

Apart from these core methods, we also enable the registration of new nodes **“/nodes/register”**, the consensus method to handle potential conflicts **“/nodes/resolve”**, the retrieval **“/chain”** and *html* display of the chain **“/chain/show”**.

To host these methods, we use rplumber, which is a wonderful package to expose R functions as *REST* and *RESTFULL* services. It’s really versatile and easy to use .Apart from the aforementioned methods, we define a filter to log all requests with the server (*logger*).

Advanced Steps for Hosting

Once you have developed your Plumber API, the next step is to find a way to host it. If you haven’t dealt with hosting an application on a server before, you may be tempted to run the run() command from an interactive session on your development machine (either your personal desktop or an RStudio Server instance) and direct traffic there. This is a dangerous idea for a number of reasons:

1. Your development machine likely has a dynamic IP address. This means that clients may be able to reach you at that address today, but it will likely break on you in the coming weeks/months.
2. Networks may leverage firewalls to block incoming traffic to certain networks and machines. Again, it may appear that everything is working for you locally, but other users elsewhere in the network or external clients may not be able to connect to your development machine.
3. If your Plumber process crashes (for instance, due to your server running out of memory), the method of running Plumber will not automatically restart the crashed service for you. This means that your API will be offline until you manually login and restart it. Likewise if your development machine gets rebooted, your API will not automatically be started when the machine comes back online.
4. This technique relies on having your clients specify a port number manually. Non-technical users may be tripped up by this; some of the other techniques do not require clients specifying the port for an API.
5. This approach will eternally run one R process for your API. Some of the other approaches will allow you to load-balance traffic between multiple R processes to handle more requests. RStudio Connect will even dynamically scale the number of running processes for you so that your API isn’t consuming more system resources than is necessary.
6. Most importantly, serving public requests from your development environment can be a security hazard. Ideally, you should separate your development instances from the servers that are accessible by others.

For these reasons and more, you should consider setting up a separate server on which you can host your Plumber APIs. There are a variety of options that you can consider.

RStudio Connect

RStudio Connect is an enterprise publishing platform from RStudio. It supports push-button publishing from the RStudio IDE of a variety of R content types including Plumber APIs. Unlike all the other options listed here, RStudio Connect automatically manages the dependent packages and files your API has and recreates an environment closely mimicking your local development environment on the server.

RStudio Connect automatically manages the number of R processes necessary to handle the current load and balances incoming traffic across all available processes. It can also shut down idle processes when they’re not in use. This allows you to run the appropriate number of R processes to scale your capacity to accommodate the current load.

DigitalOcean

DigitalOcean is an easy-to-use Cloud Computing provider. They offer a simple way to spin up a Linux virtual machine and access it remotely. You can choose what size machine you want to run – with options ranging from small machines with 512MB of RAM for a few dollars a month up to large machines with dozens of GB of RAM – and only pay for it while it’s online.

To deploy your Plumber API to DigitalOcean, please check out the plumber companion package [plumberDeploy](https://github.com/meztez/plumberDeploy).

Docker (Basic)

Docker is a platform built on top of Linux Containers that allow you to run processes in an isolated environment; that environment might have certain resources/software pre-configured or

You can get this image with a

docker pull rstudio/plumber

Remember that this will get you the current snapshot of Plumber and will continue to use that image until you run pull again.

Default Dockerfile

We’ll start by just running a single Plumber application in Docker just to see things at work. By default, the rstudio/plumber image will take the first argument after the image name as the name of the file that you want to [plumb()](https://www.rplumber.io/reference/plumb.html) and serve on port 8000. So right away you can run one of the examples that’s included in plumber as it is already installed on the image.

docker run --rm -p 8000:8000 rstudio/plumber

which is the same as:

docker run --rm -p 8000:8000 rstudio/plumber \

/usr/local/lib/R/site-library/plumber/plumber/04-mean-sum/plumber.R

* docker run tells Docker to run a new container
* --rm tells Docker to clean-up after the container when it’s done
* -p 8000:8000 says to map port 8000 from the plumber container (which is where we’ll run the server) to port 8000 of your local machine
* rstudio/plumber is the name of the image we want to run
* /usr/local/lib/R/site-library/plumber/plumber/03-mean-sum/plumber.R is the path **inside of the Docker container** to the Plumber file you want to host. You’ll note that you do not need plumber installed on your host machine for this to work, nor does the path /usr/local/... need to exist on your host machine. This references the path inside of the docker container where the R file you want to plumb() can be found. This mean-sum path is the default path that the image uses if you don’t specify one yourself.

This will ask Plumber to plumb and run the file you specified on port 8000 of that new container. Because you used the -p argument, port 8000 of your local machine will be forwarded into your container. You can test this by running this on the machine where Docker is running: curl localhost:8000/mean, or if you know the IP address of the machine where Docker is running, you could visit it in a web browser. The /mean path is one that’s defined in the plumber file we just specified – you should get an single number in an array back ([-0.1993]).

If that works, you can try using one of your own plumber files in this arrangement. Keep in mind that the file you want to run **must** be available inside of the container and you must specify the path to that file as it exists inside of the container. Keep it simple for now – use a plumber file that doesn’t require any additional R packages or depend on any other files outside of the plumber definition.

For instance if you have a plumber file saved in your current directory called api.R, you could use the following command

docker run --rm -p 8000:8000 -v `pwd`/api.R:/plumber.R rstudio/plumber /plumber.R

You’ll notice that we used the -v argument to specify a “volume” that should be mapped from our host machine into the Docker container. We defined that the location of that file should be at /plumber.R, so that’s the argument we give last to tell the container where to look for the plumber definition. You can use this same technique to share a whole directory instead of just passing in a single R file; this approach is useful if your Plumber API depends on other files.

You can also use the rstudio/plumber image just like you use any other. For example, if you want to start a container based on this image and poke around in a bash shell:

docker run -it --rm --entrypoint /bin/bash rstudio/plumber

This can be a handy way to debug problems. Prepare the command that you think should work then add --entrypoint /bin/bash before rstudio/plumber and explore a bit. Alternatively, you can try to run the R process and spawn the plumber application yourself and see where things go wrong (often a missing package or missing file).

Custom Dockerfiles

You can build upon the rstudio/plumber image and build your own Docker image by writing your own Dockerfile. Dockerfiles have a vast array of options and possible configuration.

A couple of commands that are relevant here:

* RUN runs a command and persists the side-effects in the Docker image you’re building. So if you want to build a new image that has the broom package, you could add a line in your Dockerfile that says RUN R -e "install.packages('broom')" which would make the broom package available in your new Docker image.
* ENTRYPOINT is the command to run when starting the image. rstudio/plumber specifies an entrypoint that starts R, plumb()s a file, then run()s the router. If you want to change how plumber starts, or run some extra commands (like add a global processor) before you run the router, you’ll need to provide a custom ENTRYPOINT.
* CMD these are the default arguments to provide to ENTRYPOINT. rstudio/plumber uses only the first argument as the name of the file that you want to plumb().

So your custom Dockerfile could be as simple as:

FROM rstudio/plumber

MAINTAINER Docker User <docker@user.org>

RUN R -e "install.packages('broom')"

CMD ["/app/plumber.R"]

This Dockerfile would just extend the rstudio/plumber image in two ways. First, it RUNs one additional command to install the broom package. Second, it customizes the default CMD argument that will be used when running the image. In this case, you would be expected to mount a Plumber application into the container at /app/plumber.R

You could then build your custom Docker image from this Dockerfile using the command docker build -t mycustomdocker . (where . – the current directory – is the directory where that Dockerfile is stored).

Then you’d be able to use docker run --rm -vpwd:/app mycustomdocker to run your custom image, passing in your application’s directory as a volume mounted at /app.

Automatically Run on Restart

If you want your container to start automatically when your machine is booted, you can use the --restart parameter for docker run.

docker run -p 1234:8000 -dit --restart=unless-stopped myCustomDocker would run the custom image you created above automatically every time your machine boots and expose the plumber service on port 1234 of your host machine, unless the container is explicitly stopped. Like all other hosting options, you’ll need to make sure that your firewall allows connections on port 1234 if you want others to be able to access your service.

Docker (Advanced)

In order to coordinate and run multiple Plumber processes on one server, **you should install docker-compose on your system.** This is not included with some installations of Docker, Docker Compose helps orchestrate multiple Docker containers. If you’re planning to run more than one Plumber process, you’ll want to use Docker Compose to keep them all alive and route traffic between them.

Multiple Plumber Applications

We’ll use Docker Compose to help us organize multiple Plumber processes. We won’t go into detail about how to use Docker Compose, so if you’re new you should familiarize yourself using the official docs.

You should define a Docker Compose configuration that defines the behavior of every Plumber application that you want to run. You’ll first want to setup a Dockerfile that defines the desired behavior for each of your applications. You could use a docker-compose.yml configuration like the following:

version: '2'

services:

app1:

build: ./app1/

volumes:

- ./data:/data

- ./app1:/app

restart: always

ports:

- "7000:8000"

app2:

image: rstudio/plumber

command: /app/plumber.R

volumes:

- ../app2:/app

restart: always

ports:

- "7001:8000"

More detail on what each of these options does and what other options exist can be found [here](https://docs.docker.com/compose/compose-file/). This configuration defines two Docker containers that should run app1 and app2. The associated files in this case are laid out on disk as follows:

docker-compose.yml

app1

├── Dockerfile

├── api.R

app2

├── plumber.R

data

├── data.csv

You can see that app2 is the simpler of the two apps; it just has the plumber definition that should be run through [plumb()](https://www.rplumber.io/reference/plumb.html). So we merely use the default plumber Docker image as its image, and then customize the command to specify where the Plumber API definition can be found in the container. Since we’re mapping our host’s ./app2 to /app inside of the container, the definition would be found in /app/plumber.R. We specify that it should always restart if anything ever happens to the container, and we export port 8000 from the container to port 7001 on the host.

app1 is our more complicated app. It has some extra data in another directory that needs to be loaded, and it has a custom Dockerfile. This could be because it has additional R packages or system dependencies that it requires.

If you now run docker-compose up, Docker Compose will build the referenced images in your config file and then run them. You’ll find that app1 is available on port 7000 of the machine running Docker Compose, and app2 is available on port 7001. If you want these APIs to run in the background and survive restarts of your server, you can use the -d switch just like with docker run.

Multiple Applications on One Port

It may desirable to run all of your Plumber services on a standard port like 80 (for HTTP) or 443 (for HTTPS). In that case, you’d prefer to have a router running on port 80 that can send traffic to the appropriate Plumber API by distinguishing based on a path prefix. Requests for myserver.com/app1/ could be sent to the app1 container, and myserver.org/app2/ could target the app2 container, but both paths would be available on port 80 on your server.

In order to do this, we can use another Docker container running [nginx](https://www.nginx.com/) which is configured to route traffic between the two Plumber containers. We’d add the following entry to our docker-compose.yml below the app containers we already have defined.

nginx:

image: nginx:1.9

ports:

- "80:80"

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf:ro

restart: always

depends\_on:

- app1

- app2

This uses the nginx Docker image that will be downloaded for you. In order to run nginx in a meaningful way, we have to provide a configuration file and place it in /etc/nginx/nginx.conf, which we do by mounting a local file at that location on the container.

A basic nginx config file could look something like the following:

events {

worker\_connections 4096; ## Default: 1024

}

http {

default\_type application/octet-stream;

sendfile on;

tcp\_nopush on;

server\_names\_hash\_bucket\_size 128; # this seems to be required for some vhosts

server {

listen 80 default\_server;

listen [::]:80 default\_server ipv6only=on;

root /usr/share/nginx/html;

index index.html index.htm;

server\_name MYSERVER.ORG;

location /app1/ {

proxy\_pass http://app1:8000/;

proxy\_set\_header Host $host;

}

location /app2/ {

proxy\_pass http://app2:8000/;

proxy\_set\_header Host $host;

}

location ~ /\.ht {

deny all;

}

}

}

You should set the server\_name parameter above to be whatever the public address is of your server. You can save this file as nginx.conf in the same directory as your Compose config file.

Docker Compose is intelligent enough to know to route traffic for http://app1:8000/ to the app1 container, port 8000, so we can leverage that in our config file. Docker containers are able to contact each other on their non-public ports, so we can go directly to port 8000 for both containers. This proxy configuration will trim the prefix off of the request before it sends it on to the applications, so your applications don’t need to know anything about being hosted publicly at a URL that includes the /app1/ or /app2/ prefixes.

We should also get rid of the previous port mappings to ports 7000 and 7001 on our other applications, as we don’t want to expose our APIs on those ports anymore.

If you now run docker compose up again, you’ll see your two application servers running but now have a new nginx server running, as well. And you’ll find that if you visit your server on port 80, you’ll see the “welcome to Nginx!” page. If you access /app1 you’ll be sent to app1 just like we had hoped.

Load Balancing

If you’re expecting a lot of traffic on one application or have an API that’s particularly computationally complex, you may want to distribute the load across multiple R processes running the same Plumber application. Thankfully, we can use Docker Compose for this, as well.

First, we’ll want to create multiple instances of the same application. This is easily accomplished with the docker-compose scale command. You simply run docker-compose scale app1=3 to run three instances of app1. Now we just need to load balance traffic across these three instances.

You could setup the nginx configuration that we already have to balance traffic across this pool of workers, but you would need to manually re-configure and update your nginx instance every time that you need to scale the number up or down, which might be a hassle. Luckily, there’s a more elegant solution.

We can use the dockercloud/haproxy Docker image to automatically balance HTTP traffic across a pool of workers. This image is intelligent enough to listen for workers in your pool arriving or leaving and will automatically remove/add these containers into their pool. Let’s add a new container into our configuration that defines this load balancer

lb:

image: 'dockercloud/haproxy:1.2.1'

links:

- app1

volumes:

- /var/run/docker.sock:/var/run/docker.sock

The trick that allows this image to listen in to our scaling of app1 is by passing in the docker socket as a shared volume. Note that this particular arrangement will differ based on your host OS. The above configuration is intended for Linux, but MacOS X users would require a slightly different config.

We could export port 80 of our new load balancer to port 80 of our host machine if we solely wanted to load-balance a single application. Alternatively, we can actually use both nginx (to handle the routing of various applications) and HAProxy (to handle the load balancing of a particular application). To do that, we’d merely add a new location block to our nginx.conf file that knows how to send traffic to HAProxy, or modify the existing location block to send traffic to the load balancer instead of going directly to the application.

So the location /app1/ block becomes:

location /app1/ {

proxy\_pass http://lb:8000/;

proxy\_set\_header Host $host;

}

Where lb is the name of the HAProxy load balancer that we defined in our Compose configuration.

The next time you start/redeploy your Docker Compose cluster, you’ll be balancing your incoming requests to /app1/ across a pool of 1 or more R processes based on whatever you’ve set the scale to be for that application.

Do keep in mind that when using load-balancing that it’s not longer guaranteed that subsequent requests for a particular application will land on the same process. This means that if you maintain any state in your Plumber application (like a global counter, or a user’s session state), you can’t expect that to be shared across the processes that the user might encounter. There are at least three possible solutions to this problem:

1. Use a more robust means of maintaining state. You could put the state in a database, for instance, that lives outside of your R processes and your Plumber processes could get and save their state externally.
2. You could serialize the state to the user using (encrypted) session cookies, assuming it’s small enough. In this scenario, your workers would write data back to the user in the form of a cookie, then the user would include that same cookie in its subsequent requests. This works best if the state is going to be set rarely and read often (for instance, the cookie could be set when the user logs in, then read on each request to detect the identity of this user).
3. You can enable “sticky sessions” in the HAProxy load balancer. This would ensure that each user’s traffic always gets routed to the same worker. The downside of this approach is that it will distribute traffic less evenly. You could end up in a situation in which you have 2 R processes for an application but 90% of your traffic is hitting one of them if it happens the users triggering the majority of the requests are all “stuck” to one particular worker.

pm2

pm2 is a process manager initially targeting Node.js. Here we’ll show the commands needed to do this in Ubuntu 14.04, but you can use any Operating System or distribution that is supported by pm2. At the end, you’ll have a server that automatically starts your plumber services when booted, restarts them if they ever crash, and even centralizes the logs for your plumber services.

Server Deployment and Preparation

The first thing you’ll need to do, regardless of which process manager you choose, is to deploy the R files containing your plumber applications to the server where they’ll be hosted. Keep in mind that you’ll also need to include any supplemental R files that are source()d in your plumber file, and any other datasets or dependencies that your files have.

You’ll also need to make sure that the R packages you need (and the appropriate versions) are available on the remote server. You can either do this manually by installing those packages or you can consider using a tool like Packrat to help with this.

There are a myriad of features in pm2 that we won’t cover here. It is a good idea to spend some time reading through their documentation to see which features might be of interest to you and to ensure that you understand all the implications of how pm2 hosts services (which user you want to run your processes as, etc.).

Install pm2

Now you’re ready to install pm2. pm2 is a package that’s maintained in npm (Node.js’s package management system); it also requires Node.js in order to run. So to start you’ll want to install Node.js. On Ubuntu 14.04, the necessary commands are:

sudo apt-get update

sudo apt-get install nodejs npm

Once you have npm and Node.js installed, you’re ready to install pm2.

sudo npm install -g pm2

If you find errors like SSL Error: CERT\_UNTRUSTED while using npm command, you can bypass the ssl error using:

npm config set strict-ssl false

or set the registry URL from https:// to http://:

npm config set registry="http://registry.npmjs.org/"

This will install pm2 globally (-g) on your server, meaning you should now be able to run pm2 --version and get the version number of pm2 that you’ve installed.

In order to get pm2 to startup your services on boot, you should run sudo pm2 startup which will create the necessary files for your system to run pm2 when you boot your machine.

Wrap Your Plumber File

Once you’ve deployed your Plumber files onto the server, you’ll still need to tell the server *how* to run your server. You’re probably used to running commands like

[pr](https://www.rplumber.io/reference/pr.html)("myfile.R") [%>%](https://www.rplumber.io/reference/pipe.html) [pr\_run](https://www.rplumber.io/reference/pr_run.html)(port=4500)

Unfortunately, pm2 doesn’t understand R scripts natively; however, it is possible to specify a custom interpreter. We can use this feature to launch an R-based wrapper for our plumber file using the Rscript scripting front-end that comes with R. The following script will run the two commands listed above.

#!/usr/bin/env Rscript

[library](https://rdrr.io/r/base/library.html)([plumber](https://www.rplumber.io/))

[pr](https://www.rplumber.io/reference/pr.html)("myfile.R") [%>%](https://www.rplumber.io/reference/pipe.html)

[pr\_run](https://www.rplumber.io/reference/pr_run.html)(port=4000, host="0.0.0.0")

# Setting the host option on a VM instance ensures the application can be accessed externally.

# (This may be only true for Linux users.)

Save this R script on your server as something like run-myfile.R. You should also make it executable by changing the permissions on the file using a command like chmod 755 run-myfile.R. You should now execute that file to make sure that it runs the service like you expect. You should be able to make requests to your server on the appropriate port and have the plumber service respond. You can kill the process using Ctrl-c when you’re convinced that it’s working. Make sure the shell script is in a permanent location so that it won’t be erased or modified accidentally. You can consider creating a designated directory for all your plumber services in some directory like /usr/local/plumber, then put all services and their associated Rscript-runners in their own subdirectory like /usr/local/plumber/myfile/.

Introduce Our Service to pm2

We’ll now need to teach pm2 about our Plumber API so that we can put it to work. You can register and configure any number of services with pm2; let’s start with our myfile Plumber service.

You can use the pm2 list command to see which services pm2 is already running. If you run this command now, you’ll see that pm2 doesn’t have any services that it’s in charge of. Once you have the scripts and code stored in the directory where you want them, use the following command to tell pm2 about your service.

pm2 start --interpreter="Rscript" /usr/local/plumber/myfile/run-myfile.R

You should see some output about pm2 starting an instance of your service, followed by some status information from pm2. If everything worked properly, you’ll see that your new service has been registered and is running. You can see this same output by executing pm2 list again.

Once you’re happy with the pm2 services you have defined, you can use pm2 save to tell pm2 to retain the set of services you have running next time you boot the machine. All of the services you have defined will be automatically restarted for you.

At this point, you have a persistent pm2 service created for your Plumber application. This means that you can reboot your server, or find and kill the underlying R process that your plumber application is using and pm2 will automatically bring a new process in to replace it. This should help guarantee that you always have a Plumber process running on the port number you specified in the shell script. It is a good idea to reboot the server to ensure that everything comes back the way you expected.

You can repeat this process with all the plumber applications you want to deploy, as long as you give each a unique port to run on. Remember that you can’t have more than one service running on a single port. And be sure to pm2 save every time you add services that you want to survive a restart.

Run netstat -tulpn to see how the application is being ran. If you see the application on host 127.0.0.0 or 127.0.0.1, the application cannot be accessed externally. You should change the host parameter to 0.0.0.0, for example: `pr\_run(host = “0.0.0.0”).

Logs and Management

Now that you have your applications defined in pm2, you may want to drill down into them to manage or debug them. If you want to see more information, use the pm2 show command and specify the name of the application from pm2 list. This is usually the same as the name of the shell script you specified, so it may be something like pm2 show run-myfile.

You can peruse this information but keep an eye on the restarts count for your applications. If your application has had to restart many times, that implies that the process is crashing often, which is a sign that there’s a problem in your code.

Thankfully, pm2 automatically manages the log files from your underlying processes. If you ever need to check the log files of a service, you can just run pm2 logs run-myfile, where myfile is again the name of the service obtained from pm2 list. This command will show you the last few lines logged from your process, and then begin streaming any incoming log lines until you exit (Ctrl-c).

If you want a big-picture view of the health of your server and all the pm2 services, you can run pm2 monit which will show you a dashboard of the RAM and CPU usage of all your services.

systemd

systemd is the service manager used by certain Linux distributions including RedHat/CentOS 7, SUSE 12, and Ubuntu versions 16.04 and later.

If you use a Linux server you can use systemd to run Plumber as a service that can be accessed from your local network or even outside your network depending on your firewall rules. One of the main advantages of using systemd over using Docker is that systemd won’t bypass firewall rules (Docker does!) and avoids the overhead of running a container.

To implement this option you’ll complete the following three steps from the terminal:

1. Verify that you have the plumber package available globally on the server:

R -e 'install.packages("plumber", repos = "https://cran.rstudio.com/")'

1. Run sudo nano /etc/systemd/system/plumber-api.service, then paste and adapt this content:

[Unit]

Description=Plumber API

# After=postgresql

# (or mariadb, mysql, etc if you use a DB with Plumber, otherwise leave this commented)

[Service]

ExecStart=/usr/bin/Rscript -e "library(plumber); pr('/your-dir/your-api-script.R') %>% pr\_run(port=8080, host='0.0.0.0')"

Restart=on-abnormal

WorkingDirectory=/your-dir/

[Install]

WantedBy=multi-user.target

1. Activate the service (for auto-start on power/reboot) and start it:

sudo systemctl enable plumber-api # automatically start the service when the server boots

sudo systemctl start plumber-api # start the service right now

To check if your API is running, type systemctl | grep running in the terminal and should display plumber-api.service \ loaded active running Plumber API.

Library(rplumber)

list.of.packages <- c("uuid")

new.packages <- list.of.packages[!(list.of.packages %in% installed.packages()[,"Package"])]

if(length(new.packages)) install.packages(new.packages)

require(uuid)

# make sure you put the path of your blockchain.R file

source('blockchain.R')

# Generate a globally unique address for this node

node\_identifier = gsub('-','',UUIDgenerate())

# Instantiate the Blockchain

blockchain = Blockchain()

# genesis block

blockchain$nextBlock(previousHash=1, proof=100)

#\* @serializer custom\_json

#\* @post /transactions/new

function(req)

{

#eg req\_json <- '{"sender": "my address", "recipient": "someone else address", "amount": 5}'

#values <- jsonlite::fromJSON(req\_json)

values <- jsonlite::fromJSON(req$postBody)

# Check that the required fields are in the POST'ed data

required = c('sender','recipient', 'amount')

if (!all(required %in% names(values))) {

return ('Missing Values - sender, recipient and amount are required')

}

index = blockchain$addTransaction(values$sender, values$recipient, values$amount)

list('message' = paste('Transaction will be added to Block', index))

}

#\* @serializer custom\_json

#\* @get /chain

function(req)

{

list('chain'=blockchain$chain, 'length'=length(blockchain$chain))

}

#\* @serializer custom\_json

#\* @get /mine

function(req)

{

# We run the proof of work algorithm to get the next proof

lastBlock = blockchain$lastBlock()

lastProof = lastBlock$block$proof

proof = blockchain$proofOfWork(lastProof)

# We must receive a reward for finding the proof.

# The sender is "0" to signify that this node has mined a new coin.

blockchain$addTransaction(sender="0",recipient = node\_identifier, amount=1)

# Forge the new block by adding it to the chain

previousHash = blockchain$hashBlock(lastBlock)

block = blockchain$nextBlock(proof, previousHash)

list('message'='New block forged', 'index'= block$block$index, 'transactions'= block$block$transactions, 'proof'=block$block$proof,'previousHash'=block$block$previousHash)

# list('message'='New block forged', c('index'= block$block$index, 'transactions'= block$block$transactions, 'proof'=block$block$proof,'previousHash'=block$block$previousHash))

}

#\* @serializer custom\_json

#\* @post /nodes/register

function (req)

{

# req\_json <- '{"sender": "my address", "recipient": "someone else address", "amount": 5}'

values <- jsonlite::fromJSON(req$postBody)

nodes <- values$nodes

if (is.null(nodes))

{

return("Error: the list of nodes is not valid")

}

for (i in 1:length(nodes))

{

blockchain$registerNode(nodes[i])

}

TRUE

}

#\* @serializer custom\_json

#\* @get /nodes/resolve

function (req)

{

replaced = blockchain$handleConflicts()

if (replaced)

{

list('message'='Replaced', 'chain'=blockchain$chain)

} else {

list('message'='Authoritative block chain - not replaceable ', 'chain'=blockchain$chain)

}

}

#\* Log some information about the incoming request

#\* @filter logger

function(req){

cat(as.character(Sys.time()), "-",

req$REQUEST\_METHOD, req$PATH\_INFO, "-",

req$HTTP\_USER\_AGENT, "@", req$REMOTE\_ADDR, "\n")

plumber::forward()

}

#\* @get /chain/show

#\* @html

function(req)

{

render.html <- ""

paste0(render.html, '  
')

render.html <- paste0(render.html, 'Current transactions:  
')

for (i in 1:length(blockchain$currentTransactions))

{

render.html <- paste0(render.html, 'Transaction' , i ,'  
')

render.html <- paste0(render.html, 'sender:', blockchain$currentTransactions[i]$transaction$sender)

render.html <- paste0(render.html, '  
')

render.html <- paste0(render.html, 'recipient:', blockchain$currentTransactions[i]$transaction$recipient)

render.html <- paste0(render.html, '  
')

render.html <- paste0(render.html, 'amount:', blockchain$currentTransactions[i]$transaction$amount)

render.html <- paste0(render.html, '  
')

}

render.html <- paste0(render.html, '  
')

render.html <- paste0(render.html, 'Current transactions:')

render.html <- paste0(render.html, '

')

for (i in 1:blockchain$lastBlock()$block$index)

{

render.html <- paste0(render.html, '  
')

render.html <- paste0(render.html, '**Block nr:**', blockchain$chain[i]$block$index)

render.html <- paste0(render.html, '  
')

render.html <- paste0(render.html, '**Transactions**')

render.html <- paste0(render.html, '  
')

render.html <- paste0(render.html, blockchain$chain[i]$block$transactions)

render.html <- paste0(render.html, 'Proof')

render.html <- paste0(render.html, '  
')

render.html <- paste0(render.html,blockchain$chain[i]$block$proof)

render.html <- paste0(render.html, '  
')

}

render.html <- paste0(render.html, '

')

render.html

}

We stored the code above in a file called “blockchain-node-server.R” to “plumb” it with the script below. First of all, we define a custom Serializer to handle a json boxing issue, as shown [here](https://stackoverflow.com/questions/41965032/r-plumber-json-serializer-auto-unbox).

list.of.packages <- c("plumber","jsonlite")

new.packages <- list.of.packages[!(list.of.packages %in% installed.packages()[,"Package"])]

if(length(new.packages)) install.packages(new.packages)

require(plumber)

require(jsonlite)

custom\_json <- function(){

function(val, req, res, errorHandler){

tryCatch({

json <- jsonlite::toJSON(val,auto\_unbox=TRUE)

res$setHeader("Content-Type", "application/json")

res$body <- json

return(res$toResponse())

}, error=function(e){

errorHandler(req, res, e)

})

}

}

addSerializer("custom\_json",custom\_json)

# Make sure you put the path to your blockchain-node-server.R script

r <- plumb('blockchain-node-server.R')

r$run(port=8000)

library(plumber)

library(jsonlite)

custom\_json <- function(){

function(val, req, res, errorHandler){

tryCatch({

json <- jsonlite::toJSON(val,auto\_unbox=TRUE)

res$setHeader("Content-Type", "application/json")

res$body <- json

return(res$toResponse())

}, error=function(e){

errorHandler(req, res, e)

})

}

}

addSerializer("custom\_json",custom\_json)

r <- plumb("~/Work/myfile.R")

r$run(port=8000)

And Now when I perform an POST request on it using I get

curl -XPOST 'localhost:8000/test

-> {"speech":"aa","source":"bb","displayText":"cc"}

Rplumber implements the [swagger UI](https://swagger.io/swagger-ui/), which can be reached under *http://127.0.0.1:8000/\_\_swagger\_\_/*. The picture below shows our methods as we declared them in the blockchain-node-server.R script:

**Interacting with our Blockchain**

Once we have our Blockchain API running in the url we specified in the plumb command, we can try it with a simple client:

library(jsonlite)

library(httr)

# to register a node

req <- POST("http://127.0.0.1:8000/nodes/register",

body = '{"nodes": "http://127.0.0.1:8000"}')

cat(jsonlite::fromJSON(content(req, "text")))

# create a new transaction

req <- POST("http://127.0.0.1:8000/transactions/new",

body = '{"sender": "d4ee26eee15148ee92c6cd394edd964",

"recipient": "23448ee92cd4ee26eee6cd394edd964", "amount": 15}')

object <- jsonlite::fromJSON(content(req, "text"));object$message

# create a new transaction

req <- POST("http://127.0.0.1:8000/transactions/new",

body = '{"sender": "6eee15148ee92c6cd394edd974d4ee2",

"recipient": "15148ee92cd4ee26eee6cd394edd964", "amount": 225}')

object <- jsonlite::fromJSON(content(req, "text"));object$message

# start mining

req <- GET("http://127.0.0.1:8000/mine")

object <- jsonlite::fromJSON(content(req, "text"));object$message

# create a new transaction

req <- POST("http://127.0.0.1:8000/transactions/new",

body = '{"sender": "334e15148ee92c6cd394edd974d4ee2",

"recipient": "8ee98ee92cd4ee26eee6cd3334e1514", "amount": 887}')

object <- jsonlite::fromJSON(content(req, "text"));object$message

# mine again

req <- GET("http://127.0.0.1:8000/mine")

object <- jsonlite::fromJSON(content(req, "text"));object$message

To try it locally, I opened 2 instances of RStudio: the first one for the server, where rplumber executes the server part, and the second one for the client, from where I fired all the sample requests from the previous script.  
You can check the status of the chain in the browser, as shown in the picture below

But you can also interact with the chain programmatically from the client:

# retrieve the chain

req <- GET("http://127.0.0.1:8000/chain")

chain <- jsonlite::fromJSON(content(req, "text"))

# check the amount of the first transaction in the first block of the chain

chain$chain$block.1$transactions$transaction$amount