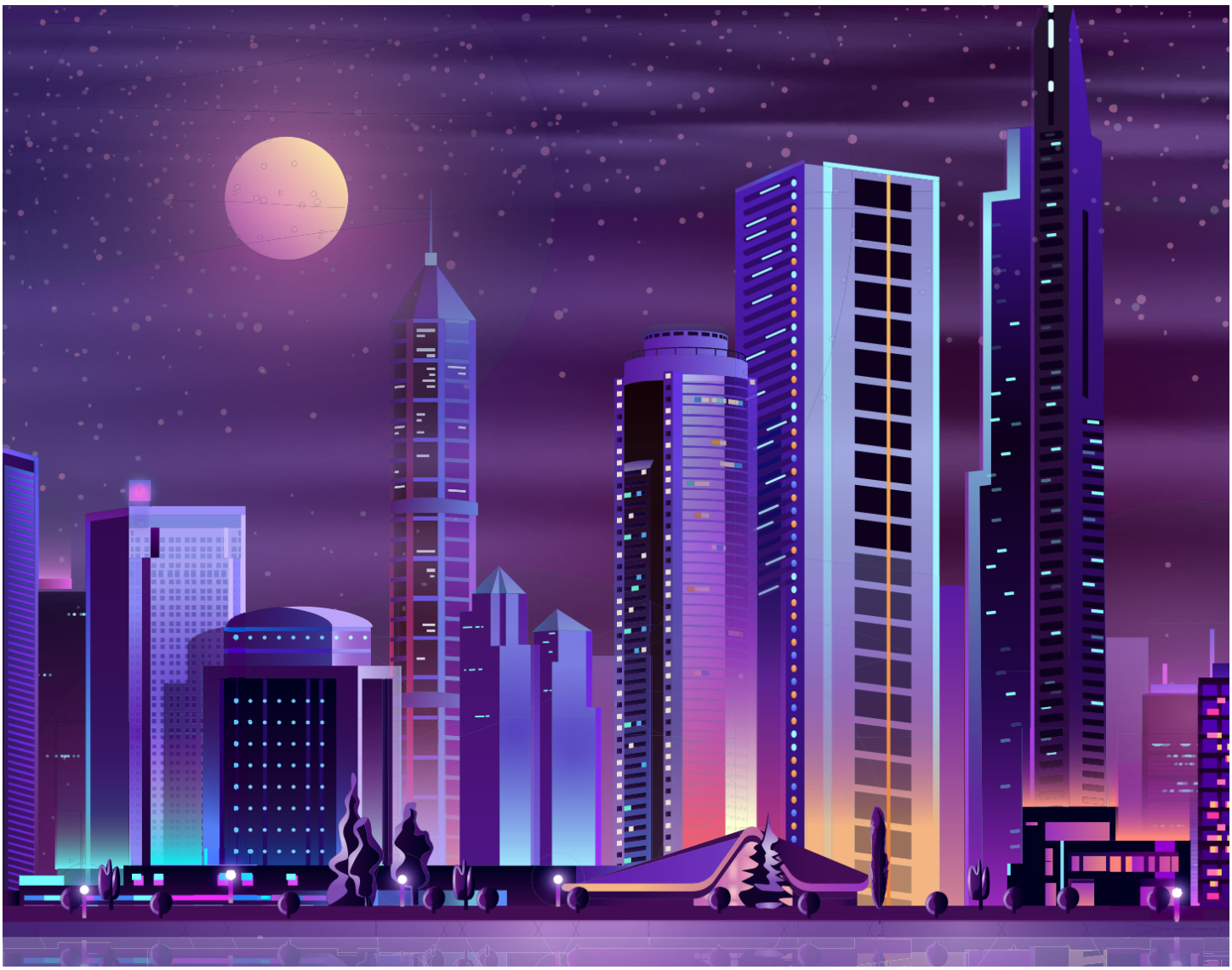
Automating importing of existing infrastructure in terraform.

Craig Godden-Payne
Jun 28 · 6 min read ★



I recently wrote a blog post about importing existing infrastructure into terraform, it

got a lot of views, and somebody pointed out about an elegant tool that could automate the import.

The tool was called *terraformer,* and is much simpler to use, and can automate the reverse engineer back into a usable terraform resource.

It has some caveats, but something this post will explore further.

Importing Existing Infrastructure into Terraform

It is always much simpler to create the terraform first, you would only ever import when you need to do something...

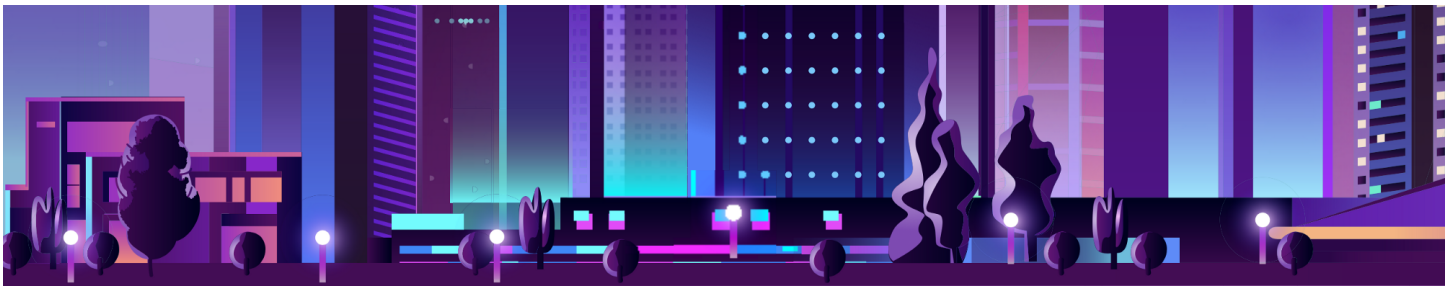
medium.com

So what is terraformer?

> *A CLI tool that generates* `tf` / `json` *and* `tfstate` *files based on existing infrastructure (reverse Terraform).*
>
> *Terraformer uses Terraform providers and is designed to easily support newly added resources. To upgrade resources with new fields, all you need to do is upgrade the relevant Terraform providers.*

I'll go over the exact same scenarios I went through, so you can see exactly how much better and simpler it is to import using terraformer, but first of all;

. . .



Why would you desire to import existing infrastructure?

Because like everything else in life, it is sometimes impossible to plan for the future. Without adequate planning with the creation of infrastructure, it can lead to situations where infrastructure needs to be created manually due to time pressures, emergency releases or just the fact that the infrastructure exists, and terraform was never used in the first instance.

. . .

Example: You have already defined the resource and want to tell the state that this resource already exists.

> *Imagine that something was going wrong in production, and a change had to be applied quickly to prevent an outage. A change was added manually in route53 to add a DNS*

> *record.*
>
> *Once things had settled down, the same record was defined as a terraform resource, but when apply is ran, a messages is returned to say that the resource already exists. It causes the apply stage to fail.*

What needs to happen, is to import the state with the existing resource, so that next time a terraform apply is run, the terraform software will consider the resource in its state. Going forward, this means any changes made will be picked up as modifications, rather than additions.

In this hypothetical situation, let us imagine that the following resources were created from within the AWS console:

```
Route53 Record Set Name: www.mywebsite.com.
Route53 Record Set Type: CNAME
Route53 Record Set Value: mywebsite.com.
```


Now since the three resources are straightforward, and it is known what exactly was created, they can be added into your terraform project:

```
resource aws_route53_record www {
  name = "www.mywebsite.com"
  type = "CNAME"
  zone_id = aws_route53_zone.zone.id
  records = ["mywebsite.com"]
  ttl = 300
}

resource aws_route53_zone zone {
  name         = "mywebsite.com"
}
```

The error message when the terraform is applied would look something like this:

```
* aws_route53_record.www: 1 error(s) occurred:

* aws_route53_record.www: [ERR]: Error building changeset:
InvalidChangeBatch: RRSet of type CNAME with DNS name
www.mywebsite.com. is not permitted as it conflicts with other
records with the same DNS name in zone mywebsite.com.
        status code: 400
```

Terraform will exit at this point because of the conflict.

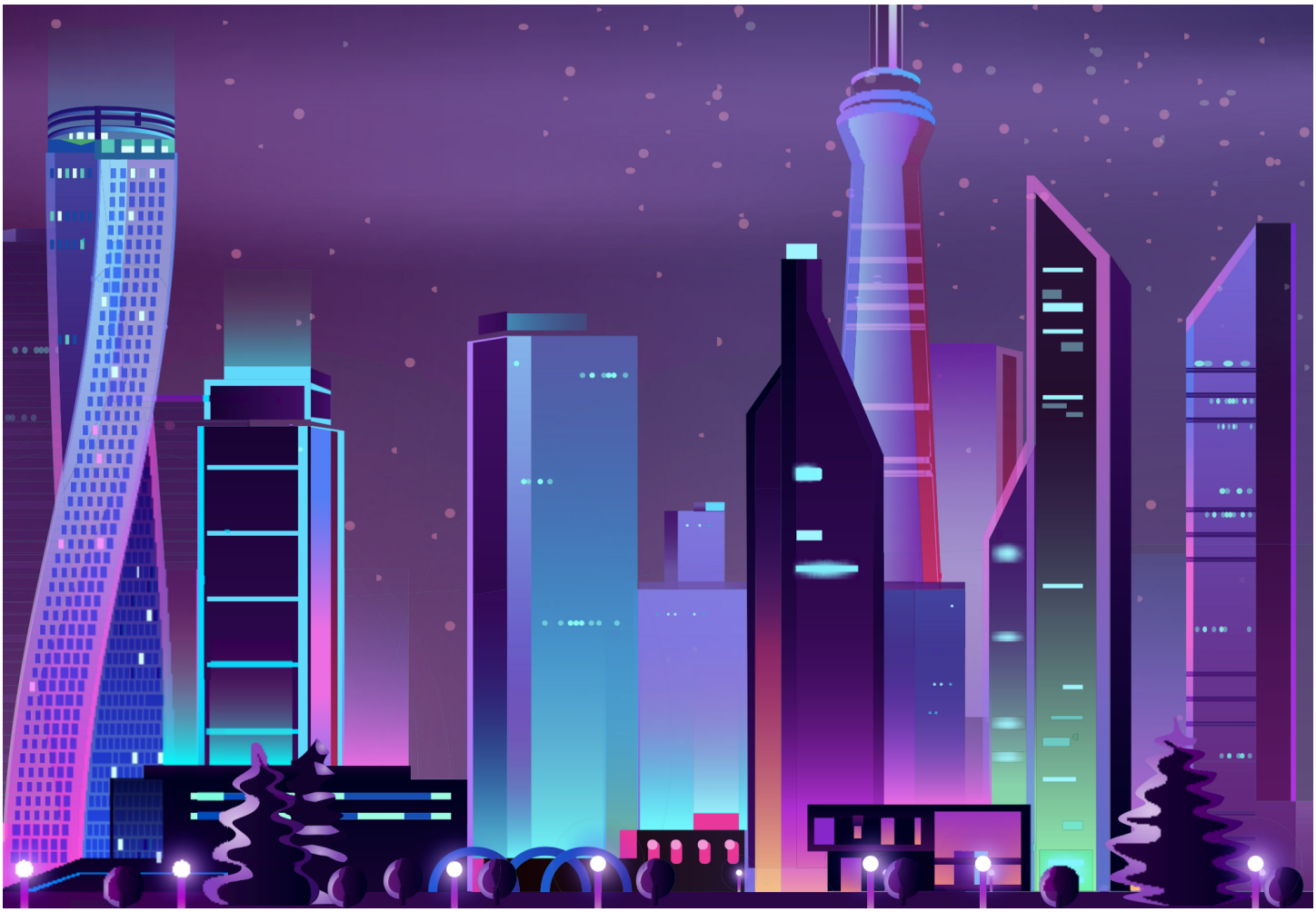
To import the state and even the existing resource file (if you no longer have it), the following Terraformer CLI command can be run:

```
AWS_PROFILE=craig terraformer import aws --resources=route53 --
filter=aws_route53_record=mywebsite.com --regions=eu-west-2
```

terraformer documentation

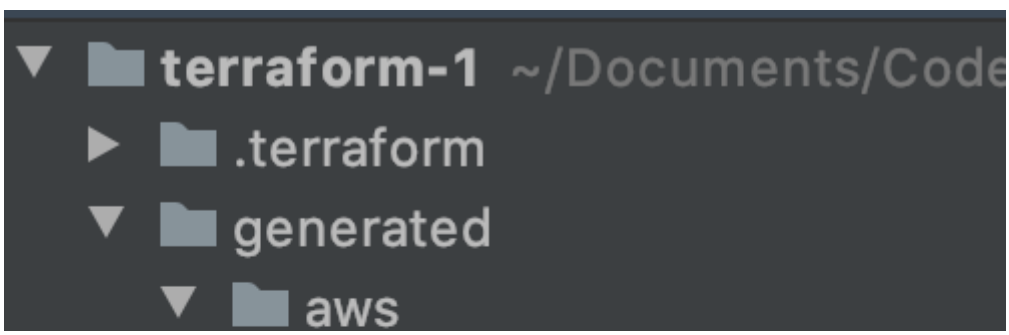
The beauty with terraformer is its use of filters over the naming convention terraform uses.

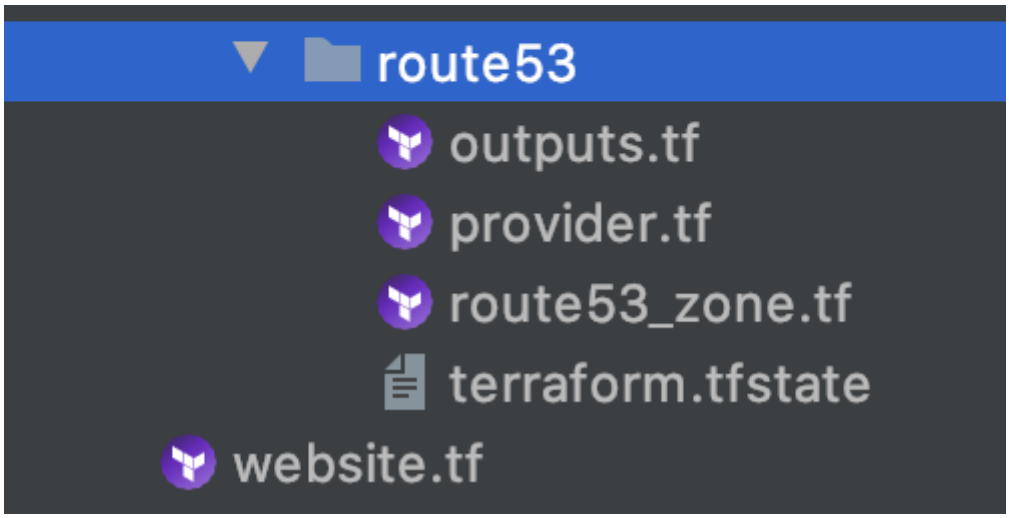
You can pretty much guess the resource name based on the name you gave it, whereas, with terraform, this will have to be in a particular format.



The import took a few seconds, but the output in the window was quite helpful

```
2020/06/28 21:50:14 aws importing default region
2020/06/28 21:50:14 aws importing... route53
2020/06/28 21:50:17 Refreshing state... aws_route53_zone.tfer--
Z0621280100AQL6BP58RC_mywebsite-002E-com
2020/06/28 21:50:19 aws Connecting....
2020/06/28 21:50:19 aws save route53
2020/06/28 21:50:19 aws save tfstate for route53
```





A directory structure is created, from which you can get all the information you need

The route53_zone contained a similar-looking definition to my resource, and the terraform.tfstate file contains state information.

You have to bear in mind though, that the terraform.tfstate only contains state information for the filtered resources. If you use this in place of an existing terraform state, you will likely remove resources that already have state.

The point of this state file seems to be that you could now get the required section from the state file, and paste it into your existing file.

Creating a Simple, Low-Cost Twitter Bot, utilising Serverless Technologies.

People have a love-hate relationship with twitter bots.

medium.com

Here are the files for reference:

Code file:

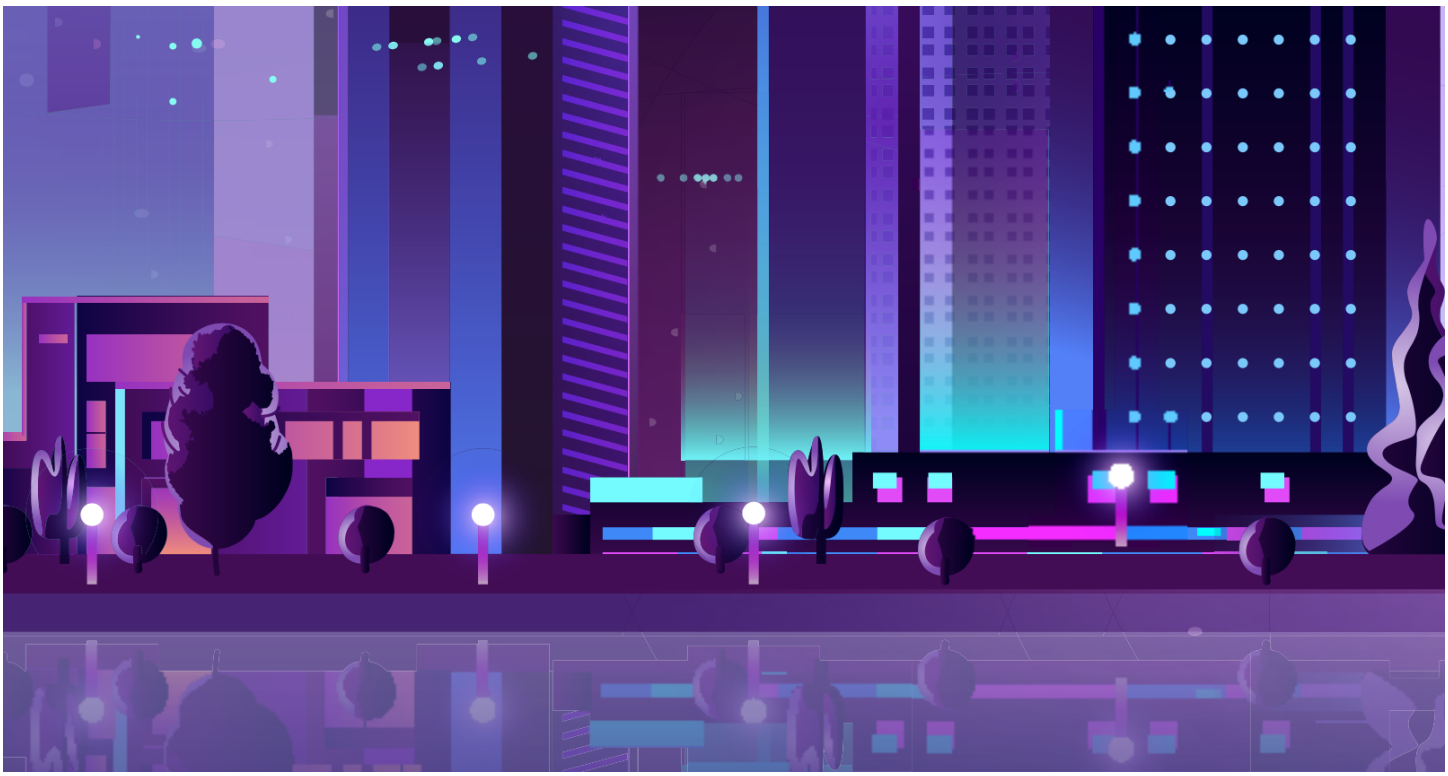
```
resource "aws_route53_zone" "tfer--Z0621280100AQL6BP58RC_mywebsite-
002E-com" {
  comment       = "Managed by Terraform"
  force_destroy = "false"
  name          = "mywebsite.com."
}
```

State file:

```
{
    "version": 3,
    "terraform_version": "0.12.18",
    "serial": 1,
    "lineage": "17034a7a-eadd-b496-c4e3-0ca3639e33ee",
    "modules": [
        {
            "path": [
                "root"
            ],
            "outputs": {
                "aws_route53_zone_tfer--
Z0621280100AQL6BP58RC_mywebsite-002E-com_id": {
                    "sensitive": false,
                    "type": "string",
                    "value": "Z0621280100AQL6BP58RC"
                }
            },
            "resources": {
                "aws_route53_zone.tfer--
Z0621280100AQL6BP58RC_mywebsite-002E-com": {
                    "type": "aws_route53_zone",
                    "depends_on": [],
                    "primary": {
                        "id": "Z0621280100AQL6BP58RC",
                        "attributes": {
                            "comment": "Managed by Terraform",
                            "delegation_set_id": "",
                            "force_destroy": "false",
                            "id": "Z0621280100AQL6BP58RC",
```

```
                                "name": "mywebsite.com.",
                                "name_servers.#": "4",
                                "name_servers.0": "ns-1428.awsdns-
50.org",
                                "name_servers.1": "ns-1616.awsdns-
10.co.uk",
                                "name_servers.2": "ns-307.awsdns-
38.com",
                                "name_servers.3": "ns-944.awsdns-
54.net",
                                "tags.%": "0",
                                "vpc.#": "0",
                                "zone_id": "Z0621280100AQL6BP58RC"
                            },
                            "meta": {
                                "schema_version": 0
                            },
                            "tainted": false
                        },
                        "deposed": [],
                        "provider": "provider.aws"
                    }
                },
                "depends_on": []
            }
        ]
    }
```


. . .

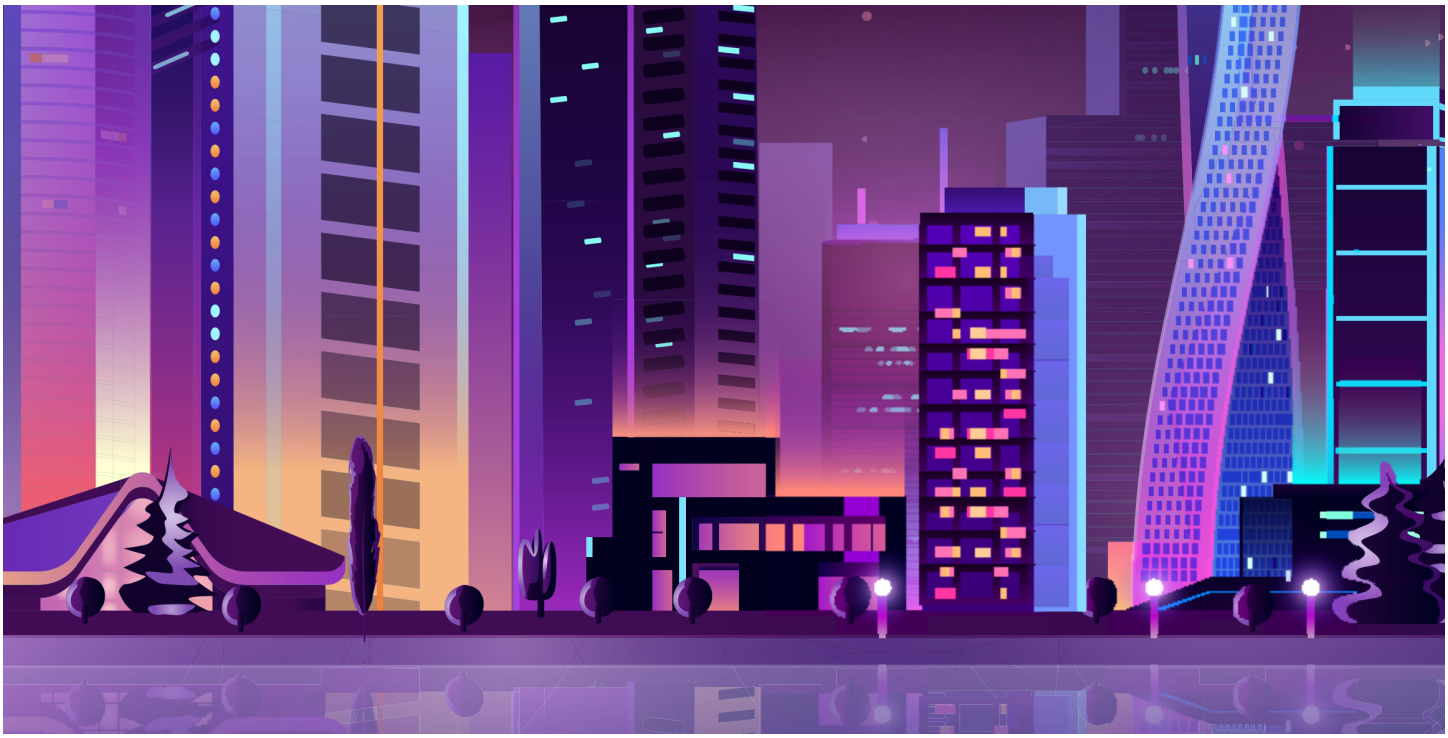


Conclusion

Terraformer definitely has its uses, although for everyday import of a single resource, alternative methods may be quicker to use.

I can see its main purpose would be for importing a larger amount of infrastructure.

I would be interested to see this in a larger capacity, and something I will definitely be trying out next time I need to import more than a single resource!



Graphics Attribution:
https://www.freepik.com/free-photos-vectors/background
vectorpocket

Terraform Infrastructure DevOps Code Software Development