Importing Existing Infrastructure into Terraform

Craig Godden-Payne
Jun 22, 2020 · 6 min read ★

Terraform is a powerful tool to have in your toolset.

It's effortless to use, for creating new infrastructure, but not so much for importing existing infrastructure, and hopefully, this post will demystify some of these complexities!

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.

> It's worth reiterating that its always much simpler to create the terraform first, you would only ever import when you need to do something reactive, like an emergency release

Before the terraform import is run, two places can be used as a starting point:

- The terraform resource definition exists in code and just needs to be imported.
- The terraform resource does not exist; you need to import it so that you can backfill the terraform resource.

How can we do this?

At present, it is not possible to directly take an AWS resource and import it into a terraform resource definition. Still, it is possible to import into a state equivalent and then convert that into a terraform resource definition.

Scenario One — You have already defined the resource and want to tell the state that this resource already exists.

This situation is the easiest to work with, as you already have the resource definition defined.

> *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 = "${data.aws_route53_zone.zone.id}"
  records = ["mywebsite.com"]
  ttl = 300
}

data "aws_route53_zone" "zone" {
  name         = "mywebsite.com"
  private_zone = false
}
```

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.

medium.com

To resync the state with what exists back to the resource, the following Terraform CLI commands can be run:

```
AWS_PROFILE=mywebsite terraform import aws_route53_record.www
Z0ZZZZZZ0ZZZZ0_www.mywebsite.com_CNAME

Which corresponds to:
AWS_PROFILE={AwsProfileName} terraform import {resource_type}.
{resource_name} {zone_id}_{record_name}_{record_type}
```

terraform documentation

The state will then be updated, and the CLI will print a message like:

```
aws_route53_record.www: Importing from ID
"Z0ZZZZZZ0ZZZZ0_www.mywebsite.com_CNAME"...
aws_route53_record.www: Import complete!
  Imported aws_route53_record (ID:
Z0ZZZZZZ0ZZZZ0_www.mywebsite.com_CNAME)
aws_route53_record.www: Refreshing state... (ID:
Z0ZZZZZZ0ZZZZ0_www.mywebsite.com_CNAME)

Import successful!

The resources that were imported are shown above. These resources
are now in
your Terraform state and will henceforth be managed by Terraform.
```

If an error is returned, then something must be incorrect, check the documentation to make sure the syntax is correct:

```
aws_route53_record.www: Importing from ID
"Z0ZZZZZZ0ZZZZ0_www.mywebsite.com_CNAME"...
aws_route53_record.www: Import complete!
  Imported aws_route53_record (ID:
Z0ZZZZZZ0ZZZZ0_www.mywebsite.com_CNAME)

Error: aws_route53_record.www (import id:
Z0ZZZZZZ0ZZZZ0_www.mywebsite.com_CNAME): Can't import
aws_route53_record.www, would collide with an existing resource.

Please remove or rename this resource before continuing.
```

2 — A resource has not been defined, and we need to build a terraform resource from an existing state.

This will usually happen when something like an EC2 instance is created, but it is not possible to get the record of what settings were used etc.

> *Imagine that something went wrong, and you had to quickly migrate from a physical server to EC2.*
>
> *You spin up an EC2 and applied a load of settings.*

> *Once things settled down after the deployment, you wanted to build the terraform and sync the state so that it can be managed via terraform going forward.*

What needs to happen is we need to understand what currently exists in AWS, so that we can build a terraform resource, so that it can be imported.

In this scenario, I will work with the hypothetical AWS resource:

```
EC2 instance Name: mywebsite-server
```


In order to import, a terraform resource will need to be created within your terraform project, with a matching type to be able to do the import. This will look something like:

```
resource "aws_instance" "mywebsite-server" {
}
```

It is then possible to run the import, based on what is described in the terraform documentation:

```
AWS_PROFILE=mywebsite terraform import aws_instance.mywebsite-server
i-0Z000ZZ0Z0Z00Z0Z0

Which corresponds to:
AWS_PROFILE={AwsProfileName} terraform import {resource_type}.
{resource_name} {instance_id}
```

When this is run, it will show this within the CLI window.

```
aws_instance.mywebsite-server: Importing from ID "i-
0Z000ZZ0Z0Z00Z0Z0"...
aws_instance.mywebsite-server: Import complete!
  Imported aws_instance (ID: i-0Z000ZZ0Z0Z00Z0Z0)
aws_instance.mywebsite-server: Refreshing state... (ID: i-
0Z000ZZ0Z0Z00Z0Z0)

Import successful!

The resources that were imported are shown above. These resources
are now in
your Terraform state and will henceforth be managed by Terraform.
```

Now it is possible to reverse engineer the state file into what will eventually be the terraform resource. Look at the structure below, and it becomes clear how we might do this:

```
"resources": {
                "aws_instance.mywebsite-server": {
                    "type": "aws_instance",
                    "depends_on": [],
                    "primary": {
                        "id": "i-0Z000ZZ0Z0Z00Z0Z0",
                        "attributes": {
                            "ami": "ami-zzz00zz0",
                            "arn": "arn:aws:ec2:eu-west-
2:XXXXXXXXXXXX:instance/i-0Z000ZZ0Z0Z00Z0Z0",
                            "associate_public_ip_address": "true",
                            "availability_zone": "eu-west-2a",
                            "cpu_core_count": "1",
                            "cpu_threads_per_core": "1",
                            "credit_specification.#": "1",
                            "credit_specification.0.cpu_credits":
"standard",
                            "disable_api_termination": "false",
                            "ebs_block_device.#": "0",
                            "ebs_optimized": "false",
                            "ephemeral_block_device.#": "0",
                            "get_password_data": "false",
                            "iam_instance_profile": "",
                            "id": "i-0Z000ZZ0Z0Z00Z0Z0",
                            "instance_state": "running",
                            "instance_type": "t2.micro",
                            "ipv6_addresses.#": "0",
                            "key_name": "mywebsite",
                            "monitoring": "false",
```

```
                            "network_interface.#": "0",
                            "network_interface_id": "eni-
00zzzzz00zz000z00",
                            "password_data": "",
                            "placement_group": "",
                            "primary_network_interface_id": "eni-
00zzzzz00zz000z00",
                            "private_dns": "ip-1-1-1-1.eu-west-
2.compute.internal",
                            "private_ip": "1.1.1.1",
                            "public_dns": "ec2-1-1-1-1.eu-west-
2.compute.amazonaws.com",
                            "public_ip": "1.1.1.1",
                            "root_block_device.#": "1",

"root_block_device.0.delete_on_termination": "true",
                            "root_block_device.0.iops": "0",
                            "root_block_device.0.volume_id": "vol-
0z00zz0zzz000z0zz",
                            "root_block_device.0.volume_size": "8",
                            "root_block_device.0.volume_type":
"standard",
                            "security_groups.#": "0",
                            "source_dest_check": "true",
                            "subnet_id": "subnet-000000z0000z00z0z",
                            "tags.%": "1",
                            "tags.Name": "MyWebsite",
                            "tenancy": "default",
                            "user_data":
"zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz",
                            "volume_tags.%": "0",
                            "vpc_security_group_ids.#": "1",
                            "vpc_security_group_ids.3064782471":
"sg-00zz00zzz00zz0000"
                        },
                        "meta": {
                            "e2bfb730-ecaa-11e6-8f88-34363bc7c4c0":
{
                                "create": 600000000000,
                                "delete": 1200000000000,
                                "update": 600000000000
                            },
                            "schema_version": "1"
                        },
                        "tainted": false
                    },
                    "deposed": [],
                    "provider": "provider.aws"
                }
            },
```


Use the terraform documentation to work out which fields need to be populated, and use the values from within the state.

Be wary though, you can't set some properties, as they are autogenerated, so it is worth running a plan to see if your import looks right after converting into the terraform resource.

> Automating importing of existing infrastructure in terraform.
>
> An elegant tool exists, which can reverse engineer a cloud resource into terraform without an explicit import. Its...
>
> medium.com

Graphic Attributions:
https://www.freepik.com/free-photos-vectors/car

Terraform Infrastructure As Code DevOps Best Practices