Infrastructure as Code (IaC)

Dr Peadar Grant

January 24, 2024

1 Infrastructure as Code

- Infrastructure as Code in the broad sense refers to stored textual descriptions of system configurations.
- This applies to any computing system (e.g. desktop configuration, physical server configuration) as well as cloud systems.
- Other terms: desired state configuration.

1.1 Imperative vs declarative

IaC can be broadly broken down into imperative / procedural vs declarative.

Consider the basic act of making a cup of tea: StackOverflow question)

Imperative / **Procedural** is where we specify instructions to be carried out sequentially to achieve the state we want.

- "(1) Go to kitchen. (2) Get sugar, milk, and tea. (3) Mix them (4) heat over the fire till it boils (5) Put that in a cup and bring it to me"
- Examples: PowerShell Script, Java program

Declarative is where we specify the state we want to achieve, rather than how to do it.

- "Get me a cup of tea."
- Examples: SQL, HTML.

Many people incorrectly assume IaC to be automatically declarative when mentioned. Often tools offer a blend of imperative and declarative input.

1.2 Requirements

For IaC to be an option:

1. Desired state of the system (resources, configuration of each, connections to each, permissions) needs to be known in a form that can be written down / diagrammed.
2. All of the resource providers must allow some method of automated configuration E.g. web API (Zoom API), command-line tools (AWS CLI), text-based serial console (network hardware).

From now on we will assume that we are trying to automate AWS resource creation only.

1.3 Tooling

Standard scripting environments including Bash, PowerShell that can utilise the AWS CLI.

General purpose IaC tools like Ansible, CFEngine, Chef, Puppet, cloud-init, EC2 launch that are designed to run on servers (virtual or physical).

- Will meet cloud-init / EC2Launch later on!

Cloud-first IaC tools like AWS CloudFormation or Terraform that are built primarily for automating cloud configuration and can interact directly with cloud providers.

Can encounter more complex configurations that are a hybrid of tools.

2 Shell-based automation

We have already encountered shell-based automation. Some important practices:

2.1 Dynamic lookup

Required names (AMIs, regions, Account ID) should be looked up dynamically as far as possible. Assume that resources may change. Don't hard-code anything that can be looked up.

As a side note, it's usually best to look up anything ONCE in your script and re-use that value for the duration of the script.

2.2 Capture output

AWS returns information in response to commands. Needs to be captured appropriately (using `ConvertFrom-Json` or `jq`).

Derived data (like ARNs) are often best looked up rather than created, as the rules may change.

2.3 Idempotency

An operation is deemed to be idempotent if it can be applied multiple times without changing the result beyond its original application. In the context of a scripted setup, it should:

- Set up the defined environment if it doesn't already exist.
- Complete the setup of the defined environment if it already partially exists.
- Do nothing if everything is already as it should be.
- Handles changes if required: either by modification (harder, sometimes required), delete / recreation (easier in principle, problems if resources contain data (e.g. S3, databases).

Dealing with partial setup and/or changes is difficult. As an alternative, it should at least prevent re-running script if to do so would cause clashes.

2.4 Facilitate troubleshooting

Scripts should ease troubleshooting by:

1. **Labelling resources** as far as possible to facilitate visual inspection in GUI / manual CLI. Often done using the Tag facilities.
2. **Logging error messages** when they occur (or at least not hiding them!)

3 CloudFormation

CloudFormation is a declarative IaC tool available in AWS. Uses **templates** to create **stacks**.

AWS CloudFormation best practices: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-pract

3.1 Templates

Templates specifies the resources, their configuration, permissions etc.

- Templates are written in YAML (or JSON).
- Any editor can be used to make a template. As an alternative:
  - templates can be constructed visually using AWS CloudFormation Designer.
  - could write a tool to output the correct template

3.2 Stacks

A stack is an instance of a template created by CloudFormation.

- There may be multiple stacks in existence from the same template.

4 CloudFormation example

4.1 Basic template

Consider we have the file `s3_buckets_template.yaml`:

```yaml
Resources: # every template must have Resources

  Bucket1: # logical name (in CF GUI and template only)
    Type: AWS::S3::Bucket # list available on AWS guides

  Bucket2: # another resource
    Type: AWS::S3::Bucket
```

1. As a minimum, template will contain a `Resources` object with at least one child.
2. Every `Resource` has a `Type` from the: AWS resource and property types reference
3. Usually contains a `Properties` object. Required properties depend on the resource type.

We can then create a stack based on this template from the console or CLI.

4.2 Stack creation

To create a stack based on a given template, specify the template file (on our local computer) and give the stack a name:

```
aws cloudformation create-stack --stack-name buckets --template-body file://s3_buckets_template.json
```

Note:

1. The `StackId` is returned as an ARN in the JSON from this command.
2. The command returns immediately but stack creation takes time.

If we list our buckets we can see the newly created buckets from the stack:

```
aws s3api list-buckets
```

Notice how CloudFormation generates a unique name for the buckets derived from the Resource names.

4.3 List stacks

Can list stacks created in our account on the console and command-line:

```
aws cloudformation list-stacks
```

4.4 Multiple instances

A template can be instantiated multiple times in different stacks:

- Use same template file (no need to duplicate / copy)
- Give the stack a different name

Things to watch out for:

1. Avoid forcing resource names (e.g. `QueueName` property)

4.5 Stack deletion

To delete a stack (i.e. all of its resources) we can issue the command:

```
aws cloudformation delete-stack --stack-name buckets
```

5 Drift detection

Consider a situation where the actual resources on AWS are modified outside of CloudFormation. This leads to a situation known as stack drift. Detecting these is a two-step process

```
# detection
aws cloudformation detect-stack-drift --stack-name buckets

# display
aws cloudformation describe-stack-resource-drifts --stack-name buckets
```

If, for example we deleted one of the buckets manually:

```
# delete bucket (created by cloudformation, but name will vary)
aws s3api delete-bucket --bucket buckets-mypublicbucket-xge67cn5a1kv

# re-do detection and display
aws cloudformation detect-stack-drift --stack-name buckets
aws cloudformation describe-stack-resource-drifts --stack-name buckets
```

```
# notice DELETED status on MyPublicBucket
```