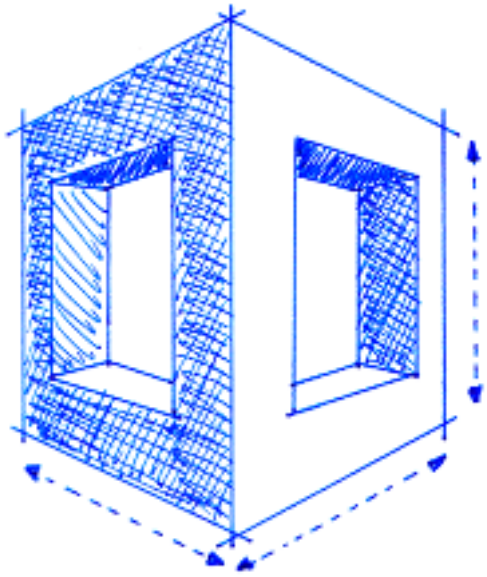
# StackFormation Documentation

***Release dev-master***

**xxx**

**Feb 22, 2018**

# Getting started Documentation



**Lightweight AWS CloudFormation Stack, Template and Parameter Manager and Preprocessor**

Deploying CloudFormation stacks to AWS can be done using the AWS Console, AWS Cli or any SDK. While this is perfectly ok it can be a challenge to keep track of what template is used for what stack and manage the input parameters. This is where "StackFormation" comes in.

StackFormation (note the wordplay: CloudFormation / Stacks) will read *blueprint.yml* files that contains information about stacks, the templates they use and their input parameters. It also allows you to query values for input parameters from other stack's resources or outputs. In addition to that StackFormation makes it easy to embed scripts into UserData.

If you have any questions please feel free to contact us:

- Fabrizio Branca
- Julian Kleinhans

This version of the documentation covering StackFormation dev-master has been rendered at: Feb 22, 2018

CHAPTER 1

# Getting Started

## 1.1 Installation/Usage

### 1.1.1 Via composer

Install composer first, then:

```
$ composer require aoepeople/stackformation
```

### 1.1.2 Via docker

Example .. code-block:: shell

    $ docker run –rm -it -v $(pwd):/app -w /app kj187/stackformation:latest setup $ docker run –rm -it -v $(pwd):/app -w /app kj187/stackformation:latest blueprint:deploy

Or if you use lambda with golang for instance

For more details, see https://hub.docker.com/r/kj187/stackformation/

### 1.1.3 Using the phar

Grab the latest release from https://github.com/AOEpeople/StackFormation/releases/latest or use this shortcut (requires `jq` to be installed)

```
$ wget $(curl -s https://api.github.com/repos/AOEpeople/StackFormation/releases/
↪latest | jq -r '.assets[0].browser_download_url')
```

**Tip:** If you want to use StackFormation globally:

```
$ mv stackformation.phar /usr/local/bin/stackformation
$ chmod +x /usr/local/bin/stackformation
```

## 1.2 Quickstart

### 1.2.1 AWS access keys

Execute the setup command to add all necessary AWS env vars

```
$ vendor/bin/stackformation.php setup
```

Add it to your gitignore: `echo .env.default >> .gitignore`

### 1.2.2 Create a blueprint

Create a `blueprints.yml` in your project directory:

```
blueprints:
  - stackname: my-stack
    template: my-stack.template
```

### 1.2.3 Create a CloudFormation template

Create a CloudFormation template `my-stack.template` in your project directory:

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Resources": {
    "MyResource1": { "Type": "AWS::CloudFormation::WaitConditionHandle" }
  }
}
```

### 1.2.4 Deploy your stack

```
$ bin/stackformation.php deploy my-stack
```

## 1.3 Kickstart a project

Imagine we are starting from scratch, on a so called green field.

```
$ mkdir DemoProject
$ cd DemoProject
```

### 1.3.1 Installation

First of all, we have to install StackFormation, for this demo we will da that via composer

```
$ composer require aoepeople/stackformation
```

Your first level project structure should be looking like that now

```
├── composer.json
├── composer.lock
└── vendor
```

To check if StackFormation is working properly execute the following command

```
$ vendor/bin/stackformation.php
```

You should see all available StackFormation commands and options now.

### 1.3.2 Required environment settings

Execute the setup command to add all necessary AWS env vars

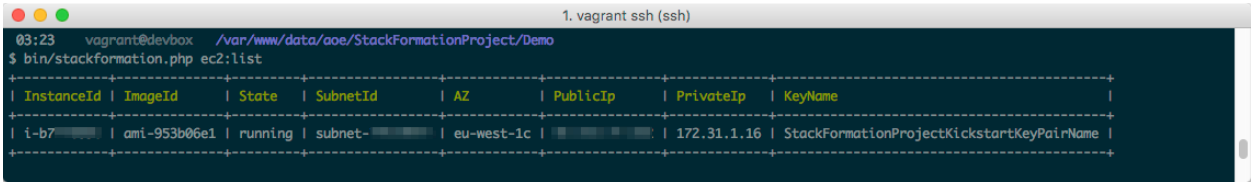
```
$ vendor/bin/stackformation.php setup
```

Add it to your gitignore: `echo .env.default >> .gitignore`

### 1.3.3 Short check

If your access and secret key are correct and the user behind that have enough permissions, you are now able to use the whole magic of StackFormation. Just a quick example, you want to know what and how many ec2 instances are currently running?

```
$ vendor/bin/stackformation.php ec2:list
```




### 1.3.4 Your first blueprint

Create a `blueprints.yml` in your current directory:

```
blueprints:
  - stackname: my-stack
    template: my-stack.template
```

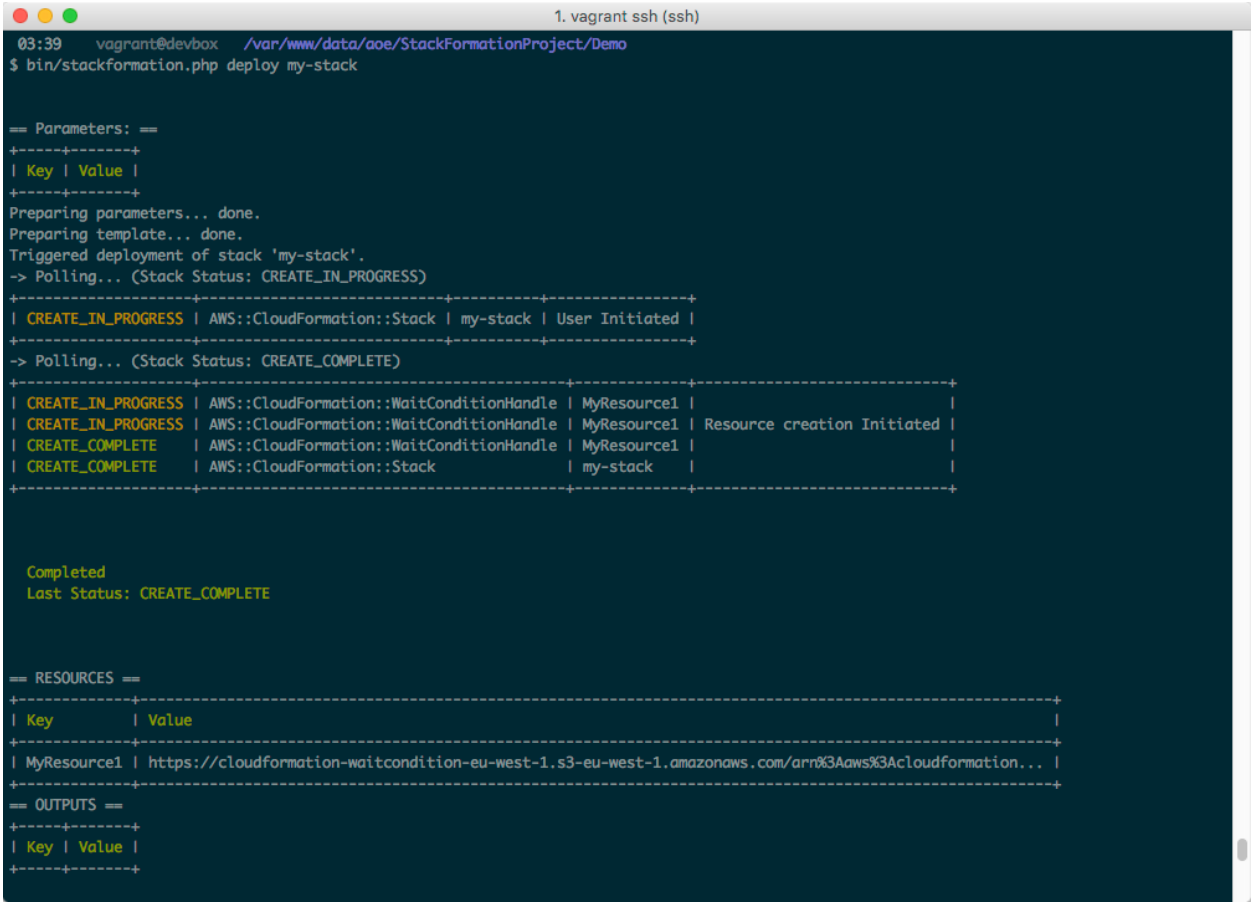
Create you CloudFormation template `my-stack.template`:

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Resources": {
    "MyResource1": { "Type": "AWS::CloudFormation::WaitConditionHandle" }
  }
}
```

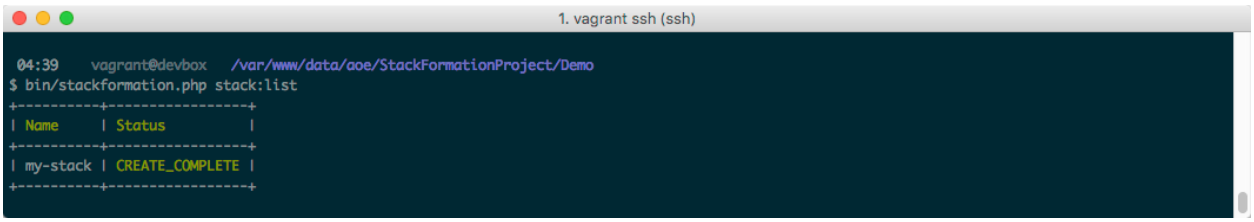
### 1.3.5 Deploy your stack

```
$ vendor/bin/stackformation.php deploy my-stack
```

The output should be the following

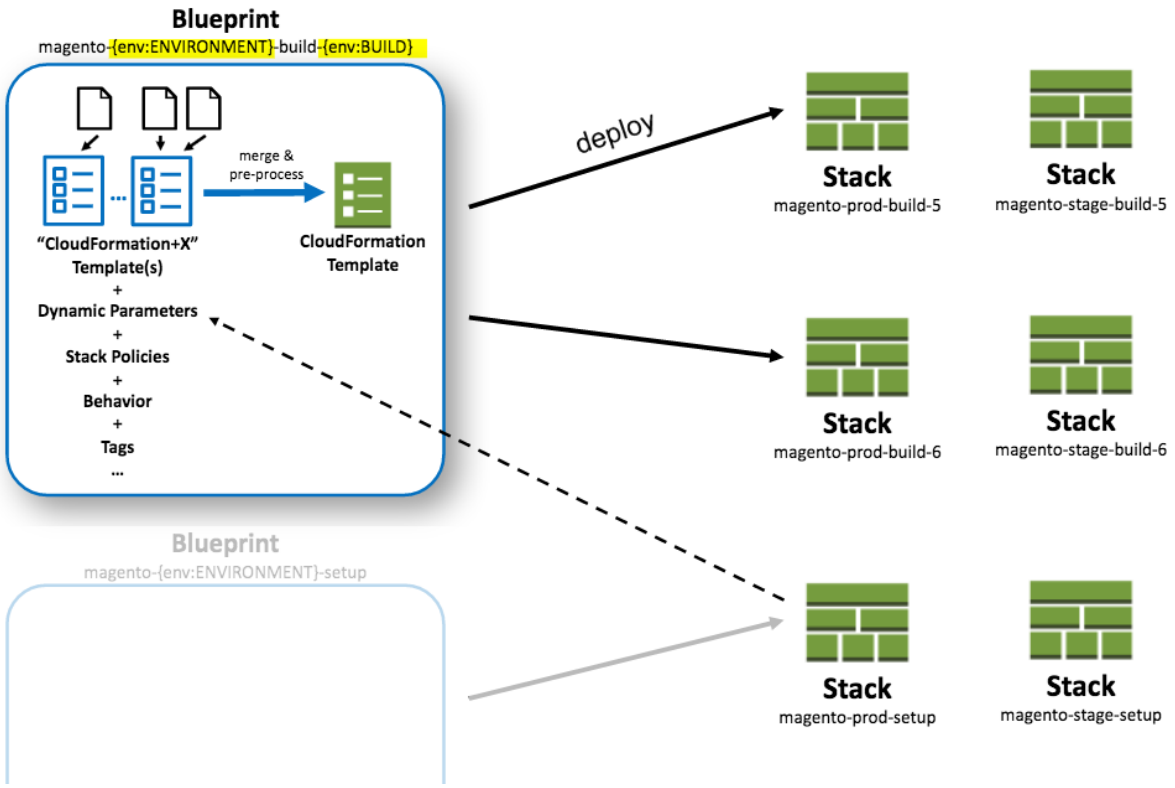



```
$ vendor/bin/stackformation.php stack:list
```



CHAPTER 2

# Blueprints




## 2.1 Structuring your blueprints

Structure your blueprints including all templates and other files (e.g. userdata) in "modules". StackFormation will load all stack.yml files from following locations:

- `blueprints/*/*/*/blueprints.yml`
- `blueprints/*/*/blueprints.yml`
- `blueprints/*/blueprints.yml`
- `blueprints/blueprints.yml`
- `blueprints.yml`

So it's suggested to create a directory structure like this one:

```
blueprints/
  stack1/
    userdata/
      provisioning.sh
    blueprints.yml
    my.template
  stack2/
    blueprints.yml
  ...
```

All `blueprints.yml` files will be merged together.

## 2.2 Parameters

### 2.2.1 Adding parameters

Add parameters in your `my-stack.template`:

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Parameters: {
    "MyParameter1": { "Type": "String" }
  },
  "Resources": {
    "MyResource1": { "Type": "AWS::CloudFormation::WaitConditionHandle" }
  }
}
```

... and configure that parameter in the `blueprint.yml` file:

```
blueprints:
  - stackname: my-stack
    template: my-stack.template
    parameters:
      MyParameter1: 'Hello World'
```

### 2.2.2 Parameter values

| Parameter | Syntax | Result |
|---|---|---|
| Output lookup | `{output:<stack>:<output>}` | Output value |
| Resource lookup | `{resource:<stack>:<logicalId of the resource>}` | Physical Id of that resource |
| Parameter lookup | `{parameter:<stack>:<parameterName>}` | Parameter value (note that some parameters will not be shown if they're 'no_echo') |
| Env var lookup | `{env:<var>}` | Value of environment variable `var` |
| Env var lookup with fallback | `{env:<var>:<defaultValue>}` | Value of environment variable `var` falling back to `defaultValue` if env var is not set |
| Stack/global variable lookup | `{var:<var>}` | Value variable `var` |
| Current timestamp | `{tstamp}` | e.g. `1453151115` |
| MD5 sum | `{md5:<filename>}` | e.g. `fdd747e9989440289dcfb476c75b4268` |
| Clean | `{clean:2.1.7}` | `217` removes all characters that arent allowed in stack names |
| Switch profile | `[profile:<profileName>:..]` | Will switch to a different profile and evaluate the second parameter there. This is useful in cross account setups |

Output and resource lookup allow you to "connect" stacks to each other by wiring the output or resources created in one stack to the input parameters needed in another stack that sits on top of the first one without manually managing the input values.

Example

```
blueprints:
  - stackname: stack1-db
    template: templates/stack1.template
    [...]
  - stackname: stack2-app
    template: templates/stack2.template
    parameters:
      build: 's3://{output:stack1:bucketName}/{env:BUILD}/build.tar.gz'
      db: '{output:stack1-db:DatabaseRds}'
```

Variables (global/local, nested into other placeholders)

```
vars:
  KeyPair: 'mykeypair'

blueprints:
  - stackname: mystack
    vars:
      ParentStack: 'MyParentStack'
    parameters:
      KeyPair: '{var:mykeypair}'
      Database: '{output:{var:ParentStack}:DatabaseRds}'
    [...]
```

Switch Profile Example (in this example an AMI is baked in a different account and shared with this account)

```
blueprints:
  - stackname: mystack
    parameters:
      BaseAmi: '[profile:myDevAccountProfile:{output:bakestack:BaseAmi}]'
```

### 2.2.3 Conditional parameter values

You might end up deploying the same stacks to multiple environments or accounts. Instead of duplicating the blueprints (or using YAML reference) you'll probably want to parameterize your blueprints like this

```
blueprints:
  - stackname: 'app-{env:Environment}-build'
    template: 'build.template'
    parameters:
      KeyPair: 'MyKeyPair'
    [...]
```

... and then before deploying (locally or from your CI server) you'd set the env var first and then deploy:

```
$ export Environment=prod
$ vendor/bin/stackformation.php blueprint:deploy 'app-{env:Environment}-build'
```

But in many cases those stacks do have some minor differences in some of the parameters (e.g. different VPCs or KeyNames,...) You could solve it like this with nested placeholders:

```
blueprints:
  - stackname: 'app-{env:Environment}-build'
    template: 'build.template'
    vars:
      prod-KeyName: MyProdKey
      stage-KeyName: MyStageKey
    parameters:
      KeyPair: '{var:{env:Environment}-KeyName}'
```

While this is perfectly possible this gets very confusing soon. Plus you'll have to mention every variation of the variable explicitely.

Instead you can use a conditional value:

```
blueprints:
  - stackname: 'app-{env:Environment}-build'
    template: 'build.template'
    parameters:
      KeyPair:
        '{env:Environment}==prod': MyProdKey
        '{env:Environment}==stage': MyStageKey
        '{env:Environment}~=/^dev[0-9]+$/': MyDevKey
        'default': MyDevKey
```

StackFormation will evaluate all keys from top to bottom and the first key that evaluates to true will be returned. Allowed conditions: - `A==B` - `A!=B` - `A~=/^regex$/` - 'default' (will always evaluate to true. Make sure you put this at the very end since everything after this will be ignored). Placeholders will be resolved before the conditions are evaluated.

## 2.3 Wildcards

When referencing a stack in `{output:<stack>:<output>}`, `{resource:<stack>:<logicalResource>}`, or `{parameter:<stack>:<logicalResource>}` you can use a wildcard to specify a stack. In this case

StackFormation looks up all live stacks and finds a stack matching the pattern. If there's no stack or more than a single stack matching the pattern StackFormation will throw an exception. This feature is helpful when you know there's always only a single stack of one type that has a placeholder in it's stackname:

Example: Stackname: `deployment-{env:BUILD_NUMBER}` In blueprints.yml:

```
blueprints:
  - stackname: mystack
    parameters:
      Elb: '{output:deployment-*:Elb}'
```

## 2.4 Effective stackname

You can include environment variable in your stackname (which is very handy for automation via Jenkins). In this case your effective stackname (e.g. `build-5`) will be different from the configured stackname (e.g. `build-{env:BUILD_NUMBER}`)

Example

```
blueprints:
  - stackname: 'build-{env:BUILD_NUMBER}'
    template: templates/deploy_build.template
```

## 2.5 Reverse blueprint match

Let's say you have a blueprint `ecom-{env:ACCOUNT}-{env:ENVIRONMENT}-static-stack` and you want to deploy it with ACCOUNT=t and ENVIRONMENT=dpl. You would do this by setting the env vars ACCOUNT and ENVIRONMENT and then run the deploy command:

```
$ export ACCOUNT=t
$ export ENVIRONMENT=dpl
$ vendor/bin/stackformation.php deploy 'ecom-{env:ACCOUNT}-{env:ENVIRONMENT}-static-
↪stack'
```

But instead you can also simply run the deploy command with the resulting stack name `ecom-t-tst-static-stack` StackFormation will then attempt to find a matching tag, determine which environments need to be set and run the original blueprint for you:

```
$ vendor/bin/stackformation.php deploy 'ecom-t-tst-static-stack'

Blueprint reverse match found: ecom-{env:ACCOUNT}-{env:ENVIRONMENT}-static-stack
With ENV vars: ACCOUNT=t; ENVIRONMENT=tst
Use this blueprint and set env vars? [y/N] y
Setting env var: ACCOUNT=t
Setting env var: ENVIRONMENT=tst
...
```

## 2.6 Forcing ENV vars

This will automatically set environment variables in the context of that stack.

```
blueprints:
  - stackname: 'demo'
    env:
      ACCOUNT: t
      ENVIRONMENT: prod
```

CHAPTER 3

# Templates

## 3.1 Template merging

StackFormation allows you to configure more than one template:

```
  blueprints:
    - stackname: iam
      template:
        - iam_role_jenkins.template
        - iam_user_inspector.template
      description: 'IAM users and roles'
```

The template files cannot have duplicate keys in any of the top level attributes. StackFormation will then merge them into a single CloudFormation template and deploy this one instead. This feature helps you to structure your template logically without having to deploy and manage them separatly. Also with this you can choose which template to include in case you're pulling in a StackFormation module like https://github.com/AOEpeople/cfn-lambdahelper.

You can always inspect the final merged and preprocessed template:

```
$ vendor/bin/stackformation.php stack:template iam
```

### 3.1.1 Prefixed template merging

If you list your templates with attributes instead of a plain list, the attribute keys will be used to prefix every element of that template. This way you can use the same template with different input parameters instead of duplicating resources. This comes in handy for VPC setups.

```
blueprints:
  - stackname: vpc-subnets
    template:
      ZoneA: az.template
      ZoneB: az.template
    parameters:
```

```
      ZoneAVpc: MyVPC
      ZoneAPublicSubnetCidrBlock: '10.0.0.0/24'
      ZoneAPrivateSubnetCidrBlock: '10.0.10.0/24'
      ZoneAAZ: 'eu-west-1a'
      ZoneBVpc: MyVPC
      ZoneBAPublicSubnetCidrBlock: '10.0.1.0/24'
      ZoneBPrivateSubnetCidrBlock: '10.0.11.0/24'
      ZoneBAZ: 'eu-west-1b'
      [...]
```

If you have a parameter that needs to be passed to all templates you can prefix it with '*' (make sure you add quotes around that key since JSON will consider this a reference instead) and StackFormation will replace '*' with each prefix used in the `template:` section.

```
  blueprints:
    - stackname: vpc-subnets
      template:
        ZoneA: az.template
        ZoneB: az.template
      parameters:
        '*Vpc': MyVPC # Will automatically be expanded to 'ZoneAVpc: MyVPC' and
↪'ZoneBVpc: MyVPC'
        '*Igw': MyInternetGateway
        ZoneAPublicSubnetCidrBlock: '10.0.0.0/24'
        ZoneAPrivateSubnetCidrBlock: '10.0.10.0/24'
        ZoneAAZ: 'eu-west-1a'
        ZoneBVpc: MyVPC
        ZoneBAPublicSubnetCidrBlock: '10.0.1.0/24'
        ZoneBPrivateSubnetCidrBlock: '10.0.11.0/24'
        ZoneBAZ: 'eu-west-1b'
        [...]
```

## 3.2 Inject Parameters

The scripts (included via `Fn::FileContent`) may contain references to other CloudFormation resources or parameters. Part of the pre-processing is to convert snippets like `{Ref:MagentoWaitConditionHandle}` or `{Ref:AWS::Region}` or `{Fn::GetAtt:[resource,attribute]}` (note the missing quotes!) into correct JSON snippets and embed them into the `Fn::Join` array.

Usage Example:

```
#!/usr/bin/env bash
/usr/local/bin/cfn-signal --exit-code $? '{Ref:WaitConditionHandle}'
```

will be converted to:

```
{"Fn::Join": ["", [
"#!\/usr\/bin\/env bash\n",
"\/usr\/local\/bin\/cfn-signal --exit-code $? '", {"Ref": "WaitConditionHandle"}, "'"
]]}
```

Usage Example:

```
#!/usr/bin/env bash
EIP="{Fn::GetAtt:[NatIp,AllocationId]}"
```

will be converted to:

```
{"Fn::Join": ["", [
"#!\/usr\/bin\/env bash\n",
"EIP=\"",
{
    "Fn::GetAtt": [
        "NatIp",
        "AllocationId"
    ]
},
"\"\n",
]]}
```

## 3.3 Include file content

You can include content from a different file into a script. Use this is you have duplicate code that you need to embed into multiple resource's UserData:

Example:

```
  #!/usr/bin/env bash

  ###INCLUDE:../generic/includes/base.sh
  [...]
```

## 3.4 Inject raw Json

```
###JSON###
{ "hello": "world" }
######
```

## 3.5 Using composer

You can pull in StackFormation modules via composer. Look at the cfn-lambdahelper for an example. A custom composer installer (configured as `require` dependency) will take care of putting all the module files in your `blueprints/` directory. This way you can have project specific and generic modules next to each other.

Please note that a "StackFormation module" will probably not come with a `blueprints.yml` file since this (and especially the stack parameter configuration) is project specific.

You will need to create the stack configuration for the parts you want to use. A good place would be `blueprints/blueprints.yml` where you reference the imported module.

Example:

```
  blueprints:
    - stackname: 'lambdacfnhelpers-stack'
      template: 'cfn-lambdahelper/lambda_cfn_helpers.template'
      Capabilities: CAPABILITY_IAM
```

## 3.6 Comments

You can add comments to your JSON file. Due to a current bug you can't have double quotes in your comment block.

Example:

```
{"IpProtocol": "tcp", "FromPort": "80", "ToPort": "80", "CidrIp": "1.2.3.4/32"}, /*␣
↪Office */
{"IpProtocol": "tcp", "FromPort": "80", "ToPort": "80", "CidrIp": "5.6.7.8/32"}, /*␣
↪Max Musterman HomeOffice */
```

## 3.7 Port

`"Port":"..."` will automatically expanded to `"FromPort": "...", "ToPort": "..."`. So if you're specifying a single port instead of a range of ports you can reduce the redundancy:

Example:

```
{"IpProtocol": "tcp", "Port": "80", "CidrIp": "1.2.3.4/32"},

/* expands to: */
{"IpProtocol": "tcp", "FromPort": "80", "ToPort": "80", "CidrIp": "1.2.3.4/32"},
```

## 3.8 Expand strings with {Ref:...}

Tired of concatenating strings with `{"Fn::Join": ["", [` manually? Just add the references in a string and StackFormation will expand this for you:

Example:

```
"Key": "Name", "Value": "magento-{Ref:Environment}-{Ref:Build}-instance"

/* will be replaced with: */
"Key": "Name", "Value": {"Fn::Join": ["", ["magento-", {"Ref":"Environment"}, "-", {
↪"Ref":"Build"}, "-instance"]]}
```

CHAPTER 4

# Stack references

Referencing outputs/resources/parameters from other stacks

TODO

CHAPTER 5

# User data

## 5.1 Inject user data

TODO

CHAPTER 6

# Stack

## 6.1 Stackname filter

You can configure a regular expression in the `STACKFORMATION_NAME_FILTER` environment variable (e.g. via `.env.default`) which will filter all your stack lists to the stacks matching this pattern. This is useful if you have a naming convention in place and you don't want to see other team's stacks in your list.

Example:

```
STACKFORMATION_NAME_FILTER=/^myproject-(a|b)-/
```

CHAPTER 7

# Stack policies

## 7.1 Using stack policies

To prevent stack resources from being unintentionally updated or deleted during a stack update you can use stack policies. Stack policies apply only during stack updates and should be used only as a fail-safe mechanism to prevent accidental updates to certain stack resources.

It's suggested to create a stack_policies directory below the corresponding stack directory:

```
blueprints/
  stack1/
    stack_policies/
    blueprints.yml
    ...
  stack2/
    stack_policies/
    blueprints.yml
    ...
  ...
```

You have to tell StackFormation where it could find the stack policy.

Example:

```
blueprints:
  - stackname: 'my-stack'
    template: 'templates/my-stack.template'
    stackPolicy: 'stack_policies/my-stack.json'
```

CHAPTER 8

# Functions

## 8.1 Fn::FileContent

Before uploading CloudFormation template to the API there's some pre-processing going on: I've introduced a new function "FileContent" that accepts a path to a file. This file will be read, converted into JSON (using `Fn::Join`). The path is relative to the path of the current CloudFormation template file.

Usage Example:

```
[...]
"UserData": {"Fn::Base64": {"Fn::FileContent":"../scripts/setup.sh"}},
[...]
```

## 8.2 Fn::FileContentTrimLines

These function are similar to `Fn::FileContent` but additional it trim whitespace. This comes in handy when deploying Lambda function where the content can't be larger than 2048kb if you want to directly embed the source code via CloudFormation (instead of deploying a zip file).

## 8.3 Fn::FileContentMinify

These function are similar to `Fn::FileContent` but additional it minify the code. This comes in handy when deploying Lambda function where the content can't be larger than 2048kb if you want to directly embed the source code via CloudFormation (instead of deploying a zip file).

## 8.4 Fn::FileContentUnpretty

This function is the same as `Fn::FileContent` expect it will return the resulting JSON without formatting it, which will reduce the file size significantly due to the missing whitespace in the JSON structure (not inside the file content!) This is useful if you're seeing the "...at 'templateBody' failed to satisfy constraint: Member must have length less than or equal to 51200" error message.

## 8.5 Fn::Split

Sometimes you have a dynamic number of array items. `Fn::Split` allows you to configure them as a single string and transforms them into an array:

```
"Aliases": { "Fn::Split": [",", "www.example.com,cdn.example.com"]}
```

results in:

```
"Aliases": ["www.example.com", "cdn.example.com"]
```

CHAPTER 9

# File paths

## 9.1 Relative file paths

Please note that all files paths in the `template` section of a `blueprints.yml` are relative to the current `blueprints.yml` file and all files included via `Fn::FileContent`/ `Fn:FileContentTrimLines` or `Fn:FileContentMinify` are relative to the CloudFormation template file.

Example:

```
blueprints/
  stack1/
    userdata/
      provisioning.sh
    blueprints.yml
    my.template
```

blueprints.yml:

```
blueprints:
  - stackname: test
    template: my.template
```

my.template

```
{ [...]
  "Ec2Instance": {
    "Type": "AWS::AutoScaling::LaunchConfiguration",
    "Properties": {
      "UserData": {"Fn::Base64": {"Fn::FileContent": "userdata/provisioning.sh"}}
    }
  }
}
```

# CHAPTER 10

# Shell commands

You can run shell commands before or/and after the CloudFormation is being deployed. The commands will be executed in the directory where the blueprints.yml file lives.

## 10.1 Before

Example:

```
blueprints:
  - stackname: 'my-lambda-function'
    template: lambda.template
    Capabilities: CAPABILITY_IAM
    before:
    - cd function
    - npm install aws-sdk
    - zip -r nat_gateway.zip nat_gateway.js node_modules/
    - aws s3 cp nat_gateway.zip s3://mybucket/lambda/nat_gateway.zip
```

and you can even use placeholders:

```
blueprints:
  - stackname: 'my-lambda-function'
    template: lambda.template
    Capabilities: CAPABILITY_IAM
    vars:
      bucket: mybucket
      key: 'lambda/nat_gateway.zip'
    parameters:
      # these are the input parameters passed to the cfn template that match the␣
↪upload location in the custom script below
      S3Bucket: '{var:bucket}'
      S3Key: '{var:key}'
    before:
    - cd function
```

```
      - npm install aws-sdk
      - zip -r nat_gateway.zip nat_gateway.js node_modules/
      - aws s3 cp nat_gateway.zip s3://{var:bucket}/{var:key}
```

## 10.2 After

Similar to `before` scripts you can define scripts that are being executed after the stack has been deployed. Please note this only work if you're 'observing' the deploying (no if you deployed with '–no-observe' or if you're stopping the process (e.g. CTRL+C) during the deployment.

The `after` configuration equals the `before` configuration with the addition that you have access to the status in the `${STATUS}` variable/ (Special status values in addition to the default ones like 'CREATE_COMPLETE',... are 'NO_UPDATES_PERFORMED' and 'STACK_GONE')

Example

```
  blueprints:
    - stackname: 'my-static-website'
      description: 'Static website hosted in S3'
      template: 'website.template'
      after:
        - 'if [[ $STATUS =~ ^(UPDATE|CREATE)_COMPLETE|NO_UPDATES_PERFORMED$ ]] ; then␣
↪aws s3 sync --delete content/ s3://www-tst.aoeplay.net/; fi'
```

## 10.3 Before and after

`before` or `after` are being executed in the base directory of the current blueprint (that's the directory the blueprint's blueprint.yml file is located at). But you can switch directories in your script. The `${CWD}` variable holds the current working directory (the project root) in case you want to switch to that.

When a profile is being used (even if the profile is loaded via the `profiles.yml` file) the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` variables will be set in the script context, so you can safely call the aws cli tool in the same context the blueprint is being deployed.

In addition to that `${BLUEPRINT}` will hold the current blueprint's name and `${STACKNAME}` the current resulting stack name Also `${STATUS}` will hold the last status of the stack that has just been deployed (`after` scripts only).

You can separate the script lines in an array (that will then be concatenated with `\n` before executing:

```
  blueprints:
    - stackname: 'my-static-website'
      [...]
      after:
        - 'echo "Line 1"'
        - 'echo "Line 2"'
```

or you can use the YAML multiline notation:

```
  blueprints:
    - stackname: 'my-static-website'
      [...]
      after: |
        echo "Line 1"
        echo "Line 2"
```

CHAPTER 11

# AWS Sdk

StackFormation uses the AWS SDK for PHP. You should configure your keys in env vars:

```
$ export AWS_ACCESS_KEY_ID=INSERT_YOUR_ACCESS_KEY
$ export AWS_SECRET_ACCESS_KEY=INSERT_YOUR_PRIVATE_KEY
$ export AWS_DEFAULT_REGION=eu-west-1
```

CHAPTER 12

# Misc

Use the `jq` tool to create a simple list of all parameters (almost) ready to paste it in the blueprints.yml

```
$ cat my.template | jq '.Parameters | keys' | sed 's/",/: \'\'/g' | sed 's/"//g'
```

CHAPTER 13

# Contributing

Your contributions are always welcome! Please feel free to fork this repository and submit pull request as many you want! If you have any questions please feel free to contact us.

## 13.1 Contributors

- Fabrizio Branca
- Julian Kleinhans
- Lee Saferite
- Daniel Niedergesäß

**License**

Open Software License v. 3.0 (OSL-3.0)