EMBL-EBI Cloud Portal Documentation

Release 0.1.1

Gianni dalla Torre, Dario Vianello

Jan 25, 2019

Contents:

CHAPTER 1

Using the EMBL-EBI Cloud Portal

The EMBL-EBI Cloud Portal is offered as a service to ELIXIR users as well as users coming from other research communities and institutions.

1.1 How to access the EMBL-EBI Cloud Portal

The EMBL-EBI Cloud Portal is available at the address https://cloud-portal.ebi.ac.uk.

At the moment, users needs to sign-up for an ELIXIR account to access the EMBL-EBI Cloud Portal. You can easily obtain your ELIXIR identity for free at the ELIXIR sign-up page, and go to the EMBL-EBI Cloud Portal login page at https://cloud-portal.ebi.ac.uk/welcome/login.

1.2 Setting up up your Cloud Profile

Your Cloud Profile contains all the information required to deploy your applications to a given cloud provider. It's subdivided in three sections: *Configurations* , *Cloud Credentials*, and *Deployment Parameters*. *Cloud Credentials* are combined with a set of *Deployment Parameters* and a SSH public key to give the *Configuration* required to deploy a certain application in the cloud provider of choice.

You can access your current *Cloud Profile* here (You need to log-in first!)

1.2.1 Cloud Credentials

Interacting with clouds, being them private or public, usually requires authentication. Many of the tools used to interact with cloud APIs rely on environment variables to source authentication tokens, with the clear advantage of not requiring credentials to lie in some file with your filesystem. Terraform, the open-source tools that the EMBL-EBI Cloud Portal uses to provision the required virtual infrastructure, is not an exception to this.

Each cloud provider requires a set of *cloud credentials*. While it's possible somebody has *shared* some credentials with you already via a **Team**, you can find the instructions to insert you personal *Cloud Credentials* below.

Amazon Web Service (AWS)

Add your *Cloud Credentials* like this:

Key	Value
AWS_ACCESS_KEY_ID	YOUR_ACCESS_KEY
AWS_SECRET_ACCESS_KEY	YOUR_SECRED_KEY
AWS_DEFAULT_REGION *(Optional)*	YOUR_AWS_REGION

You can find this information in you AWS user page under the section IAM —> USERS (or via the official docs). AWS_DEFAULT_REGION is *optional* and can be omitted if the region is selected in another way (i.e. via a Terraform variable specificied in the *Deployment Parameters*). A list of the AWS regions can be found here.

Azure

Add your Cloud Credentials like this:

Key	Value
ARM_SUBSCRIPTION_ID	YOUR_SUBSCRIPTION_ID
ARM_CLIENT_ID	YOUR_CLIENT_ID
ARM_CLIENT_SECRET	YOUR_CLIENT_SECRET
ARM_TENANT_ID	YOUR_TENANT_ID
ARM_ENVIRONMENT	public

In order to use Terraform with Azure it is required to create a Service Principal via the Azure portal. The Terraform documentation provide an extensive explanation on how to obtain this in its official documentation.

OpenStack

Add your Cloud Credentials like this:

Key	Value
OS_AUTH_URL	YOUR_AUTH_URL
OS_USERNAME	YOUR_USERNAME
OS_PASSWORD	YOUR_PASSWORD
OS_TENANT_ID	YOUR_TENANT_ID
OS_TENANT_NAME	YOUR_TENANT_NAME
OS_REGION_NAME	YOUR_REGION_NAME

The specific set of values for your OpenStack provider might be slightly different and is contained in the OpenStack RC file, which is project-specific and contains the credentials used by all the OpenStack services.

You can download the OpenStack RC file file from the OpenStack dashboard as an administrative user or any other user.

1. Log in to the OpenStack dashboard.
2. Choose the Project for which you want to download the OpenStack RC file
3. Click Compute —> API Access. (In older version of OpenStack click Compute —>‘‘Access & Security‘‘)
4. Click Download OpenStack RC File and save the file.

Google Cloud Platform (GCP)

Authenticating with Google Cloud services requires the set up of the `GOOGLE_CREDENTIALS` variable, as described in the official GCP documentation.

While locally `GOOGLE_CREDENTIALS` can simply point to a JSON file containing your access keys, in the EMBL-EBI Cloud Portal you're required to upload the JSON string directly.

The JSON file can be downloaded directly from the Google Developers Console following these steps:

1. Log into the Google Developers Console and select a `project`.
2. The API Manager view should be selected, click on `Credentials` on the left, then `Create credentials`, and finally "Service account key".
3. Select `Compute Engine default service account` in the `Service account` drop-down, and select `JSON` as the key type.
4. Clicking `Create` will download your `credentials`.

Once you have your credentials you can add them to `Cloud Credentials` in this form:

Key	Value
`GOOGLE_CREDENTIALS`	`{ "type":"service_account", [..]}`

1.2.2 Deployment parameters

`Deployment parameters` represent a set of inputs specific that are related to the cloud provider and eventually to a specific application. In general, they provide information about the shared instances that you can have in place in your cloud provider or just information that you prefer to set up just the first time and avoid to repeat every time you deploy the instance.

The deployment parameters required by an appliance are expressed in the documentation page of the git repository of the same appliance.

For your convenience you can use a single `deployment parameter` configuration for different appliances: it will make use only of the share inputs ignoring the ones that are not relevant. A deployment parameter can also be used to overwrite any of the variables defined in the `terraform.tfvars` file even when it is not reported as input in the `manifest` file.

1.2.3 Configurations

Configurations represent a way to link a set of *Cloud Credentials* with a set of *Deployment Parameters* and an SSH public key. The use of a configuration simplifies the deployment of the applications, allowing to store and reuse as much configuration as possible.

Specify a new configuration is very easy:

- click on the + button;
- assign a name of your choice;
- choose one of the `Cloud Provider` that you have previously defined;
- choose one of the `Deployment parameters` that you have previously defined;
- (optionally) add a public SSH key.

1.3 Inputs

`Inputs` parameters represent a set of parameters that are likely going to change per deployment and thus cannot determined in advance. The best example of this is the number of nodes a compute cluster you're about to deploy will need to have.

Inputs can also refer to variables defined in *Deployment Parameters*, and thus allow to override them only when required.

1.4 Managing the Registry

1.4.1 How to add an Application to the Registry

Adding a new Application is very simple: you just need to know the URL of the git repository where the Applications is stored. As a test, you can add one of the applications maintained by the TSI team: https://github.com/EMBL-EBI-TSI/cpa-instance

Starting from the EMBL-EBI Cloud Portal Home: - Click Application Repository in the menu on the left-hand side; - Click on the + button; - Enter the `URL` of the git repository; - Click `Add`.

Your new application is now included in your Repository!

1.4.2 Applications compliance

The `EMBL-EBI Cloud Portal` requires the presence of a well-formed Manifest file in the root directory of each git repository containing an Application. This file is is a simple `JSON` file

The `manifest.json` file contains a simple dictinary specifying, for example, the Application name and mainainer along with the supported Cloud Providers. Trying to add an Application repository that does not contain - or contains a malformed manifest file, will result in an error.

CHAPTER 2

Packaging Applications for the EMBL-EBI Cloud Portal

Note: The EMBL-EBI Cloud Portal is still in a very active development phase. We strive to keep this documentation in sync with the EMBL-EBI Cloud Portal developments, but there might be some minor delays.

The EMBL-EBI Cloud Portal has been built to provide an App-Store-like experience when deploying applications to private as well as public clouds, independently from their complexity. Leveraging multiple open source tools (see *The tools*), the Portal can orchestrate complex virtual infrastructures ranging from batch systems to *Pipeline-as-a-service* scenarios, where the limit is only set by the application developer. This approach, called Infrastructure as Code, allows to reproducibily deploy virtual resources in one or multiple cloud providers in an highly automated way. The EMBL-EBI Cloud Portal builds on top of these tools to provide a resilient REST API and a web interface allowing users with little to no knowledge of IT management to easily self-provision the infrastructure they require. The process of wrapping infrastructure and workloads in an App that will be then understood and deployed by the EMBL-EBI Cloud Portal is called *packaging*.

2.1 The tools

While the EMBL-EBI Cloud Portal can easily fall in the category of the **Cloud Orchestrators**, one of its streghts is taking advantage of widely adopted Open Source tools to deploy and configure the infrastructure it is required to manage. Adopting this approach limits the amount of efforts that are required to, for example, stay on top of all the changes in the Cloud APIs that providers expose to access their services, to focus on its own very mission: support IT staff as well as researchers in easily deploy the infrastructure they require for their needs.

As previously mentioned, the EMBL-EBI Cloud Portal takes advantage of two open source tools to orchestrate infrastructure, namely Terraform and Ansible. Let's explore them a little bit further, then!

2.1.1 Terraform

Terraform allows to define the virtual infrastructure an application requires to run in an easily understandable declarative template written in HCL (HashiCorp Configuration Language). VMs, networks, firewalls and storage volumes can easily be defined in a single or multiple files, leaving to Terraform to understand dependencies between all these resources and thus the order in which they must be created. Cloud providers offering are, however, too diverse to allow a single template to be automatically deployed across several of them. Terraform doesn't try to hide these differences, For this reason, the person in charge of packaging applications will need to define a Terraform template for each cloud provider he or she intends to support the application for. However, this usually comes down to a very reasonable mapping exercise between each cloud provider object names. This can be seen as a downside of Terraform but, on the other hand, it allows to take advantage many of the vendor-specific features and services that wouldn't otherwise be possible to access. At the time of writing, Terraform supports the major public cloud providers (AWS, Google Cloud Platform, Microsoft Azure, and many more) as well as OpenStack. There are (as always) other cloud orchestrators that are able to deliver similar functionalities, but they are usually bound to a single platform (i.e. AWS CloudFormation or OpenStack Heat).

The Terraform lifecycle

Terraform is based on a declarative language, which allows you to define the desired layout of the infrastructure you want to provision in the cloud. The state of each Terraform deployment is tracked in what is called a *state* file, which is basically a list of all the resources Terraform has deployed in the previous run. Comparing the state file with the *desired* state defined in the templates, Terraform can compute their *diff* and incrementally change a deployment, i.e. increasing the number of nodes in a cluster, or recover from failures.

Planning

Depending on the initial state being an empty or a partially provisioned environment, the operations Terraform will need to perform will be different. For this reason, the software allows listing all the tasks that will be carried out in the following run, comparing the desired state defined in the template and the state file and defining a *plan* that you can revise. This is obtained simply running `terraform plan` within the folder containing the Terraform template. Keep in mind that if the state file reports that some components are already deployed, Terraform will check if they are still in place and adjust the plan accordingly.

Applying

`terraform apply` is the operation that deploys a Terraform template to a cloud provider. Terraform will read the template and the state file (if any) figuring out which operations must be carried out to reach convergence, and apply them. This process may take a while, depending on the extent of the required changes and their dependencies, but can usually be greatly speeded up increasing the *parallelism*, or the number of objects Terraform will act on at the same time. Once the deployment is complete, Terraform will print out any output defined in the template and exit.

Destroying

After its honourable service, your infrastructure is ready to be torn down or destroyed, following Terraform's nomenclature. Not violating dependencies is an important factor to consider here, as this might cause errors in the destroy process (i.e. removing a subnetwork while instances are still hooked into it). Terraform wraps all this into an easy to use a single command: `terraform destroy`.

2.1.2 Ansible

While Terraform provides some features to configure (or *localise*) VMs after they're launched, this is limited to uploading bash scripts or run bash commands through SSH. Configuring and orchestrating complex deployments usually requires a fully fledged configuration management system. Countless different software are available to solve this problem, each of them having its own strong and weak points. We've eventually chosen Ansible as the configuration management system for the EMBL-EBI Cloud Portal deployments for several reasons:

- configuration is written in YAML, an *easy-to-read* and *easy-to-write* language.
- the learning curve is very gentle, and most bash scripts can be easily mapped (and improved!) in Ansible tasks.
- it doesn't require any agent on the target VMs, only SSH access.

After a set of resources is created by Terraform, Ansible can take over and apply the configuration changes (i.e. install packages, update configuration files and so on).

While Ansible represents our choice in all the deployment situations, it doesn't imply that applications themselves are forced to use this tool. For example, it would be quite easy to to use Ansible to bootstrap a Salt server (or a Puppet master) that is then used by other VMs to configure themselves.

Note: Since version 2.0 Ansible has added several modules to provision virtual infrastructure as Terraform does. However, Terraform still provides clear advantages, such as dependencies resolution, state tracking, and a much wider range of supported clouds. For these reasons, it still represents our preferred choice.

Warning: While there's nothing to stop you from using Ansible to provision the virtual infrastructure required by your Application, doing so will prevent the EMBL-EBI Cloud Portal from tracking resource consumption as this feature relies on inspecting the Terraform state file.

2.1.3 Linking Terraform and Ansible

Terraform outputs the final state of the deployment in a state file. However, Ansible relies on an inventory file to know to IP addresses of the VMs it needs to talk with and their logical grouping. To bridge this gap, the EMBL-EBI Cloud Portal supports terraform-inventory, a small GO app that is able to parse a Terraform state file and output its content as an Ansible inventory.

Of course, developers are not bound to use this method to connect Terraform and Ansible. Solutions such as the Terraform Ansible Provisioner or even custom scripts are viable options, depending on the needs of the App developer.

2.2 The EMBL-EBI Cloud Portal packaging structure

The EMBL-EBI Cloud Portal has been designed to provide as much flexibility as possible when dealing with Apps development. However, some conventions need to be followed while designing your App in order for it to work properly and to take advantage of all the features we provide.

2.2.1 Cloud providers

The Cloud world can be, as the name says, very *cloudy*. However, the EMBL-EBI Cloud Portal needs to be absolutely sure of which cloud provider an application can be deployed to ensure it's providing the right set of configurations to the final user of your App. For this reason, the EMBL-EBI Cloud Portal relies on a homogeneous labelling of Cloud Providers in the Apps definition as well as in the REST API and the web application. You *must* follow this convention:

Cloud Provider	Label
Amazon Web Services	AWS
Google Compute Platform	GCP
Microsoft Azure	AZURE
OpenStack	OSTACK

If the Cloud Provider you want to write an App for isn't listed here, please get in touch with us - we'll be happy to add it to the list!

2.2.2 Where to store your code

First things first, where do you need to store your code?

The code defining an application for the EMBL-EBI Cloud Portal must be tracked within a Git repository publicly clonable over the internet. This is a **fundamental** requirement, as the way the Portal imports applications in your Registry is cloning such repositories.

Adopting Git as our main delivery mechanisms allows us to easily track code changes, keep `dev` and `prod` deployments separated in different branches, and provides a well-established approach for final users to further customise deployments above what initially foreseen by the App developer simply forking the original repository and applying the required changes.

2.2.3 The general structure

Apps, especially those supporting multiple cloud providers, can consist of a reasonable number of lines of code scattered across multiple files and written in several languages. It is thus important to keep some logical order in the codebase to help other users - and yourself in a few months! - understand how your application has been defined and operates. From the EMBL-EBI Cloud Portal perspective, there are a few requirements that must be satisfied when writing your app, and we'll cover those in the next sections.

Separate Cloud Providers

The code used to deploy to each cloud provider - being it Terraform, Ansible or anything else you require - must be stored in a dedicated folder. The names of these folders are currently not subject to any restriction, but we suggest to give them meaningful names (such as those suggested in the *Cloud providers* section above).

Following this convention ensures that the repository will be more easily understood by other developers and help configuration matching.

Separate Terraform and Ansible

As for the Cloud Providers, we suggest keeping separate the Terraform and Ansible codebases as this improves the readability and maintainability of the repository. Also, it allows for some tricks like sharing the same Ansible code among different cloud providers (symlinks are good!) or using git submodules to share code between several deployments.

Manifest file

The manifest file is a file containing a `JSON` dict providing a description of the application parsed by the EMBL-EBI Cloud Portal when loading it. You can find more information on its structure and the mandatory fields in the *The manifest file* section below.

Deployment scripts

When deploying or destroying an Application, the EMBL-EBI Cloud Portal doesn't directly execute Terraform or Ansible, but executes the `deploy.sh` and `destroy.sh` scripts that it expects to find in each folder dealing with a cloud provider deployment. A third script, `state.sh`, is executed after the deployment succeeds to capture a snapshot of the deployed infrastructure. More details on how these scripts should be coded are available in the *Deployment scripts* section.

Auxiliary scripts

There might be situations requiring additional scripts or tools to carry out the deployment successfully. Feel free to add them to a folder within the repo, either in a cloud provider-specific folder if it's needed only by a single cloud provider or in a generic folder in the root of the repository if you need it in all clouds.

The final structure

Putting everything together, here's how a repository hosting a packaged App looks like:

```
├ .gitignore
├ README.md
├ aws
│ ├ ansible -> ../gcp/ansible/
│ ├ deploy.sh
│ ├ destroy.sh
│ ├ state.sh
│ └ terraform
├ gcp
│ ├ ansible
│ ├ deploy.sh
│ ├ destroy.sh
│ ├ state.sh
│ └ terraform
├ manifest.json
└ ostack
  ├ ansible -> ../gcp/ansible/
  ├ deploy.sh
  ├ destroy.sh
  ├ state.sh
  ├ terraform
  └ volume_parser.py
```

As you can see, there's a file `manifest.json` at the root of it, and then folders storing code for each cloud provider. In this particular repo, the Ansible code is shared among the cloud providers via symlinks, but this is not a strict requirement. Being fully honest, there's hardly strict requirements at all in the way the Portal consumes applications!

2.2.4 The manifest file

Each repository defining an application must contain a `JSON` file, called a *manifest* fine, at the root of the repo. This file is parsed by the EMBL-EBI Cloud Portal when adding an application to the Registry to extract things such as application name, version, contact email of the maintainer, and so on. Here's an example of the manifest file defining a `Generic server instance` App supporting both `AWS` and `OSTACK`:

```
{
  "applicationName": "Generic server instance",
  "contactEmail": "somebody@ebi.ac.uk",
  "about": "A base virtual machine instance",
  "version": "0.6",
  "cloudProviders": [
```

(continues on next page)

(continued from previous page)

```
    {
      "cloudProvider": "AWS",
      "path": "aws",
      "inputs": [
        "instance_type"
      ]
    },
    {
      "cloudProvider": "OSTACK",
      "path": "ostack",
      "inputs": [
        "flavor_name"
      ]
    }
  ],
  "deploymentParameters": [
    "network_name",
    "floatingip_pool",
    "subnet_id"
  ],
  "inputs": [
    "disk_image"
  ],
  "outputs": [
    "external_ip"
  ],
  "volumes": [
  ]
}
```

Nothing too difficult, hopefully! The manifest is logically divided in two *parts*: one dealing with the general description of the application, and one dealing with configurations that are specific to a cloud provider. Let's start from the general one first

Cloud provider independent bits

This part of the manifest deals with all the information that is cloud provider *independent*, such as name of the App, maintainer, version, as well as inputs and outputs. While many of the fields are self-explanatory, here's a run down of all of them:

applicationName (Required) The name of the application packaged in the git repo.

As users can have many applications in their Registries, going for a descriptive name is a good approach (`some server` isn't going to get you far!).

contactEmail (Required) The email address of the person (or group) in charge of maintaining the Application and provide support for it. *Mandatory*

about (Required) A one-line description on what the Application does. *Mandatory*

This will be displayed below the title in the App card within the Repository.

version (Required) The current version of the application. This is also displayed in the App card in the Repository.

deploymentParameters (Optional) A list of the Deployment Parameters for this app.

Deployment parameters are all those parameter that *do not* change between deployments, but are *cloud provider* or *tenancy* specific. For example, the name (or id) of the external network in an Openstack cloud depends on the cloud itself, but is always the same when deploying to a given cloud. It thus makes sense to separate these parameters from deployment-dependent parameters (see *inputs* for those) to save the user the hassle to type them every time.

Variables defined here will be injected by the EMBL-EBI Cloud Portal in the deployment environment prepended with the suffix `TF_VAR_` to allow Terraform to use them directly. Values for the `deploymentParameters` variables are sourced at deployment time from the *Deployment parameters* referenced in the *configuration* selected by the user.

inputs (Optional) A list of the inputs required by the Application.

In this particular case the *disk_image* (also called **image name**) to be used when creating the virtual machine. Inputs should preferred over *deploymentParameters* when their value needs to *change* at each deployment. In our case, the base disk will be different each time the user wants to deploy a different OS (CentOS, Ubuntu, BioLinux,. . .) so it makes sense to keep it as input.

Input fields will be shown by the EMBL-EBI Cloud Portal for each of the *inputs* defined in the manifest to to allow users to customise the deployment behaviour. As for the *deploymentParameters*, all the values will be injected as environment variables with the `TF_VAR` prefix.

outputs (Optional) A list of the outputs the Application wants to show to the user.

A very common use case when deploying infrastructure to the cloud is the need to show back to the user some information resulting from the deployment itself, for example the external IP address of the VM that has just been deployed.

The EMBL-EBI Cloud Portal will scan the output of the Terraform state file looking for the strings defined in this `JSON` array, and display the result to the user.

volumes (Optional) A list of the volumes the Application requires to work.

Sometimes, a deployment requires attaching a previously defined volume. For example, some data may be staged in via a GridFTP server on a particular volume, that is then re-attached to an NFS server serving a batch system. The EMBL-EBI Cloud Portal allows to completely separate the volumes lifecycle from the lifecycle of applications. Adding a volume name (i.e. `DATA_DISK_ID`) to volumes automatically displays a drop-down menu listing all the volumes deployed through the EMBL-EBI Cloud Portal on the deployment card. The id of the selected volume (as provided by the cloud provider, not the portal internal id!) is then injected into the deployment process as an environment variable (i.e. `TF_VAR_DATA_DISK_ID` in our example).

Warning: Variables defined in *deploymentParameters*, *inputs* and *volumes* will be injected by the EMBL-EBI Cloud Portal in the deployment environment prepended with the suffix `TF_VAR_` to allow Terraform to use them directly. Keep this in mind when you're using these variables in Ansible!

Defining supported cloud providers

Each App can support one or more cloud providers, and this is defined by the `cloudProviders` list in the *manifest file*. This key is *required* in each manifest, and supported provider should be declared adding a dictionary (or hash table, following the `JSON` nomenclature) to the `cloudProviders` list with the following schema:

```
{
  "cloudProvider": "AWS",
  "path": "aws",
  "inputs": [
    "instance_type"
  ]
}
```

Allowed keys in this dictionary are:

cloudProvider (Required) Specifies which cloud provider the dictionary specifies support for.

This values is used to filter the *configurations* a user can pick when deploying this applications. It's thus *required* to follow the *nomenclature* defined earlier for this filtering to work as expected.

path (Required) Specifies the path to the folder containing the deployment code for the specified cloud provider.

There is no restriction on the name these folders can have, and this is the very reason why this key exists, but for the sake of understandability we warmly suggest to use the string defined in our *nomenclature* for Cloud Providers in lowercase.

inputs (Optional) Specifies cloud provider specific inputs.

These inputs will be only shown when the users decides to deploy the App in this cloud provider. The EMBL-EBI Cloud Portal will merge them with the *generic inputs* and ask the user to provide values during the deployment process.

Note: At the time of writing, the EMBL-EBI Cloud Portal doesn't support cloud provider specific *deploymentParameters*.

Variables precedence

If the same variable is defined both as a *deployment parameter* and as an *input* (both generic or cloud-specific), **inputs** will always take precedence. This allows to override what defined in a *Deployment parameters* on ad-hoc basis. However, this approach is *not* recommended as it obscures the flow of information in your App.

2.2.5 Deployment scripts

At the moment, the EMBL-EBI Cloud Portal doesn't execute Terraform or Ansible directly, but relies on bash scripts to interact with the deployments. These scripts needs to be provided by the App developer and should carry out all the operations required to deploy, check and destroy the application. Bash scripts can easily be seen as an *inelegant* way to deal with this, but it currently provides the best level of flexibility to Apps developers while we more closely observe their needs - a fundamental step to a more organised approach. Some exploratory work is currently in progress to move away from this approach, but this is likely to remain the paradigm the portal will follow in the close future.

Three deployment scripts are required for each cloud provider - deploy.sh, destroy.sh, state.sh - and they must be placed in the folder containing the cloud provider specific codebase (you can have a look at the anatomy of a EMBL-EBI Cloud Portal App *here*).

The deployment environment

On top of the environment variables required by Terraform to authenticate with the cloud providers and the variables defined by *deploymentParameters* and *inputs*, the EMBL-EBI Cloud Portal will inject additional variables in the deployment environment that you can use in your deploy scripts.

There are two main set of variables the EMBL-EBI Cloud Portal injects: deployment variables and ssh management variables.

Deployment variables

Deployment variables are variables that let the App developer know where to access the App repository in the filesystem, place all the output files (i.e. the Terraform state file) and the unique ID that has been assigned to the deployment. A list of these variables with their description is available below:

Environment variable	Value
`PORTAL_APP_REPO_FOLDER`	Path where the application code is stored (a copy of the cloned repo). Only available to deploy.sh and destroy.sh, **not** to state.sh
`PORTAL_DEPLOYMENTS_ROOT`	Path to the folder storing all the deployments.
`PORTAL_DEPLOYMENT_REFERENCE`	The unique ID assigned to the deployment by the EMBL-EBI Cloud Portal by the portal

Why do you need these variables? A very common use-case is to place the Terraform output in the folder belonging to your deployment: this path can be easily obtained joining `PORTAL_DEPLOYMENTS_ROOT` and `PORTAL_DEPLOYMENT_REFERENCE` as follows:

```
"$PORTAL_DEPLOYMENTS_ROOT'/'$PORTAL_DEPLOYMENT_REFERENCE'/terraform.tfstate'"
```

This will ensure that your state file will end up in the right place in the filesystem, enabling the EMBL-EBI Cloud Portal to parse it to obtain usage information.

Endpoint variables

There are *API endpoints*, that ECP exposes, that can be used by/from deployed applications. Following environment variables can be used to build HTTP requests.

Environment variable	Value
`PORTAL_BASE_URL`	Base url of the ECP API portal for HTTP request.
`PORTAL_CALLBACK_SECRET`	Alpha numeric string which is passed with http headers for authentication.

SSH variables

The EMBL-EBI Cloud Portal generates a new SSH keypair at each deployment to mitigate the risk of security issues should a private key be compromised. Also, as part of a *Configuration* or during the deployment process, users can provide a public key that needs to be injected in the VMs to grant them access to the deployed App.

These keys are exposed to the deployment environment via several variables:

Environment variable	Value
`portal_public_key_path`	Path where public deployment key is stored
`TF_VAR_portal_public_key_path`	
`portal_private_key_path`	Path where private deployment key is stored
`TF_VAR_portal_private_key_path`	
`profile_public_key`	String containing the public key provided by the user in the *Configurations* or during the deployment process
`TF_VAR_profile_public_key`	

Note: Keep in mind that `profile_public_key` and `TF_VAR_profile_public_key` contain directly the *key as a string*, while the other variables contain the *path* to the a file containing the keys.

Ideally, the flow of an App deployment when dealing with SSH keys should be

1. Inject the public part of the deployment key (`portal_public_key_path`) in the VM(s) being created. Terraform can easily be used to create a keypair, for example in OpenStack, and then inject that keypair in the VMs.
2. Use the private part of the deployment key (`portal_private_key_path`) to grant Ansible (or the Terraform remote-exec provisioner) access to the VM(s) via SSH and apply the configuration.
3. As part of the configuration, replace the public part of the deployment key with the user-specified public key (`profile_public_key`) in the target VMs.

This workflow allows the EMBL-EBI Cloud Portal to seamlessly configure the deployed infrastructure while ensuring that only the user will have access to it once it is successfully deployed.

Warning: Resist the urge to immediately swap the deployment public key with the user public key at the beginning of the deployment. If you do so, and for some reason the SSH connection drops the EMBL-EBI Cloud Portal will not be able to re-establish the connection, causing the deployment to fail. Ideally, swapping the key should be as close as possible to last step of the deployment.

deploy.sh

This script takes care of deploying the App, and usually consists of at least a Terraform call. Here's a snippet of the deploy.sh for a GridFTP server on GCP:

```
#!/usr/bin/env bash
set -e
# Provisions a GridFTP instance in GCP
# The script assumes that env vars for authentication with GCP are present.
export TF_VAR_name="$(awk -v var="$PORTAL_DEPLOYMENT_REFERENCE" 'BEGIN {print␣
↪tolower(var)}')"

# Launch provisioning of the VM
terraform apply --input=false --state=$PORTAL_DEPLOYMENTS_ROOT'/'$PORTAL_DEPLOYMENT_
↪REFERENCE'/terraform.tfstate' $PORTAL_APP_REPO_FOLDER'/gcp/terraform'

# Start local ssh-agent
eval "$(ssh-agent -s)"
ssh-add $KEY_PATH &> /dev/null

# Get ansible roles
cd gcp/ansible || exit
ansible-galaxy install -r requirements.yml

# Run Ansible
TF_STATE=$PORTAL_DEPLOYMENTS_ROOT'/'$PORTAL_DEPLOYMENT_REFERENCE'/terraform.tfstate'␣
↪ansible-playbook -i /usr/local/bin/terraform-inventory -u centos -b --tags live␣
↪deployment.yml > ansible.log 2>&1

# Kill local ssh-agent
eval "$(ssh-agent -k)
```

As you can see, there are a few additional things going on here rather than two simple Terraform and Ansible calls. Let's have a deeper look!

```
#!/usr/bin/env bash
set -e
# Provisions a GridFTP instance in GCP
# For details about expected inputs and outputs, refer to https://github.com/EMBL-EBI-
↪TSI/gridftp-server
# The script assumes that env vars for authentication with GCP are present.
export TF_VAR_name="$(awk -v var="$PORTAL_DEPLOYMENT_REFERENCE" 'BEGIN {print␣
↪tolower(var)}')"
```

This initial block defines the shebang for the script (`#!/usr/bin/env bash`) and forces the bash script to exit immediately if any command exits with a non-zero status (`set -e`). Then, it exports the `TF_VAR_name` environment variable, which will in turn be used by Terraform to populate its own internal variable `name`. This application uses the `name` variable to assign dynamic names to each resources it creates, for example the name of the VM is defined as

```
name = "${var.name}_server"
```

which ensures there will be no name collisions. Following this approach, each resource will be tagged the same EMBL-EBI Cloud Portal deployment ID.

Next step, let's get those VM(s) deployed!

```
# Launch provisioning of the VM
terraform apply --input=false --state=$PORTAL_DEPLOYMENTS_ROOT'/'$PORTAL_DEPLOYMENT_
↪REFERENCE'/terraform.tfstate' $PORTAL_APP_REPO_FOLDER'/gcp/terraform'
```

This snippet is quite easy: run Terraform to deploy the defined template to in the cloud provider. Since the EMBL-EBI Cloud Portal has already injected the correct environment variables to authenticate with the chosen cloud provider you won't need to specify anything else.

VM(s) are now up, let's configure them!

```
# Start local ssh-agent
eval "$(ssh-agent -s)"
ssh-add $portal_private_key_path &> /dev/null

# Get ansible roles
cd gcp/ansible || exit
ansible-galaxy install -r requirements.yml

# Run Ansible
TF_STATE=$PORTAL_DEPLOYMENTS_ROOT'/'$PORTAL_DEPLOYMENT_REFERENCE'/terraform.tfstate'␣
↪ansible-playbook -i /usr/local/bin/terraform-inventory -u centos -b --tags live␣
↪deployment.yml

# Kill local ssh-agent
eval "$(ssh-agent -k)"
```

This block deals with everything that is required by Ansible to work. When the Portal launches the deployment script, a new ssh-agent is spawned and the SSH key (`portal_private_key_path`) to access the VMs is pre-loaded. Then, ansible-galaxy is used to pull all the requirements for the playbook to run. Next step, invoking Ansible itself. It's not a very plain invocation, though:

- prefixing the command with `TF_STATE=...` tells terraform-inventory where to look for the Terraform state file;
- `-i /usr/local/bin/terraform-inventory` tells Ansible to use terraform-inventory to create the inventory on the flight. Keep in mind that Ansible supports as arguments of the `-i` flag both text files containing an inventory and *executables returning an inventory*;

- `-u centos -b` force Ansible to use the user centos over ssh and to execute commands with `sudo` (b = become).

The last step is to kill the previously spawned `ssh-agent`. Deployment (hopefully) done!

Note: When using Ansible Galaxy to download the required roles keep in mind that only *public* repos will be accessible from the EMBL-EBI Cloud Portal.

destroy.sh

This script is executed by the EMBL-EBI Cloud Portal to destroy an Application. It usually consists of a single Terraform call to destroy the provisioned infrastructure. Here's an example, again from a GridFTP server.

```
#!/usr/bin/env bash
set -e
# Destroys a GridFTP deployment in GCP
# The script assumes that env vars for authentication with GCP are already present.

# Export input variable in the bash environment
export TF_VAR_name="$(awk -v var="$PORTAL_DEPLOYMENT_REFERENCE" 'BEGIN {print␣
↪tolower(var)}')"

# Destroy everything
terraform destroy --force --input=false --state=$PORTAL_DEPLOYMENTS_ROOT'/'$PORTAL_
↪DEPLOYMENT_REFERENCE'/terraform.tfstate' $PORTAL_APP_REPO_FOLDER'/gcp/terraform'
```

Nothing fancy, right?

state.sh

This script is executed by the Portal immediately after the deployment to grab an updated picture of all the deployed resources. It's basically a wrapper around the Terraform state command. Here's the usual example!

```
#!/usr/bin/env bash
set -e
# Get the status of a GridFTP deployment in GCP
# The script assumes that env vars for authentication with GCP are present.

# Query Terraform state file
terraform show $PORTAL_DEPLOYMENTS_ROOT'/'$PORTAL_DEPLOYMENT_REFERENCE'/terraform.
↪tfstate'
```

Note: If the `state.sh` script is not present, or fails, the EMBL-EBI Cloud Portal will report the deployment to be in a `RUNNING_FAILED` state.

2.2.6 Testing locally

While the EMBL-EBI Cloud Portal allows you to see the deployment logs (or, to be more precise, `stdout` and `stderr` of the deployment processes), it might be quicker, at least at the beginning of the packaging process, to test deployments locally.

So, can you reproduce the Portal behaviour locally?

First, you need to install a few dependencies: Terraform, Ansible and terraform-inventory (click on the links to go to their respective "How-to install pages").

Second, you need to replicate the *deployment environment*. As you know by now, the EMBL-EBI Cloud Portal interacts with the deployments setting (or *exporting*) variables in the deployment environment. Reproducing this behaviour in a consistent way is easy thanks to source!

Open a new text file in your preferred text editor, write something similar to:

```
#!/bin/bash
# Define the three special env vars
export PORTAL_DEPLOYMENTS_ROOT="absolute/path/to/repo"
export PORTAL_DEPLOYMENT_REFERENCE="test_deployment"
export PORTAL_APP_REPO_FOLDER="."

# Define the volume id of the volume to be linked to our deployment
export TF_VAR_DATA_DISK_ID="vol-fb65c979"
```

and then run `source filename`. This will inject all the variables defined in the file into the bash environment, mimicking the EMBL-EBI Cloud Portal behaviour and saving you to manually export all the variables one by one.

At the bare minimum, you'll need to export the three *deployment variables* (`PORTAL_DEPLOYMENTS_ROOT`, `PORTAL_DEPLOYMENT_REFERENCE` and `PORTAL_APP_REPO_FOLDER`) plus one variable for each *deploymentParameter* and *input* your Application requires (`TF_VAR_DATA_DISK_ID` in our example above).

Note: The EMBL-EBI Cloud Portal automatically prepends the `TF_VAR_` prefix to all *deploymentParameters* and *inputs*. You will most likely want to do the same in your local script to ensure everything will work as expected when deploying through the EMBL-EBI Cloud Portal

Similarly, you need to source the credentials for the cloud provider you want to interact with. You can find all the details on how to obtain your credentials, as well as the environment variables you need to export, in the *Cloud Credentials* section.

Warning: Environment variables are bound to a given terminal and not persisted between restarts. If you want to use multiple terminals, or you close the terminal you are testing your App with and open a new one, you'll need to *source* the environment variables again.

2.3 The deployment process: an end-to-end overview.

We've explored how an App for the EMBL-EBI Cloud Portal should be packaged, and how the deployment process can be driven via the *deployment environment*. But what are all the steps the EMBL-EBI Cloud Portal takes every time it needs to deploy or destroy an application? How the `deploy.sh` and `destroy.sh` scripts link into that?

Let's lift the hood and have a look at all the operations the EMBL-EBI Cloud Portal carries out after an user has clicked the "Deploy" or "Destroy" buttons!

2.3.1 Deployment

1. After selecting the right *configuration* and provided the required *inputs*, a user clicks on the "Deploy" button and confirms the deployment. The web application of the EMBL-EBI Cloud Portal sends the request to the REST

API and updates the status of the deployment to `STARTING`.

2. A new *deployment environment* is created, injecting *cloud credentials* and all the specified *deployment parameters*, *inputs* and *volumes*.
3. The `deploy.sh` script for the selected cloud provider is executed. The web interface queries the API for near real-time logs to let the user monitor the deployment.
4. The deployment environment monitors the `deploy.sh` script execution and, if it fails (returns a non-zero exit code), updates the backend marking the deployment as `DEPLOYMENT_FAILED` and stops. The web interface regularly polls for updates and, once it detects the failure, offers the user the choice to destroy the deployment.
5. If the `deploy.sh` script completes successfully, the backend executes the `state.sh` script to capture a snapshot of the provisioned infrastructure and looks for the defined *outputs* in the log files. If the `state.sh` script fails (returns with a non-zero exit code) the deployment environment marks the deployment `RUNNING_FAILED` and the user is offered the choice to destroy the deployment. Otherwise, the deployment is marked as `RUNNING` and the web interface will be able to pull the outputs from the REST API.
6. Done!

2.3.2 Destroy

1. The user clicks the "Destroy" button on the deployment card of a deployed App. The web application sends a request to the REST API to terminate the destroy the deployment.
2. The same *deployment environment* that was created to deploy the application is re-created to destroy it. *Cloud credentials*, *deployment parameters*, *inputs* and *volumes* are added to the environment. While *Cloud credentials* needs to be present for obvious reasons (you still need to prove that *you* is *you* to remove your infrastructure!), all the other variables are added for two reasons:
   - support use-cases in which destroying the infrastructure requires information coming from any of the *deployment parameters* or *inputs* variable. Say you want to send some logs back to a specific server and its hostname is stored as a *deployment parameter*
   - avoid issues when Terraform variables normally sourced via environment variables are not declared with a default value. If Terraform is unable to assigned the value of one of its variables in any of the supported ways then it will resolve to ask them interactively, which is of course a scenario the EMBL-EBI Cloud Portal cannot support and will cause to deployment to get stuck. App developer are warmly encouraged to use the `--input=false` option when invoking Terraform which would cause it to immediately fail if a variable cannot be assigned in any way and not ask its value interactively.
3. The cloud-specific `destroy.sh` script is executed and monitored. If it fails (non-zero exit code) then the deployment is marked as `DESTROY_FAILED` and the user will be offered the option of forcing the destroy. If the `destroy.sh` script succeeds, the deployment is marked as `DESTROYED`.

Warning: A deployment that ends in an irreversible `DESTROY_FAILED` state might, depending on the stage at which the error occurs, leave some infrastructure behind. It is **imperative** that users experiencing this issue **independently** verify that all the provisioned infrastructure is correctly removed.

CHAPTER 3

Avoid security credentials on git public repository

AWS accounts, passwords and other sensitive information are a valuable target: outside attackers are continuously scraping GitHub for credentials embedded in the code.Security credential leaks expose sensitive data, resources utilization on your costs, or details of your infrastructure that could lead to sabotage. It is therefore essential to protect developers from releasing potentially harmful secrets on GitHub.

For our users, we strongly endorse the use of tools for automatic detection.

3.1 git-secrets

We found a helpful tool, preventing you from adding secrets to your Git repositories: git-secrets, which allows you to create hooks for your local repositories.

The tool causes a commit fails, for every commit containing (detected) security credentials. It requires configuration for each local repository that you want to protect.

It is possible to integrate its use in a CI system, detecting accidental commits, but this strategy will expose your secret.For this reason, we suggest to use it just as second layer protection (in the unfortunate case a developer forget to protect a local repository).

3.1.1 Installation

The following steps will download and install the latest version of git-secrets.

```
git clone https://github.com/awslabs/git-secrets
cd git-secrets
make install
```

Or, installing with Homebrew (for OS X users).

```
brew install git-secrets
```

3.1.2 Configuration

It is mandatory to install the git hooks for **every repo** that you wish to use with `git secrets --install`.

Here's a quick example of how to ensure a git repository is scanned for secrets on each commit:

```
cd /path/to/repository
git secrets --install
git secrets --register-aws
```

3.1.3 Advanced configuration

First of all, have a look at the official /git-secrets repository.

Add a configuration template if you want to add hooks to all repositories you initialize or clone in the future.

```
git secrets --register-aws --global
```

Add hooks to all your local repositories.

```
git secrets --install ~/.git-templates/git-secrets
git config --global init.templateDir ~/.git-templates/git-secrets
```

Add custom providers to scan for security credentials.

```
git secrets --add-provider -- cat /path/to/secret/file/patterns
```

3.1.4 Before making public a repository

With `git-secrets` is also possible to scan a repository including all revisions:

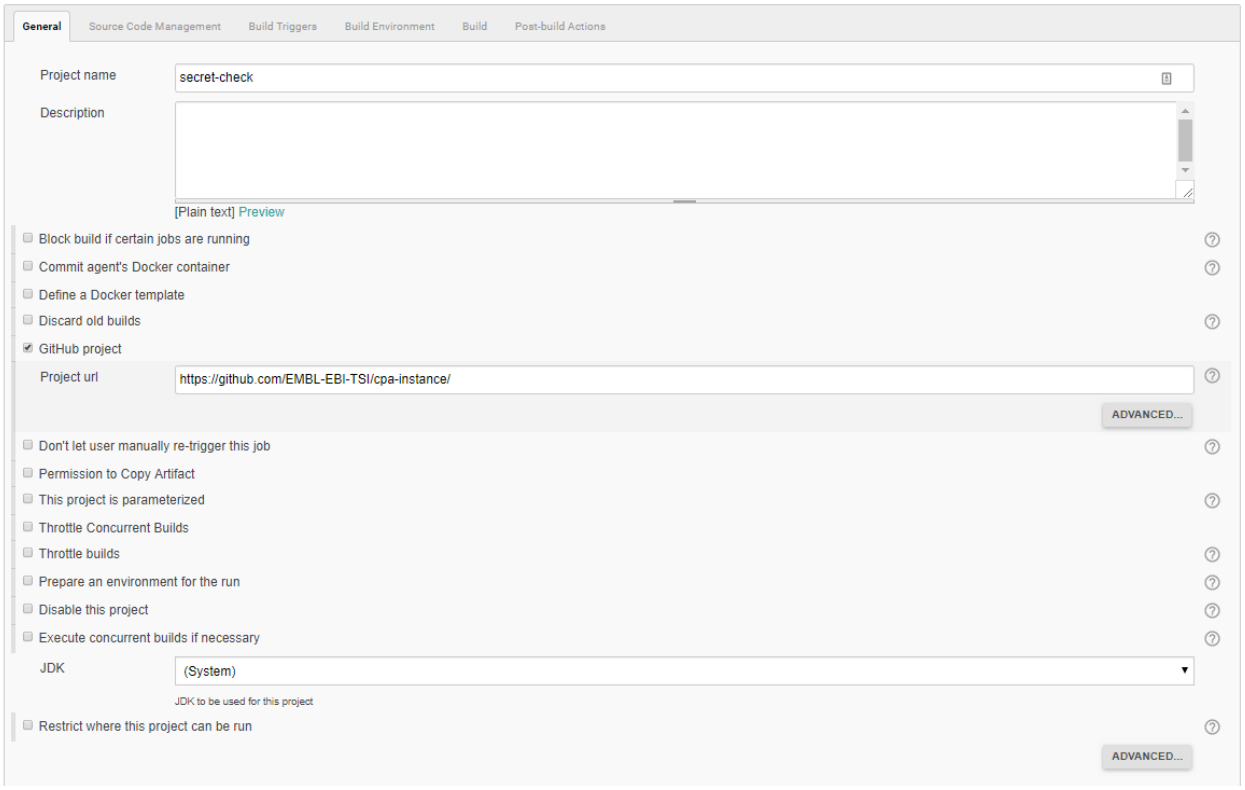
```
git secrets --scan-history
```

3.2 Setting a Jenkins job guarding a repository

We suggest to set up a second layer protection with a CI task for detect accidental commits.

- Define a `GitHub project-Project url`, under `General` i.e.:

```
        https://github.com/EMBL-EBI-TSI/cpa-instance/
```



Project name: secret-check

Description:

[Plain text] Preview

- [ ] Block build if certain jobs are running
- [ ] Commit agent's Docker container
- [ ] Define a Docker template
- [ ] Discard old builds
- [✓] GitHub project

Project url: https://github.com/EMBL-EBI-TSI/cpa-instance/

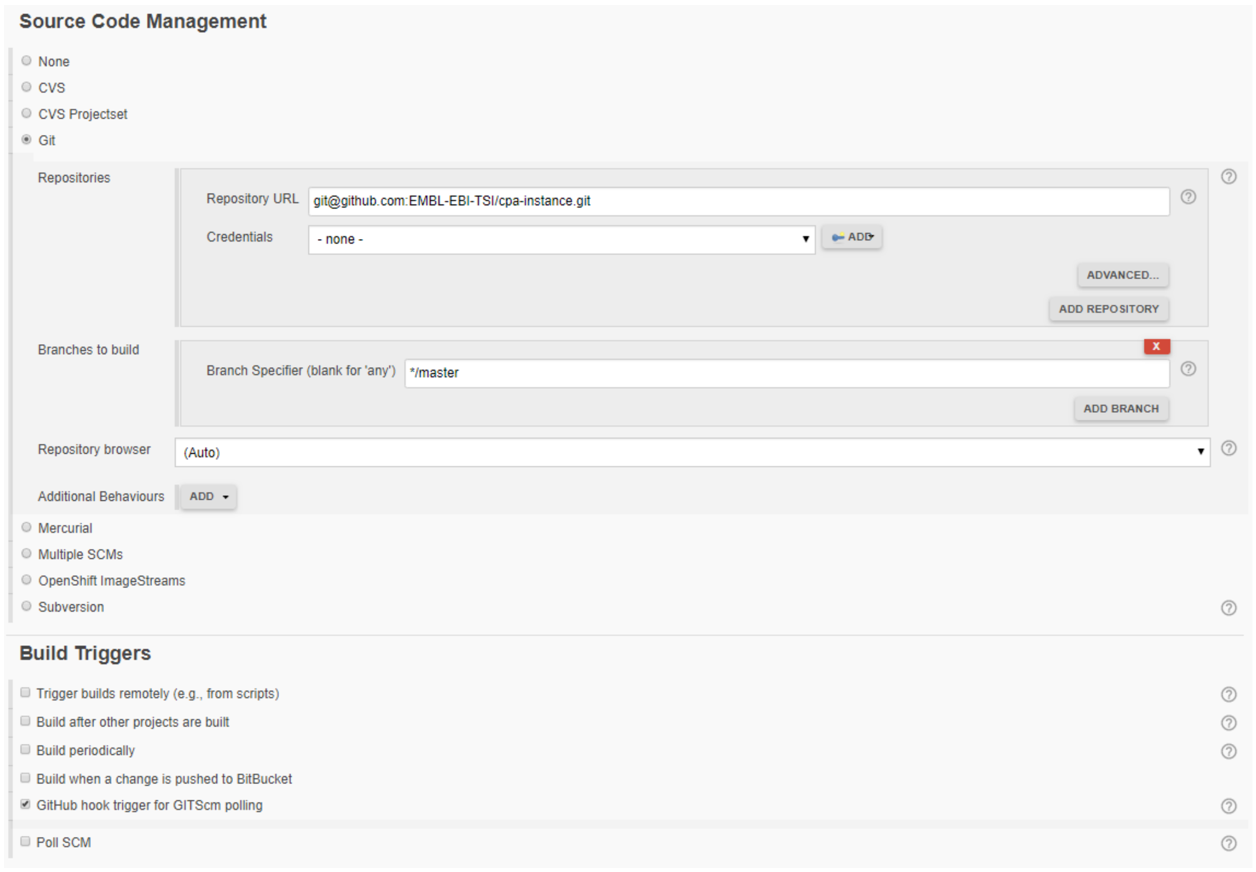
- [ ] Don't let user manually re-trigger this job
- [ ] Permission to Copy Artifact
- [ ] This project is parameterized
- [ ] Throttle Concurrent Builds
- [ ] Throttle builds
- [ ] Prepare an environment for the run
- [ ] Disable this project
- [ ] Execute concurrent builds if necessary

JDK: (System)

JDK to be used for this project

- [ ] Restrict where this project can be run

- Specify the same URL under: `Source Code Management-Git-Repositories-Repository URL`
- Flag `GitHub hook trigger for GITScm polling` under `Build Triggers`.


Source Code Management

- None
- CVS
- CVS Projectset
- Git [✓]

Repositories
Repository URL: git@github.com:EMBL-EBI-TSI/cpa-instance.git
Credentials: - none -

Branches to build
Branch Specifier (blank for 'any'): */master

Repository browser: (Auto)

Additional Behaviours

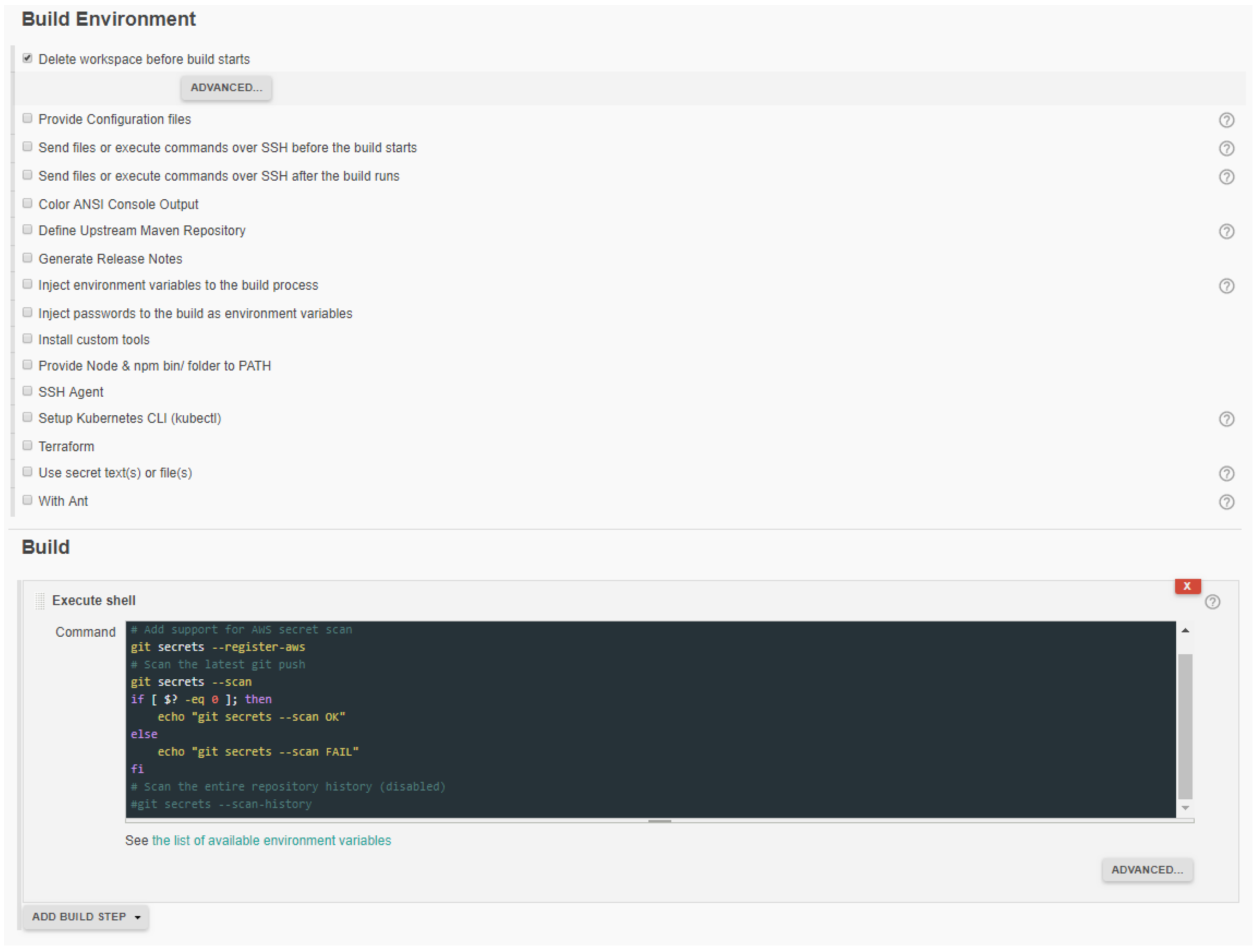
- Mercurial
- Multiple SCMs
- OpenShift ImageStreams
- Subversion

Build Triggers

- [ ] Trigger builds remotely (e.g., from scripts)
- [ ] Build after other projects are built
- [ ] Build periodically
- [ ] Build when a change is pushed to BitBucket
- [x] GitHub hook trigger for GITScm polling
- [ ] Poll SCM

- Flag `Delete workspace before build starts` under `Build Environment`.
- Under `Build-Execute shell-Command`

```
git secrets --install
# Add support for AWS secret scan
git secrets --register-aws
# Scan the latest git push
git secrets --scan
if [ $? -eq 0 ]; then
    echo "git secrets --scan OK"
else
echo "git secrets --scan FAIL"
```



Build Environment

- ☑ Delete workspace before build starts
- ☐ Provide Configuration files
- ☐ Send files or execute commands over SSH before the build starts
- ☐ Send files or execute commands over SSH after the build runs
- ☐ Color ANSI Console Output
- ☐ Define Upstream Maven Repository
- ☐ Generate Release Notes
- ☐ Inject environment variables to the build process
- ☐ Inject passwords to the build as environment variables
- ☐ Install custom tools
- ☐ Provide Node & npm bin/ folder to PATH
- ☐ SSH Agent
- ☐ Setup Kubernetes CLI (kubectl)
- ☐ Terraform
- ☐ Use secret text(s) or file(s)
- ☐ With Ant

Build

Execute shell

Command

```
# Add support for AWS secret scan
git secrets --register-aws
# Scan the latest git push
git secrets --scan
if [ $? -eq 0 ]; then
    echo "git secrets --scan OK"
else
    echo "git secrets --scan FAIL"
fi
# Scan the entire repository history (disabled)
#git secrets --scan-history
```

See the list of available environment variables

For those with accounts on our jenkins server, this Jenkins job https://ci.tsi.ebi.ac.uk/job/app-testing/job/secret-check/ has the latest configuration and can be used as a template.

CHAPTER 4

API Endpoint documentation

Create or Update Deployed Application Outputs A 'PUT' request to create or update application outputs/results.

Example request

```
PUT /deployment/$PORTAL_DEPLOYMENT_REFERENCE/outputs HTTP/1.1
Content-Type: application/json;charset=UTF-8
Deployment-Secret : $PORTAL_CALLBACK_SECRET
Host: $PORTAL_BASE_URL
Body:
[{"outputName":"internal ip","generatedValue":"192.168.3.14"},
{"outputName":"startTime","generatedValue":"2019-01-11 13:45"}]
```

Example response

```
HTTP/1.1 204 No content                                 |
```

Stop Deployed Application A 'PUT' request to stop/destroy deployed application.

Example request

```
PUT /deployment/$PORTAL_DEPLOYMENT_REFERENCE/stopme HTTP/1.1
Deployment-Secret : $PORTAL_CALLBACK_SECRET
Host: $PORTAL_BASE_URL
```

Example response

```
HTTP/1.1 200 OK
```

Note: For more information about environmental variables $PORTAL_DEPLOYMENT_REFERENCE,$PORTAL_BASE_URL, $PORTAL_CALLBACK_SECRET see *Deployment variables*.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search