Syndesis CI documentation

Ioannis Canellos

27/09/2017

Contents

1 introduction

This document intends to be a one stop doc for all things related to Syndesis CI / CD. The documentation starts by describing, the goals and challenges and then move into implementation details. Before all that it makes sense to write some notes about the environment and the syndesis project. . .

1.1 environment

The target environment for both the project and the CI/CD infrastructure is Openshift. In particular for development purposes we have: `https://api.rh-idev.openshift.com:8443`. The Openshift projects / namespace(s) of interest are:

- syndesis-ci
- syndesis-staging

1.2 source code

1.2.1 Project sources

- `https://github.com/syndesisio/syndesis-ui`
- `https://github.com/syndesisio/syndesis-rest`
- `https://github.com/atlasmap/atlasmap`
- `https://github.com/syndesisio/connectors`

1.2.2 CI/CD related sources

- `https://github.com/syndesisio/syndesis-jenkins` (our custom 4.2, with the plugins of our choice and our job defintions & more)
- `https://github.com/syndesisio/syndesis-ci` (the project that contains ci related Openshift manifests. Includes deployment configs, services, secrets & installation scripts)
- `https://github.com/syndesisio/syndesis-pipeline-librarsuv` 2017 1600 ccy (a jenkins pipeline library specific to syndesis)
- `https://github.com/syndesisio/nsswrapper` (a container that initializes the nsswrapper, and use useful in our build pods as init container, more later)

1.2.3 CI/CD related forks

We often hit the limits of the tools we use and we need to trail blaze, that's why we have maintain forks for the following projects:

- `https://github.com/syndesisio/kubernetes-plugin`
- `https://github.com/syndesisio/kubernetes-pipeline-plugin` (we can possibly remove)
- `https://github.com/syndesisio/arquillian-cube` (we can possibly now remove)
- `https://github.com/iocanel/workflow-cps-global-lib-plugin`

2 goals

The overall goal is to have a CI/CD environment that will be able to support our git workflow and will allow us to roll out features to our production environments as fast as possible.

2.1 the git workflow

Currently all development is done on github, and we are using a pull request workflow. Each pull request needs to be validated, reviewed before it can get merged. So our CI/CD environment needs to be able to react on the following events:

- Creation of a pull request.
- Modification of a pull request.
- Approval of a pull request.

2.2 rolling out changes

Each change that gets into the master branch of our individual projects needs to reach our staging environment as soon as possible. So for everything that gets into master, we need to run our system tests and if they pass, we need to roll out:

- Update deployments
- Update images
  - Update image streams tags
  - Update docker.io latest tags

Ideally, all of the above should be done, with the minimal possible configuration in the jenkins side. Using organization scanning, convention over configuration to keep things simple is really desirable.

3 challenges

3.1 security

When working with Openshift the biggest challenges stem from the security related restrictions. The two most common sources of issues are the following:

3.1.1 arbitrary user ids

Containers are not run as root. Instead the run as an arbitrary user named ‘default‘ with a generated uid. The user is not included in the /etc/passwd file, doesn’t have a valid $HOME environment variable and that often leads to issues.

For example:

- The git binary requires the presence of the uid in /etc/passwd.
- Maven doesn’t know how to read user settings.xml from ~/.m2/settings.xml

While there are workarounds for the problem like the above, sometimes they require modification of images or manifests. The suggested approach from Openshift is described here: `https://docs.openshift.com/enterprise/3.2/creating_images/guidelines.html#openshift-enterprise-specific-guidelines`.

3.1.2 permissions

Due to the multi-tenant nature of Openshift, users are more restricted than vanilla Kubernetes. Those restrictions create new challenges:

1. what is the role that allows the user to do X on Y?

   Often we need to allow a user / service account to perform an operation. We can use something like:

   ```
   oc adm policy who-can create rolebindings
   ```

   Even though its easy to find which user or service account do have the right to perform an action, there is no obivious we to find which is the role, that we need to bind to the user in order to gain permission. (If there is one, please update this section. . .)

2. how things work with generated projects?

   A service account may have permissions to create a new and use a new project (by using the self-provisioner role), but modifying an other projects service accounts & permissions can be tricky. For example: Having a service account create a new project and modify its rolebindings, can be tricky (need someone to verify that).

3. tackle differences between users and service accounts

   The example above is something that works fine when you do it as a user from the cli (due to the fact that for rh-idev we are all dedicated-admins), but doesn't work from a service account. This is a constnat source of confusion and it can easily lead to dead-ends.

3.2 pipelines

Pipelines read like groovy and that is both good and bad at the same time. It's good because its easy to understand, its bad because it doesn't execute like groovy and that is a often a source of unpleasant surprises.

3.2.1 mixed execution environments

Each statement (will call them step from now on) inside a groovy pipeline can be possilby executed in three different envionements, dependings on the implemenation and context:

1. in master

   Out of the box every single step is executed inside Jenkins master !!! Even if the build has allocated an agent, the pipeline step execution ALWAYS happens inside the master.

2. in the agent

   A step may or may not, share some of the load, by sending java.lang.Runnable objects for execution to the slave. This is something that needs to be done explicitly and is also a feautre, that's not been used really often (relatively unknown feature).

3. remote host

   Pipeline steps that need to create a process, may choose to do it either locally or remotely, by creating and defining a custom process launcher or a launcher decorator. Some examples:

   ```
   node {
     sh 'echo "I run on the node (either master or agent)"'
   }

   docker.image('alpine').inside {
     sh 'echo "I run inside a docker container"'
   }
   ```

```
kubernetes.image('alpine').inside {
  sh 'echo "I run inside a pood"'
}
```

The examples above demonstrate how the same step means 3 different depending on the context. So this can easily lead to a lot of furtration. For example: if one process requires access to an environment variable, or a volume, you need to have a very good undersanding of where this process will end up being created.

3.3 testing

To validate both our pull requests, merges and roll outs we need to run automated tests, that will test everything as a whole. So the basic requirements are:

1. fully automate the deployment process So we need to have a set of templates that can be easily installed as part of the test preparation. We also need to have a set of parameter values to pass to the templates:
    - github oauth apps
2. polyglot support This needs to be something that can support tests written in any language.
3. ci/cd self validation Doing CI/CD for a microservices architecture is one thing, doing CI/CD for your CI/CD infrastructure is an other.
    - How do you validate changes to your pipeline libraries?
    - How do you vaidate changes to your test suite itself (if its externally hosted)?

4 implementation

This section describes the current implementation. It start's by the documenting the Openshift templates, and then takes items defined in the templates piece by piece...

4.1 syndesis ci project

The syndesis ci project lives at `https://github.com/syndesisio/syndesis-ci` and it contains Openshif templates, configuration files and scripts to install everything from scratch.

Due to the recurring issues with pvc timeouts, we have two flavours of the templates:

- ephemeral
- persistent

Since jenkins feeds the system and end to end tests, with parameters like:

- github oauth client id
- github oauth secert
- github access token

The parameters are also present in the syndesis ci templates.

Along the templates, there are a couple of secrets that need to be created and require additional parameters

- gpg keys (for signing artifacts)
- ssh keys (for accessing github via cli)
- m2 settings.xml (requires sonatype credentials + gpg key name)

A one liner for creating the secrets and applying the configuration is provided by: `https://github.com/syndesisio/syndesis-ci/blob/master/install.sh` The scripts reads sensitive information from password store: `https://www.passwordstore.org/`, but if that is not available it can be easilty tuned to use environment variables.

4.2 jenkins image

Our custom jenkins image lives at: `https://github.com/syndesisio/syndesis-jenkins` and its based on: `https://github.com/openshift/jenkins`.

From the parent image we get for free the following:

- openshift oauth via `https://github.com/openshift/jenkins-openshift-login-plugin`
- a sane default configuration for the kubernetes plugin.
- an image with solved the 3.1.1.

4.2.1 why a custom image?

It's highly unlikely that an existing image, will contain, the exact same plugins as we need. On top of that we are using internal forks for some of the plugin. Last but not least, we need the job definitions to be either provided as secret or baked in to the image and the same applies to initialization groovy scripts.

4.2.2 what plugins do we use?

- The kubernetes-plugin, for providing jenkins agent via kubernetes pod. Lives at: `https://github.com/jenkinsci/kubernetes-plugin`
- The kubernetes-pipeline-plugin, for its arquillian steps (update with internal link). Lives at: `https://github.com/jenkinsci/kubernetes-pipeline-plugin`
- The github-branch-source-plugin. for providing multibracnh project support. Lives at: `https://github.com/jenkinsci/github-branch-source-plugin`
- The gloabl pipeline library plugin, for pipeline library support etc. Lives at: `https://github.com/jenkinsci/workflow-cps-global-lib-plugin`

Plugins like, pipeline, credentials, blue ocean etc are not mentioned in detail, but ARE part of our image. Same applies to transitives.

4.2.3 custom initialization

For sensitive information kubernetes/openshift use secrets and jenkins is using credentials. So we need a way of passing secrets into the jenkins container and have jenkins read them into credentials. For such kind of customizations / initializations a handy approach is to use groovy scripting. And this is what we do:

- we pass secrets as environment variables to the jenkins container.
- we use groovy to convert these environment variables into credentials. The script live at: `https://github.com/syndesisio/syndesis-jenkins/blob/master/configuration/init.groovy.d/setup-github-credentials.groovy`

It worths mentioning that passing secrets as environment variables is not ideal and we should read jenkins credentials from mounted secrets instead of environment variables.

4.2.4 job definition

To define jobs, we have picked `https://github.com/jenkinsci/github-branch-source-plugin`, because:

- It makes it easy to add and configure projects
- supports pull requests

We would like on top of that to have:

- full org scanning
- merge triggers

Since at the moment we don't have full org scanning, the jobs are defined as part of our 4.2. For example: `https://github.com/syndesisio/syndesis-jenkins/blob/master/configuration/jobs/syndesis-rest/config.xml`

Other candidates where:

- `https://github.com/jenkinsci/github-organization-folder-plugin`. Now deprecated by was an excellent example of full org scanning.
- `https://github.com/jenkinsci/ghprb-plugin`. Provides merge triggers but doesn't support pipelines: `https://github.com/jenkinsci/ghprb-plugin/issues/528`. Also seems low on activity.

These job are pipeline jobs and will use out of the box the 'Jenkinsfile' that lives in the root of the source repository.

4.3 pipeline library

For pipelines to be readable, clean and dry its a good idea to use a pipeline library. The pipeline library can hide implementation low level details from the Jenkinsfile and make the Jenkinsfile more comprehensive to everyone. For example:

#+BEGIN$_{SRC}$ groovy slave { withMaven { container('maven') { sh 'mvn clean install' } } } #+END$_{SRC}$ groovy

The pipeline above is pretty clear on what it does, but hides the how, which is encapsulated by the pipeline library itself. That is the added value of a pipeline library.

The pipeline library for the syndesis project lives at: `https://github.com/syndesisio/syndesis-pipeline-library`. The library itself is a small subset of the fabric8 pipeline library, tuned to serve best the syndesis needs and requirements.

4.3.1 why not just use the fabric8 pipeline library?

The fabric8 pipeline library has a wider scope both in functionality and in target platforms. Syndesis CI instead only needs to work great on `https://api.rh-idev.openshift.com:8443`. This means specific features, approaches, workarounds we may employ in the syndesis project, may be irrelevent for fabric8 and vice versa. Trying to tackle different scopes under a single source repository would only slow things down and cause furstration, headaches to both teams. So instead, we keep our own repository, that is being tested by our internal CI infrastructure targeting our environment etc and we contribute features to the fabric8 pipeline library where it makes sense.

Long term, when we will be CI complete and the library will not be as volatile as is right now, we need to align and merge syndesis pipeline library to fabric8.

4.3.2 creating build containers

One of the most important things this pipeline library provides, is functions that allow you to compose build containers.

1. what are build containers?

   Each build has its own requirements. Different builds use different tools or different versions of tools. In the old days, all those tools need to get configured to jenkins itslef and then referenced in the pipelines. Nowadays, an approach popularized by the `https://github.com/jenkinsci/docker-workflow-plugin` is to add tools into containers, and reference containers from pipelines. This removes the extra step and coupling of configuring tools to Jenkins.

   Of course, in openshift access to docker shouldn't be taken for granted. Also going directly to docker means we can't use kubernetes/openshift specific resources like configuration maps or secrets. So for Kubernetes and Openshift such containers are provided by the kubernetes plugin.

2. how does the kubernetes plugin work?

   The kubernetes plugin is mostly responsible for creating agent pods. But these pods may contain additional containers (as sidecars) that can be used as build containers. Then the user is able to select which is the container of the pod that will execute a certain instruction (ONLY sh based pipeline steps are supported).

The structure of the agent pod, can be described by the pod template (as the name implies a template for creating agent pods). The plugin provides pipeline steps for both defining and using these templates. For example:

```
def template = podTemplate(cloud: "openshift", name: "mytemplate", namespace: "syn
       containers: [containerTemplate(name: 'maven', image: "maven", command: '/bi
```

The template above defines a pod template that contains two containers: i) the agent container (jnlp) and ii) the maven container. The first is required, and if ommitted its added by the plugin. Pod templates are hierarchical (e.g. they can extend a parent template). This feautre can be expressed in pipeline using nesting:

```
def composite = podTemplate(cloud: "openshift", name: "mvn", namespace: "syndesis-
    podTemplate(cloud: "openshift", name: "go", namespace: "syndesis-ci", label: '
}
```

The snippet above uses nesting to compose a pod template out of two different ones: i) mvn and ii) go.

As you may have noticed build containers start and do nothing (they execute cat, which causes them to wait forever). This is desired. When we need to usem inside a pipeline, the kubernetes plugin will allow us to exec into them and run the shell command of our choice.

But how do we select in which container we want to execute each shell command? Out of the box all ‘sh‘ instructions are executed by the agent container. If the user needs to change that, then the ‘sh‘ step needs to be wrapped in a ‘container‘ step, for example:

```
container( 'maven' ) {
   sh 'mvn clean install'
}
```

More details on what are the exact features of the plugin can be found in the kubernetes plugin readme.

Below are some of the most important pieces of the pipeline library:

(a) slave

A pod template that defines a template with just the agent pod. From there the pod can be further customized using the nesting concept. The agent container can be customized by defining properties like:

   i. jnlpImage The image for the agent container. Defaults to ‘openshift/jenkins-slave-maven-centos7‘
   ii. serviceAccount The service account to use. Defaults to 0.
   iii. namespace The namespace to use. Defaults to ‘syndesis-ci‘.

(b) inside

A function that is responsible for creating a node out of a pod template. Usually after a pod template is defined, we need to create a node, that has as an argument a label that matches one of the available templates. The inside function handles that for the user:

```
slave {
  podTemplateDecorator {
      inside {
         sh 'echo "do stuff"'
      }
  }
}
```

(c) withMaven

A pod template decorator function that adds a maven container to the pod.

   i. mavenImage The image for the maven container. Defaults to ‘library/maven:3.5.0‘
   ii. envVars A list of container environment variables. Default to ‘[MAVEN$_{\text{OPTS}}$=-Duser.home=$workingDir-Dmaven.repo.local$ ={workingDir}/.m2/r
   The main reason we pass by default these environment variables to the container, is to let the maven know which is the home directory so that the user settings.xml can be found. In the same spirit we also configure the maven local repository.
   iii. workingDir The working directory. Default to ‘/home/jenkins‘.
   iv. mavenRepositoryClaim The maven repository claim to use for the maven local repository. Defaults to “.

v. mavenSettingsXmlSecret The kubernetes secret that contains a valid maven settings.xml. Defaults to ". This should possibly change to m2-settings, which is the secret created by `https://github.com/syndesisio/syndesis-ci/blob/master/install.sh`.

vi. mavenSettingsXmlMountPath The path where the settings.xml will be mounted. Default to '${workingDir}/.m2'.

vii. serviceAccount The service account to use. Defaults to ".

viii. namespace The namespace to use. Defaults to 'syndesis-ci'. In the same spirit there are decorator functions for: golang, yarn, openshift etc.

(d) withArbitraryUser

A pod template decorator, that adds an init container that sets up the nsswrapper. This is usefull for handling the issues caused by 3.1.1, with composition instead of inheritance. The idea is that when nsswrapper is loaded, it can be configured to accept an external passwd and use that instead of /etc/passwd (transparently). The provided passwd may contain a user matching the current uid, allowing us to control the user name, home directory etc and eventually allowing us to change the user under which the containers of the pod will run.

So the init container managed by this template, copies the library to folder accessible by all containers of the pod, and also generates the proper passwd template that will be used by nsswrapper. Last but not least, it decorates all containers with all the required environment variables:

- LDPRELOAD The path of libnsswrapper.so
- NSSWRAPPERPASSWD The location of the provided passwd
- NSSWRAPPERGROUP The location of the provided group file (e.g. /etc/group).

i. image The image to use. An image that provides access to the libnsswrapper.so file. Defaults to 'syndesis/nsswrapper'.

ii. username The username of that will be created for us.

iii. group The group that the user will be added.

iv. description The description of the user.

v. home The generated user home directory.

vi. nssDir This is where the libnsswrapper.so file will be copied to. Default to '/home/jenkins'.

Note: Since the basic idea is to copy a library, this can only work if containers that will use the library, are compatible. For example the default image (the nsswrapper) which is centos based will not work with alpine based containers (see glibc vs musl).

(e) withGpgKeys

A pod template decorator, that imports gpg keys, to ~/.gnupg. This requires the use of 'withArbitraryUser'. The keys can be used for signing artifacts, which is required for releases etc.

i. image The image to use. An image that provides access to the gpg binary is required. Defaults to 'centos:centos7'.
ii. home The home dir to use. Default to '/home/jenkins'.
iii. namespace The namespace to use. Defaults to 'syndesis-ci'.

(f) withSshKeys

A pod template decorator, that imports ssh keys, to ~/.ssh. This requires the use of 'withArbitraryUser'. The keys can for git+ssh on github.

i. image The image to use. An image that provides access to the gpg binary is required. Defaults to 'centos:centos7'.
ii. home The home dir to use. Default to '/home/jenkins'.
iii. namespace The namespace to use. Defaults to 'syndesis-ci'.

3. utility functions

The pipeline library is not just about pod template decorators. It also houses utilities that are reusable accross pipelines:

(a) buildId A function that returns a build id, that can be used to label our build containers. The generated id, is a string that contains information about the build, like the job name and number (as provided by jenkins). This id makes an ideal label for the build containers, as it will end up in the generated pod name, making it easy to correlate builds with pods.
(b) test A function that checks out and builds: `https://github.com/syndesisio/syndesis-system-tests`. This test suite is practically empty. But it does use arquillian cube kubernetes, to ensure that what we build is at least deployable.
(c) rollout This is practically a wrapper around 'openshiftDeploy' provided by `https://jenkins.io/doc/pipeline/steps/openshift-pipeline`.

(d) sonatypeRelease A function that given a maven container can perform a release on sonatype. Returns the release version. This is derived from the git log, after scrapping for entires generated the maven release plugin.

   Needs to run inside a container block that, provides at least:

   - Access to the maven binary
   - ~/.gpg for signing artifacts
   - ~/.ssh for pushing artifacts to github (this also means that cloning using the git url is required).

   The requirements above can easily satisfied with the use of:withMaven, withSshKeys & withGpgKeys.

   Internally this function is doing the following:

   - mvn release:clean release:prepare release:perform
   - mvn org.sonatype.plugins:nexus-staging-maven-plugin:1.6.7:rc-close
   - mvn org.sonatype.plugins:nexus-staging-maven-plugin:1.6.7:rc-release

   The last two steps are using the nexus staging plugin, to automatically close and release the staging repositroy that was generated from the first step.

   WARNING: In order to find the id of the staging repository, we are listing all repositories and grab the first that matches our group id. This needs to be improved as it means that we can have issues with concurrent releases.

   i. branch The release branch. Defaults to ‘master‘.
   ii. nexusServerId The id of the nexus server. This needs to be a server found in the server list inside the mave settings.xml. Nexus credentials will be picked from there. Defaults to ‘oss-sonatype-staging‘.
   iii. profile The maven profile to enable when performing the release. Defaults to ‘fabric8‘.

(e) imagePush

   Pushes an existing image to a docker registry.

   In openshift the lack of direct access to the docker daemon is constant source of pain. So, when we need to push images from within Openshift to an external registry its not trivial. A document describing the process can be found at: `https://blog.openshift.com/pushing-application-images-to-an-external-registry/`.

Based on this document we created this function. The idea is that we create a BuildConfig that uses DockerStrategy to refer to an existing docker image or image stream tag, and an empty dockerfile source. That we use as input. So, we are essentially create a ‘Pass Through‘ build configuration which we use to easilly push image to external registries.

WARNING: This is insanely slow to minishift, to the degrees that is completely unusable.

i. name The name of the buildconfig to generate. Defaults to “ (required).
ii. registry The target registry to push the image to. Defaults to ‘docker.io‘.
iii. user The actual user. Defaults to ‘syndesis‘.
iv. repo The repository. Defaults to “ (required).
v. tag The tag. Default to ‘latest‘.
vi. pushSecret The push secret to use for accessing the external registry.
vii. imageStreamTag. The image stream tag. Defaults “.
viii. dockerImage The dockerImage. This only makes sense if we want to push an actual docker image and no imageStreamTag has been provided. Defaults to “.

4.4 testing

This section focuses on how we test things.

4.4.1 arquillian cube

In our testing strategy arquillian cube is vital, mostly because it maintains the testing namespace for us, and also because it provides helpers tools for deploying configuration, waiting until everything is ready and providing feedback if something goes wrong. The full documentation on cube can be found at: `https://github.com/arquillian/arquillian-cube/blob/master/docs/kubernetes.adoc`

4.4.2 syndesis system tests

This project was initially created for the purpose of verifying pull requests against our various projects. No real tests have been added since. So its purpose is to mostly check that things can be installed (templates and images work). It's maven based and it directly uses arquillian cube. The basic configuration looks like this:

```
<?xml version="1.0"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://jboss.org/schema/arquillian"
   xsi:schemaLocation="http://jboss.org/schema/arquillian
   http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

   <extension qualifier="kubernetes">
     <property name="env.setup.script.url">setup.sh</property>
     <property name="env.teardown.script.url">teardown.sh</property>
     <property name="wait.for.service.list">syndesis-rest syndesis-ui syndesis-keycloak
     <property name="wait.timeout">600000</property>
   </extension>

</arquillian>
```

1. arquillian.xml Here is a break down of all the options that are passed to the arquillian.xml:

   (a) env.setup.url The url or relative path to a script that sets up the test namespace. Due to: `https://github.com/openshift/origin/issues/13197` we extended: arquillian cube so that it can work using shell scripts (to leverage 'oc new-app' and workaround the bug). This is why use the setup/teardown scripts, instead of passing directly a url to the syndesis openshift templates.

   (b) env.teardown.url The url or relative path to a script that tears down the test namespace.

   (c) wait.for.service.list A list of service that we need to wait until a valid endpoint for them has been created (and thus the pod providing it is ready). This option can generally be ommitted when passing a url with manifest we are installing, but is required when using a script instead? Why? Because in the first case, the framework knows of all the bits it installed, while in the later, it doesn't (installation is controlled by the script).

   (d) wait.timeout The amount of time to wait until everything has become ready.

2. setup.sh & teardown.sh

   Intenrally these scripts use environment variables to configure things like:

   (a) $\text{KUBERNETES}_{\text{NAMESPACE}}$ The target namespace.
   (b) $\text{SYNDESIS}_{\text{TEMPLATETYPE}}$ The template type to use (e.g. syndesis, syndesis-restricted, syndesis-ephemeral-restricted).
   (c) $\text{SYNDESIS}_{\text{TEMPLATEURL}}$ The full url to the templates to use.
   (d) $\text{OPENSHIFT}_{\text{TEMPLATEFROMWORKSPACE}}$ Flag to enable reading the template from the ${WORKSPACE} environment variable. This options is used when we want to verify pull requests against the template project itself.
   (e) $\text{OPENSHIFT}_{\text{TEMPLATESFROMGITHUBCOMMIT}}$ The sha of the actual commit to use. Meant to be used for pull request validation (currently unused?).

4.4.3 syndesis end to end tests

A test suite containing end to end tests. Internally it uses protractor.

This suite needs to replace the syndesis system tests in our pipelines, but there are some challenges we need: provide a mechanism for handling a pool github oauth apps for our pipeline.

1. testing non-java project with arquillian-cube and jenkins

   Apparently, we can't directly use arquillian-cube for orchestrating tests written javascript/typescript etc. For this purpose we are using a jenkins plugin, that provides limited access to arquillian cube features: `https://github.com/jenkinsci/kubernetes-pipeline-plugin/tree/master/arquillian-steps`

   Some examples:

   ```
   inNamespace(prefix: 'test') {
       //do stuff inside a namespace
   }
   ```

   The snippet above will create a temporary namespace (as long as the the block is being executed). The generated namespace will have the specified prefix ‘test‘. Alternatively, you can specify instead of a prefix a name and it will use a fixed name instead. If a project with a matching name already exists it will be reused, else it will be created.

Once the project / namespace has been created, the test environment can be created using something like:

```
createEnvironment(
   cloud: 'openshift',
   scriptEnvironmentVariables: ['OPENSHIFT_TEMPLATE_FROM_WORKSPACE': 'true', 'SYNI
   environmentSetupScriptUrl: "https://raw.githubusercontent.com/syndesisio/syndes
   environmentTeardownScriptUrl: "https://raw.githubusercontent.com/syndesisio/syr
   waitForServiceList: ['syndesis-rest', 'syndesis-ui', 'syndesis-keycloak', 'sync
   waitTimeout: 600000L,
   namespaceCleanupEnabled: false,
   namespaceDestroyEnabled: false
 )
```

The above will create an environment using shell scripts (as the our syndesis-system-tests arquillian.xml does), then it will wait until the specified services are 'ready to use'.

WARNING: You always need to be mindful of what you can and cannot do inside a generated namespace. For example checking out a project and building it using ‘mvn package fabric8:build‘ will fail. Why? Because ‘system:serviceAccount:syndesis-ci:jenkins‘ will not have permission to build a project in other namespace even if the namespace has been created by the particular service account. Also, it won't be possible grant that privilege without human interaction. This significantly limits our options and a solution needs to be found. So we need to find a way so that a service account can create a new project and modify its role bindings.

5 our setup

5.1 ci related servers

5.1.1 jenkins

As a CI/CD server we are using Jenkins 2. Since our target environment is Openshift, we scale Jenkins via `https://github.com/jenkinsci/kubernetes-pipeline-plugin`.

For defining build jobs, we are using Jenkins pipeline (which is current the dominant approach for non-trivial build jobs).

At the moment we are using two different Jenkins installations. Why? Mostly due to infrastructure related problems affecting the internal installation and miscommunication.

1. internal jenkins

   `https://jenkins-syndesis-ci.b6ff.rh-idev.openshiftapps.com/` The internal instance is used for pull request validation, and rolling out changes. You can access the instance via Openshift login, which eventually means via Github.

2. fabric8 jenkins

   `https://ci.fabric8.io/` Nowadays, its used for triggering merges, via the [merge] keyword, or via approving a pull request (something that involves `https://github.com/syndesisio/pure-bot`).

5.1.2 nexus

Nexus is used as a mirror of the external repositories that we use. At the moment we use the following configuration inside our settings.xml:

```
<mirrors>
  <mirror>
    <id>nexus</id>
    <name>Nexus Central Mirror</name>
    <url>http://nexus:8081/content/groups/public/</url>
    <mirrorOf>*</mirrorOf>
  </mirror>
</mirrors>
```

Of course, this is something that requires additional work in the nexus configuration.

At the moment we are using the fabric8/nexus image hosted at: `https://github.com/fabric8io/nexus-docker`. Nexus is yet another case that is affected by 3.1.1. For this reason the nsswrapper is baked into the image. Also, this image has baked the configuration inside it. However we override it by mounting a custom nexus.xml at '/sonatype-work/conf'.

6 future work

6.1 full org scanning

Investigate if newer releases of `https://github.com/jenkinsci/github-branch-source-plugin` support full organization scanning.

6.2 merge triggers

Investigate if there is an existing jenkins plugin that can enhance our job definitions, with merge trigggers (merge when a user enters a preconfigured 'phraze' or adds a 'label). Investigate if it worths contributing that pieces of functionality ourselves.

6.3 read jenkins credentials from mounted secrets instead of environment variables

This is pretty self explanatory. It's not the best of practices to pass sensitive information as environmnet variables and we should mount the secrets instead.

6.4 align and merge syndesis pipeline library to fabric8

At the moment the two libraries are similar, but there are a few key differences, that may become a challenge to align.

6.4.1 single vs dual style decration

For each kind of decorators, the fabric8 library is using the 'template' and 'node' variant. The first is the actual decorator and the last is the function the creates a node out of the decrator. In syndesis instead due to the use of composition (it wasn't provided as a feature by kubernetes-plugin when the fabric8 library was first written), we don't have need for the 'node' variant at all (we just compoise with the inside).

So, we need to update the fabric8 pipeline library to the new style, in a fashion that will not break existing consumers of the library.

6.4.2 handle general vs platform specific features

The syndesis library is ATM specific to openshift. We need to generalize it, so that it doesn't break the world when used in vanilla kubernetes (to the degree that possible as not all features wille ever be compatible with vanilla).

The opposite needs to be done in the other side, Make sure that kubernetes specific bits, do not break our existing functionality, for example:

- assuming root user.
- assuming access to docker registry.

6.4.3 make things more configurable/modular

It would be fantastic, if I could use the fabric8 pipeline library with my set of images (e.g. expose the image, service accounts etc as configuration parameters). Also it would be amazing, if I could selectively enable or disable features based on what I have available in my infra (e.g. I don't know if I can use the library without hubot. Need to check)

6.5 find a better way of finding the staging repo id when doing releases

At the moment we list all repositories on nexus, and we grep based on our group id. We end up using the first match (if found). This can cause issues if we have multiple releases going on (given that we have a single groupId for all our projects).

We need to do better than that.

6.6 provide a mechanism for handling a poll of github oauth apps for our pipelines

Each test environment we create, needs to be isolated and only be accessed by the test suite that created it. This means that it needs a unique url. Since each url is referenced by the github oauth apps under `https://github.com/settings/developers` we need to have a unique app for each test suite.

So we need a mechanism that can help us manage the "pool" of github oauth apps and urls.

6.7 find a way so that a service account can create a new project and modify its role bindings.

Currently, a service account with the 'self-provisioner' role is able to create projects. But its not possible to modify role bindings inside the target project. Or at least its not possible to further extend its own privileges within the generated project.

Example: the jenkins service account may create a temporary project X, but it cannot create builds inside X, nor grant itself the builder privilige within X.

We need to find a way to handle it.