Semantic Versioning with Jenkins CI/CD pipeline

# Cloud environments:

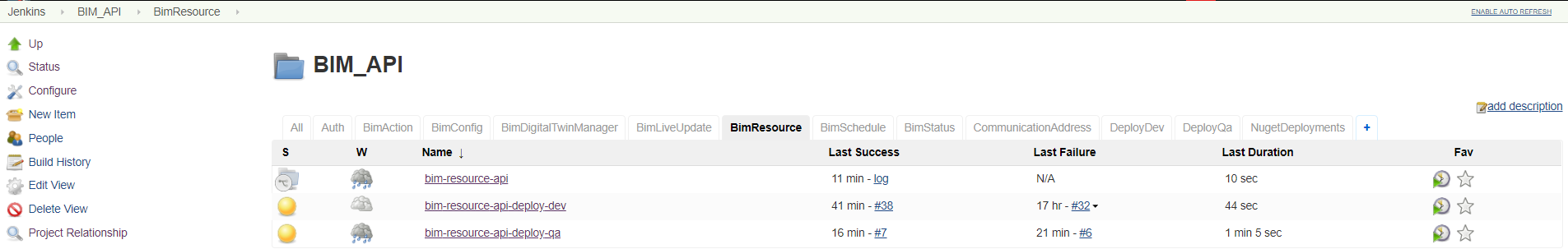
Our cloud has 4 environments: Dev, QA, Staging, and Production. (logical grouping not pysical grouping)

* For Dev and QA environment, the version is 1.0.0-Alpha-BuildID;
* For Staging environment, the version is 1.0.0-Beta-BuildID;
* For Production environment, the version is 1.0.0-BuildID;

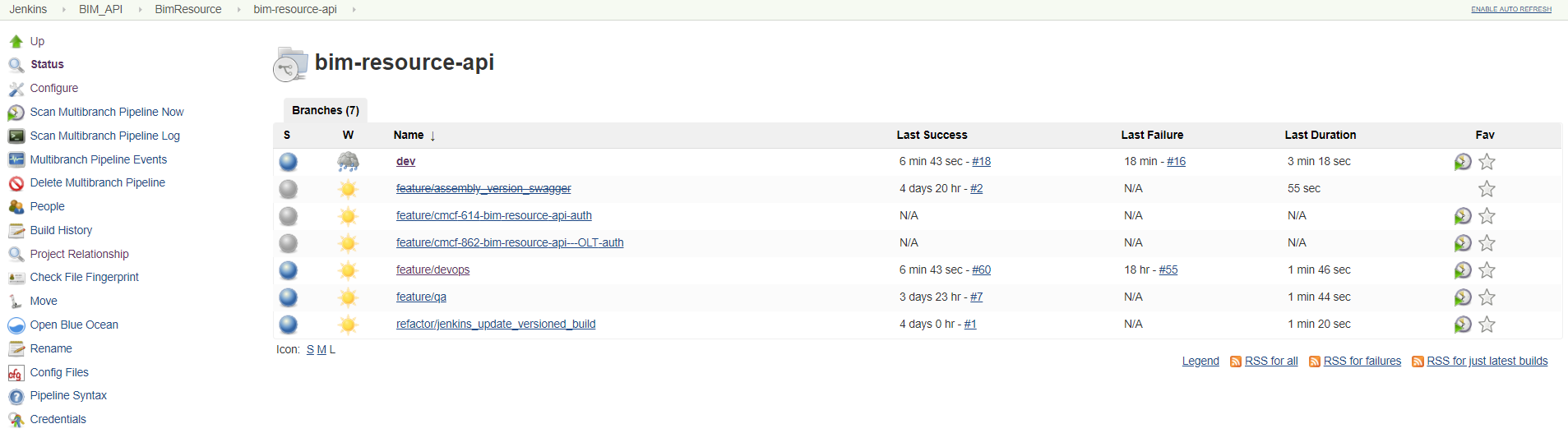
BuildID is the env.Build\_ID (same as env.Build\_Name) from the Jenkins build project “BIM\_API/bim-resource-api/dev”.

# Jenkins projects:

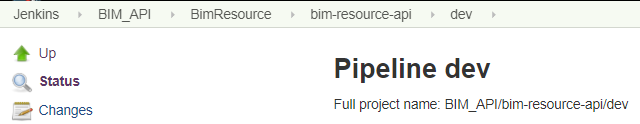
As shown below, the **build** projects are under multi-branches “bim\_resource\_api”, and there are 2 **deploy** projects: “bim\_resource\_api\_deploy\_dev” and “bim\_resource\_api\_deploy\_qa”. At this point of time, we are still under development, so we have not created projects to deploy Staging and Production environment yet.

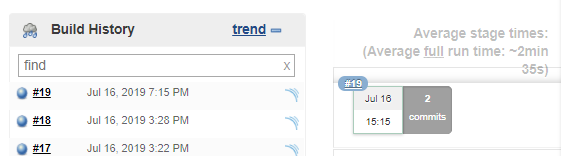
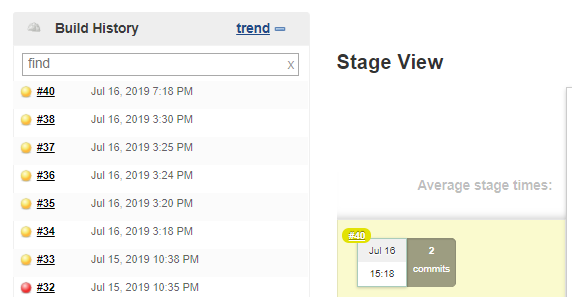


Inside the “BIM\_API/bim-resource-api” multi-branches, each branch has its own build project, whenever you push your code to your branch, Bitbucket will trigger Jenkins to build your branch automatically.



Since only the docker image created from the “dev” branch (we call it dev, which is an equivariant of master branch) will be pushed to Dev environment, so the upstream **build** project is the “BIM\_API/bim-resource-api/dev” project, and downstream **deploy** project is “BIM\_API/bim-resource-api-deploy-dev” project. Both project has their own env.Build\_ID (one is #19 and one is #40 as displayed below), we need to pass the env.Build\_ID #19 from the upstream project to downstream project, so the docker image, swagger page, and the docker image get deployed to Kubernetes all has the same env.Build\_ID, which is the Build Number from the “BIM\_API/bim-resource-api/dev” project.

# Preparation:

1, Find the following files inside the “Version\_Control\bim\_resource\_api” folder:

* Dockerfile (inside “Version\_Control\bim\_resource\_api\src\Osram.Bim.Resource.Api\Osram.Bim.Resource.Api”)
* build.yaml (stores parameters which are used by “Jenkinsfile” and “Deploy.Jenkinsfile”)
* Jenkinsfile (used by the **build** project “BIM\_API/bim-resource-api/dev”)
* Deploy.Jenkinsfile (used by the **deploy** project “BIM\_API/bim-resource-api/dev”)
* Chart.yaml\* (inside “Version\_Control\bim\_resource\_api\terraform\_app\app-chart”)
* Values.yaml\* (inside “Version\_Control\bim\_resource\_api\terraform\_app\app-chart”)

\* “Chart.yaml” and “Values.yaml” files are Helm files. Helm is the package mager for Kubernetes. We use helm as a package manager to install/deploy applications. Helm keeps a record of what's being deployed to Kubernetes cluster. When implement the version control, I didn’t modify these 2 files, but “helm upgrade” and “helm rollback” commands are used inside the “Deploy.Jenkinsfile”.

2, Install “VS Code” with “JenkinsFile Support” plugin as the IDE to view and modify the “Jenkinsfile” and “Deploy.Jenkinsfile”.

3, Go through the materials inside the “CICD\_Workshop” folder, and at the same time compare to the above listed “Jenkinsfile” and “Deploy.Jenkinsfile”. The workshop 1 materials is related to the “Jenkinsfile”, and workshop 2 materials is related to the “Deploy.Jenkinsfile”.

* workshop 1 and “Jenkinsfile” is about the BUILD part of CICD: compile the newly commited code, run unit tests, run code quality check with SonarQube (the biggest problem for SonarQube is if no human check the results then it is useless), then create the docker image via “docker build” then “docker push” it to ACR (AWS equivalent is ECR).
* Workshop 2 and “Deploy.Jenkinsfile” is about using “helm upgrade” as a package manager to install/deploy applications to Kubernetes clusters. A dry run (or a practice run) is a testing process where the effects of a possible failure are intentionally mitigated. QA tests are executed by “Deploy.Jenkinsfile”: when deploy to Dev enviroment, only execute the sanity scripts; when deploy to QA enviroment, both sanity and regression scripts are executed. Currently, since QA tests contains expected “NOT IMPLEMENTED” failures, so if QA test failes, the deploy status in Jenkins will be “unstable” instead of “failure”, this is why I only implmented “helm rollback” in “post – failure” section, but not in “post – unstable” section.

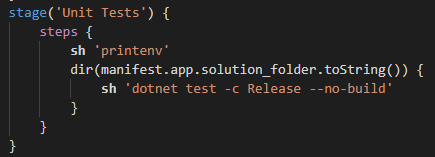
4, Go through the “Guided Tour” section of [Jenkins Documentation](https://jenkins.io/doc/) to understand “Declarative Pipeline” and “Groovy”. In our “Jenkinsfile” and “Deploy.Jenkinsfile”, we structured pipeline-stage-step via “Declarative Pipeline”, and the code inside “script {}” is “Groovy”, also “Shell script” is used in these 2 files via “sh”.

5, Go through the [Youtube tutorial](https://www.youtube.com/playlist?list=PLzvRQMJ9HDiSaisKr7OnM4Fl7JXCDDcmt) to get familiar of the Jenkins GUI side, and how Jenkins is integrated with Bitbucket and Slack.

# Variables:

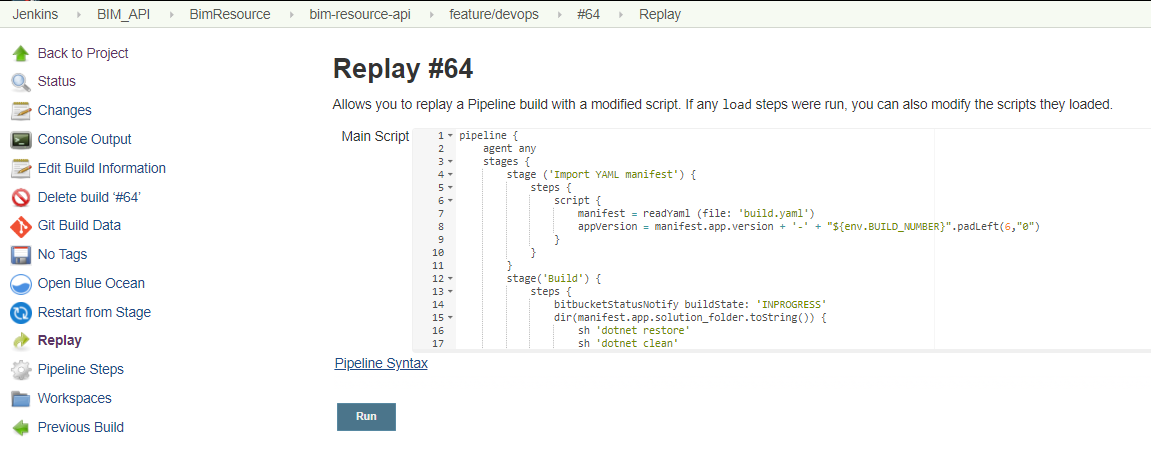
In our In our “Jenkinsfile” and “Deploy.Jenkinsfile”, besides the variables defined inside the “build.yaml” file, there are 4 sets of parameters are defined or can be displayed inside Jenkins GUI:

* Jenkins – Credentials – in Name column: encelium-devops-acr – click the arrow, select “update”: ${AZURE\_CLIENT\_ID}, ${AZURE\_CLIENT\_SECRET}, ${AZURE\_TENANT\_ID}
* Jenkins – Manage Jenkins – Configure System - Global properties: CONSUL\_URL, DOCKER\_REPO\_URL, OSRAM\_NUGET\_URL, SERVICES\_ACR\_PREFIX, UI\_ACR\_PREFIX
* Jenkins – BIM\_API – select any **deploy** project, such as “bim-resource-api-deploy-dev” and “bim-resource-api-deploy-qa” – Configure – Under “This project is parameterized”: ${ENVIRONMENT}, ${NAMESPACE}, ${env.Input\_Build\_Number} (Input\_Build\_Number is added by me to take the passed in parameter from upstream **build** project). This requires that “Parameterized Trigger Plugin” is installed in Jenkins. Note the “POST Build Action” section often mentioned by “Parameterized Trigger Plugin” use cases is not visible for our Jenkins project, because we didn’t use “Freestyle project” but “Pipeline”. I stored the “Input\_Build\_Number” in the “post – success” section of the “Jenkinsfile”, which is equivalent as store it in the upstream **build** project’s “POST Build Action” section from Jenkins’ GUI side.
* If you want to know any of the environment variable’s value, such as ${env.BUILD\_NUMBER}, you can simply add a line of code sh ‘printenv’ inside any “stage – steps” block like below, then use the debug method mentioned in next section to check the value.



# Debug

If you only want to change “Jenkinsfile” or “Deploy.Jenkinsfile”, then there is a convenient way to debug within Jenkins GUI without commit your code to bitbucket. You can debug “Deploy.Jenkinsfile” within a **deploy** project, or debug “Jenkinsfile” within a **build** project, can be the “dev” branch project, or your own branch project, such as my “feature/devops” branch shown below. From your Jenkins project, choose a build # to open, click the “Replay” button on the left-hand side to open your Jenkins script to modify then Run. After execution, you can select the newly created build #, then click “Console Output” button to check the results. Normally detail of failures are listed at the very bottom of “Console Output”.



# Requirements:

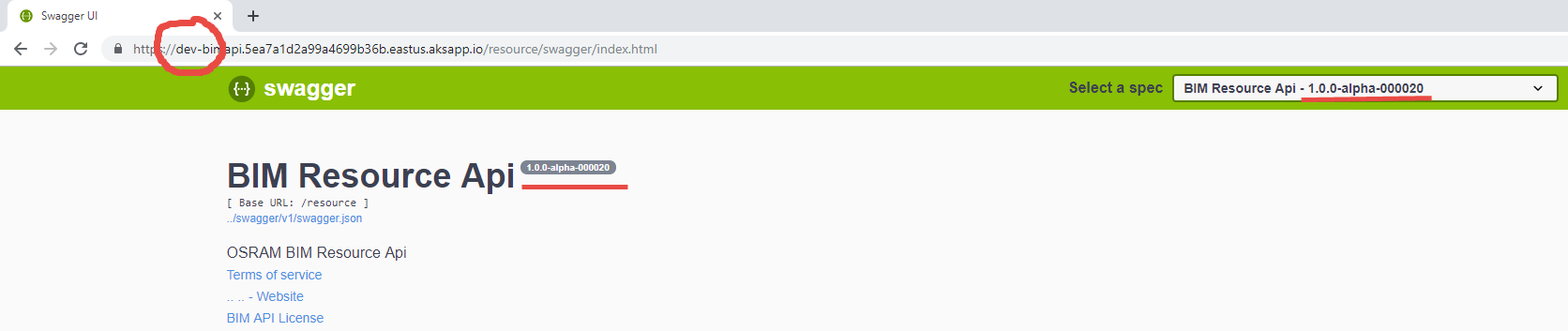
Before I started the implementation of versioning, we already have basic CICD setup with Jenkins, Bitbucket, ACR, Swagger, and Slack without versioning and roll back. The requirements of implementing semantic versioning with continues deployment contains the following things:

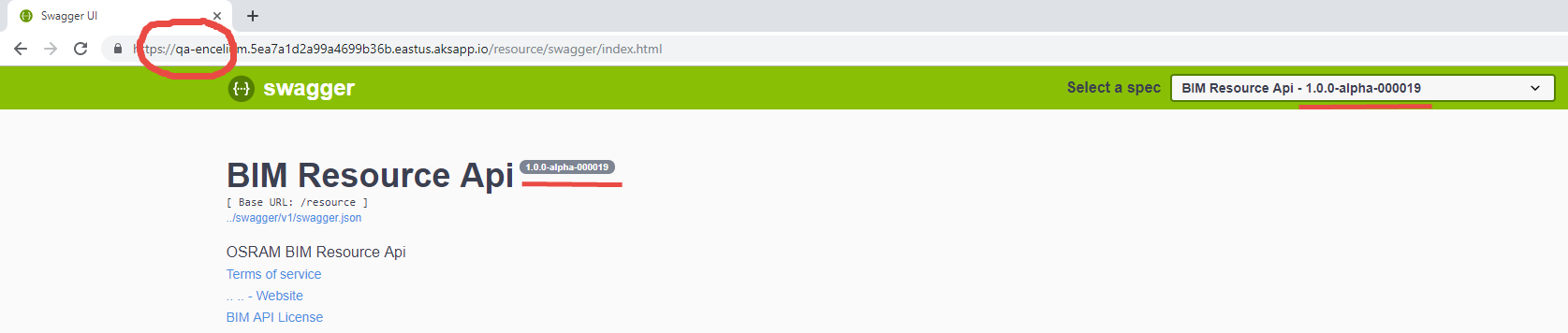
* add version to swagger page
* add version to docker image (dev branch docker image only)
* pick the docker image with right version to automatically deploy to Dev environment
* pick the docker image with right version to manually deploy to QA environment
* able to roll back to last release if the deployment failed
* modify slack deployment alert to contain version information
* implement above for all the APIs (this is a side-effect when you use Microservices…)

# Implementation:

## add version to swagger page

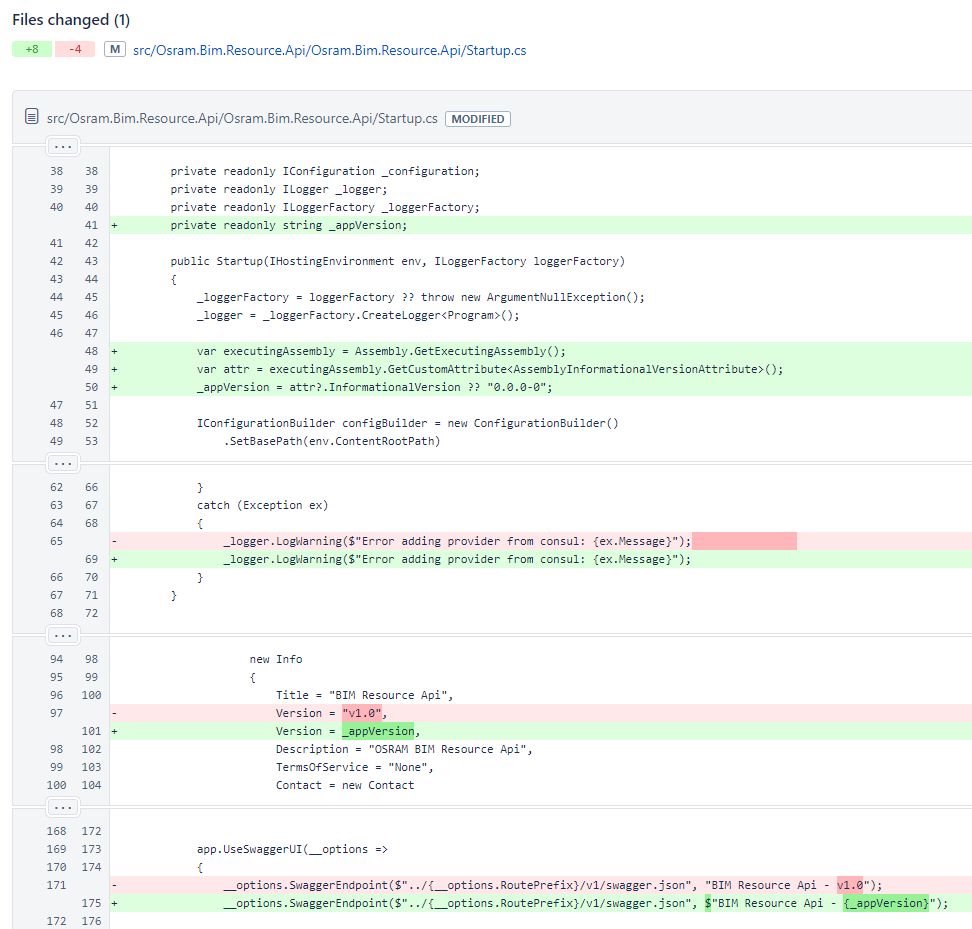
Able to add versioning to swagger page, so Dev environment users and QA environment users knows which version they are using. We automatically deploy to Dev environment every time after the code is merged to dev branch (i.e., master branch) and build successfully in Jenkins, but we only deploy to QA environment manually when the code in dev environment has been tested and stabilized. Dev user and QA user may use different versions, it will be beneficial if they can tell from the swagger page which version they are using and how many versions difference in between Dev and QA environments. Below is the result after my implementation, top is Dev environment, and bottom is QA environment.



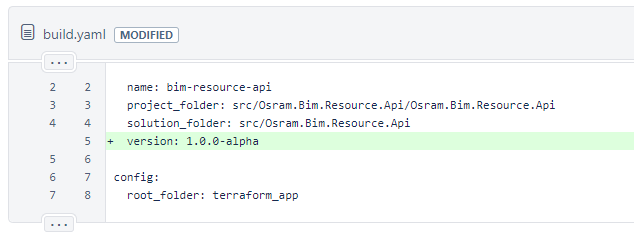


We don’t use swagger paid version. We use [Swashbuckle](https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-2.2&tabs=visual-studio) to generate swagger documentation from our code. Compare to the paid version, the only disadvantage of this approach is collaborators can no longer comment on the swagger page.

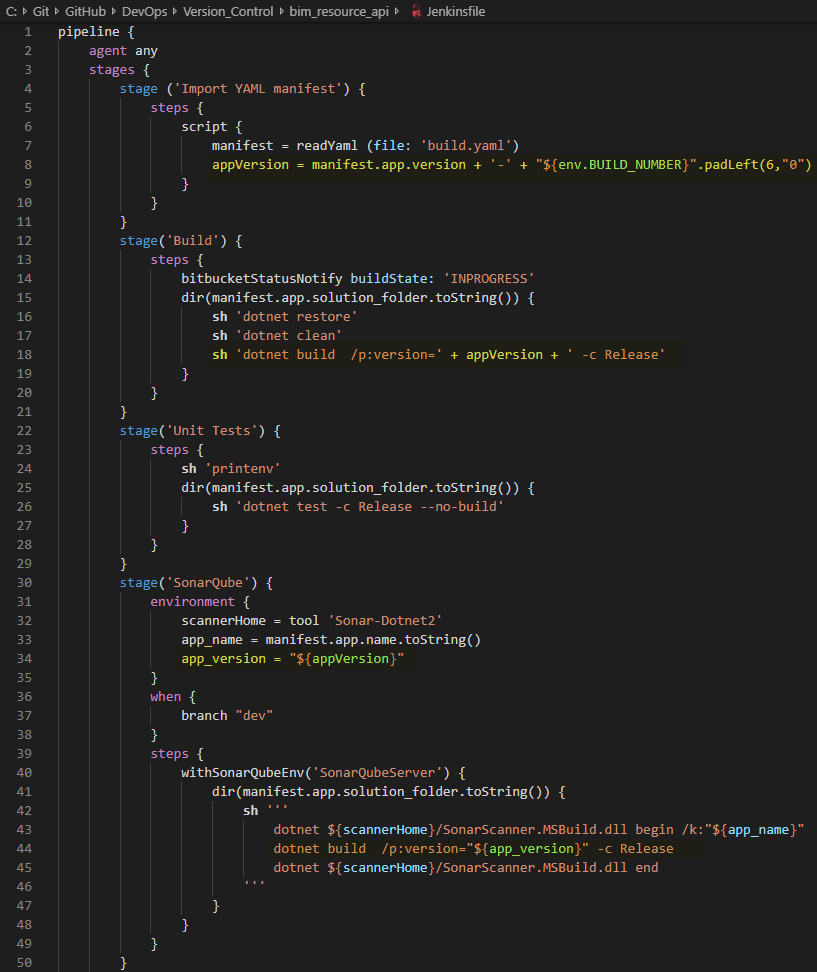
In order to add version generated dynamically by Jenkins build # to swagger page, I need to do 2 things: Ask a developer to modify from the source code side, then I can change the “Jenkinsfile” accordingly. The developer only modified the “Startup.cs” file located at “Version\_Control\bim\_resource\_api\src\Osram.Bim.Resource.Api\Osram.Bim.Resource.Api”.



Before our swagger pages displays a static “v1.0”, as you can see above, it has been replaced by “\_appVersion”, which will be passed in via “Jenkinsfile” with property version /p:version as seen in shell command: dotnet build /p:version="${app\_version}" -c Release. The version will looks like “1.0.0-Alpha-BuildID” where BuildID is the build # from a **build** Jenkins project. First I need to add the static part of the version inside the “build.yaml” file:

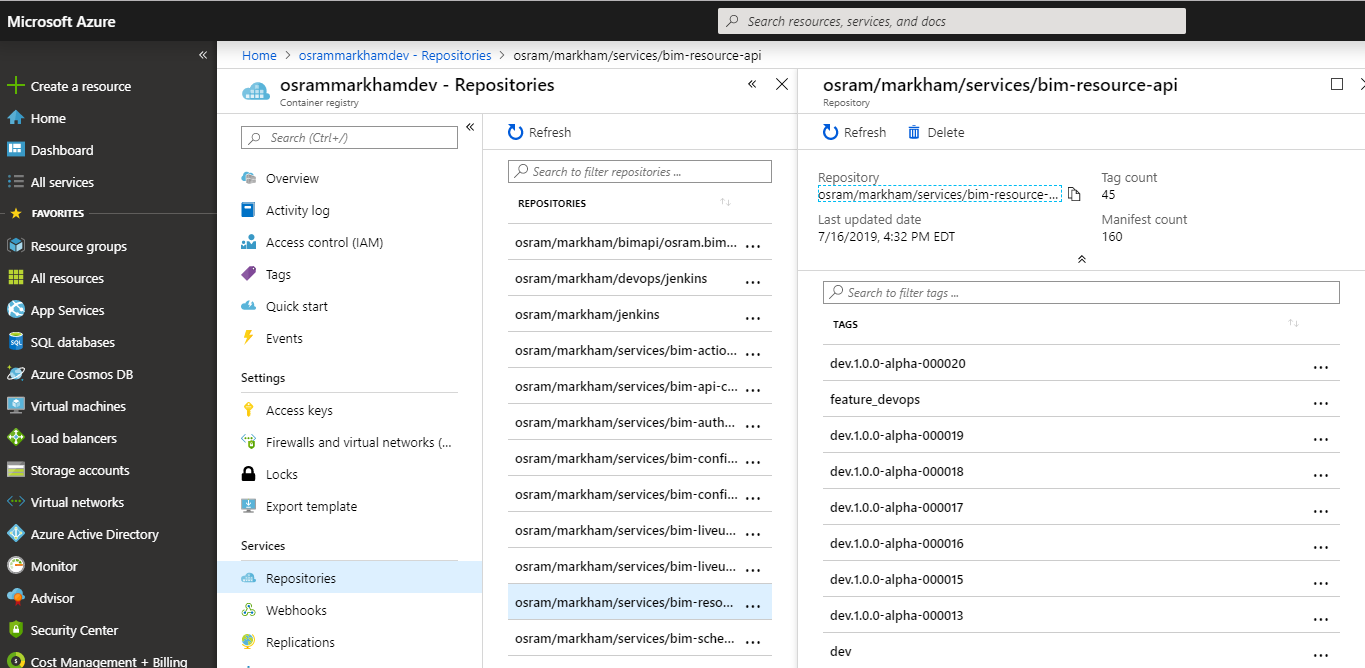


Then I need to modify the “Jenkinsfile” as displayed below by adding line 8, 34, and modify line 18, and 44. The build # from a **build** Jenkins project is passed in via "${env.BUILD\_NUMBER}". Since we are “dotnet build” twice, once in the ‘Build’ stage, and once in the ‘SonarQube’ stage, so both places I need to add /p:version (before both places were dotnet build -c Release). The formatting of string is a bit strange for shell scripts, which deserves a bit debugging by using the ‘Reply’ button mentioned in the “Debug” section.

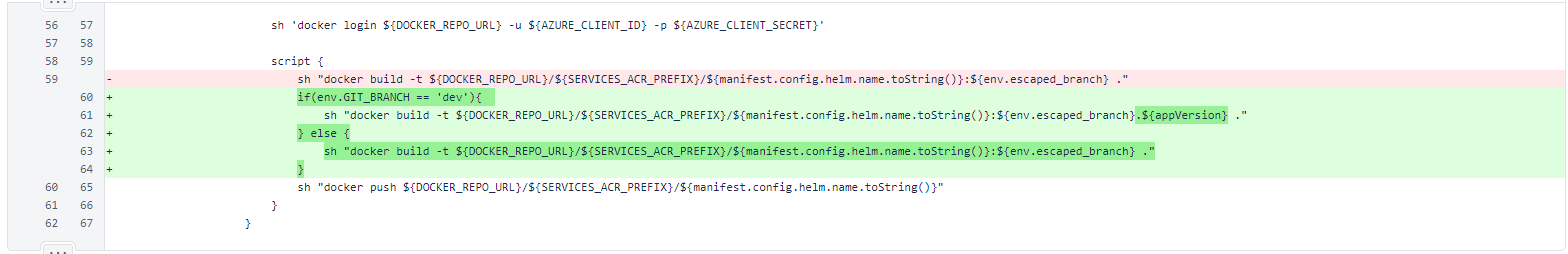


## add version to docker image (dev branch docker image only)

Able to tag the dev Docker image with version. Multi-branches “BIM\_API/bim-resource-api” contains all the branches in Bitbucket, and each branch has its own project in Jenkins, such as “BIM\_API/bim-resource-api/feature%2Fdevops” and “BIM\_API/bim-resource-api/dev”. Each branch’s **build** project will execute the same “Jenkinsfile”, and during the ‘Docker Build’ stage, a docker image will be created via docker build command, then the docker image will be pushed to Azure Container Registry (ACR) via docker push command.

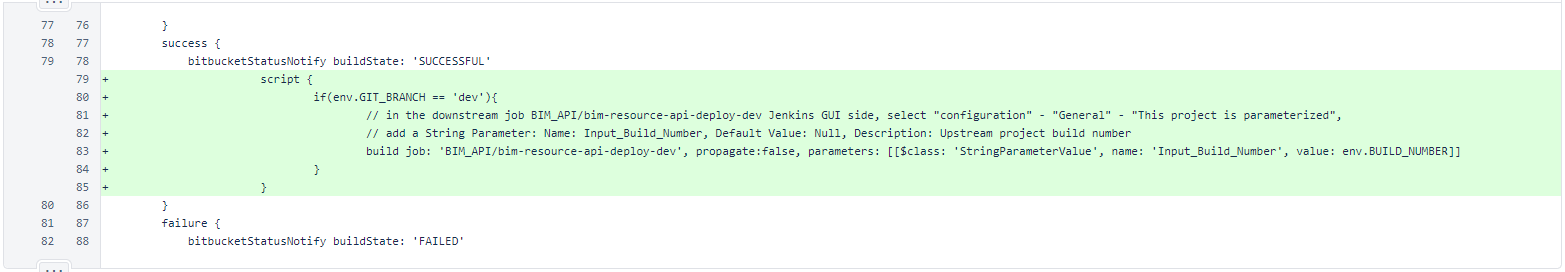


ACR is displayed above. Before the versioning implementation, the docker images are named/tagged with branch name only, such as “dev”, and “feature\_devops”. My job is to use docker build -t (-t means tag) command to tag the dev docker image with version. The reason to only tag dev docker image but not the other branches’ docker images is because we will only deploy dev docker container to Kubernetes cluster. The change of the “Jenkinsfile” is as simple as displayed below, and the effect in ACR is displayed above: “dev.1.0.0-alpha-000020”.



## pick the docker image with right version to automatically deploy to Dev environment

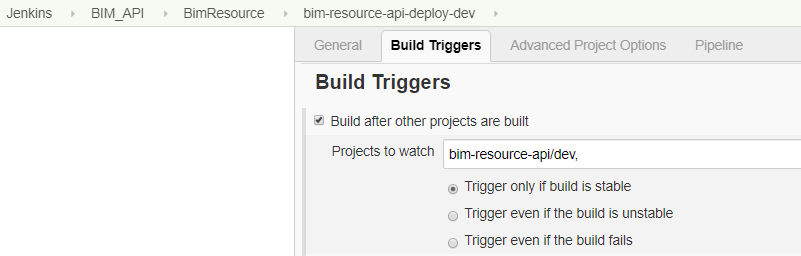
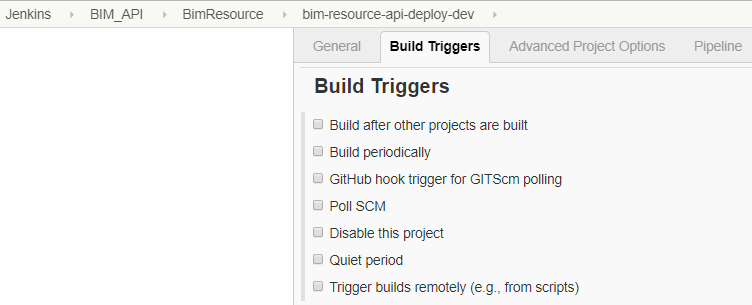
The docker image was built by the **build** Jenkins project, but it will be deployed to Kubernetes by the **deploy** Jenkins project. They are 2 separated projects in Jenkins, which have different build numbers. To identify the right docker image from ACR to deploy, we need to be able to pass the build # from the **build** upstream project to the **deploy** downstream project. To achieve this, you need to install the [Parameterized Trigger Plugin](https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Trigger+Plugin) in Jenkins. In most [Parameterized Trigger Plugin](https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Trigger+Plugin) tutorial or documentation, it will ask you to put the parameter in the “Post-build Actions” section of the upstream project from Jenkins GUI side. I couldn’t find this section, and our DevOps told me that this is because we didn’t use Freestyle project in Jenkins, but we used Pipeline which don’t support this GUI unit. However, the good news is, the “Jenkinsfile” is an equivalent of the Jenkins GUI. I can store the parameter in the “post- success” section via Groovy script.



You need to put your downstream job’s name inside “build job”. “propagate” needs to be placed outside the “parameters” section to take effect. If your downstream job fails or unstable, and you didn’t set “propagate: false”, then your upstream job will be marked as fail too.

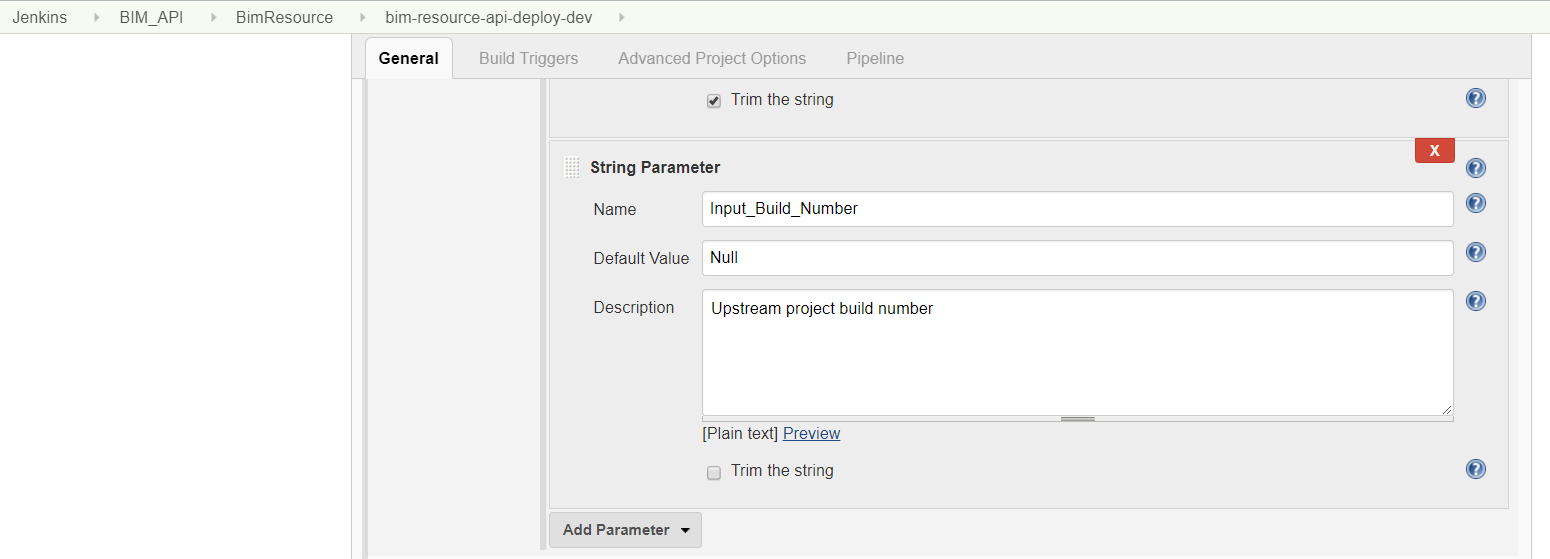
build job: 'BIM\_API/bim-resource-api-deploy-dev', propagate:false, parameters: [[$class: 'StringParameterValue', name: 'Input\_Build\_Number', value: env.BUILD\_NUMBER]]

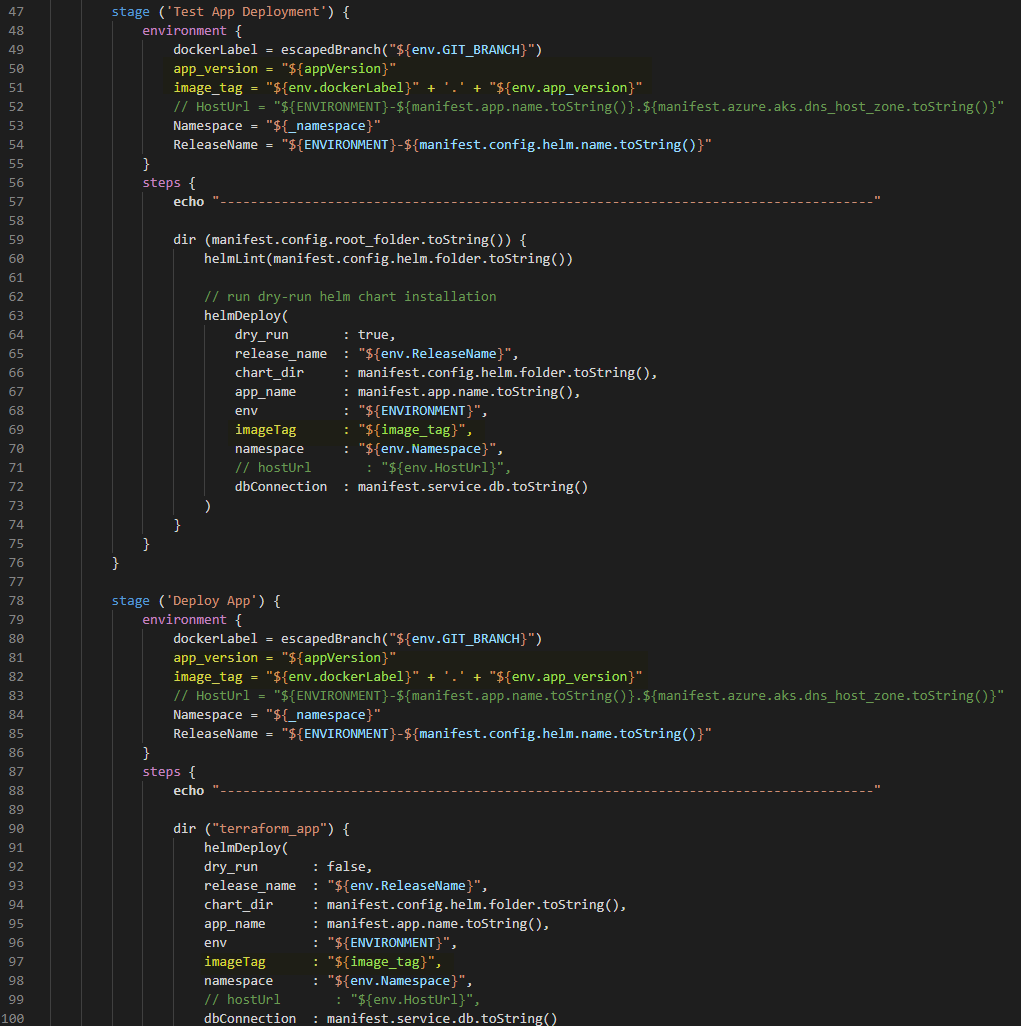
Before my change, the downstream **deploy** job was triggered by Jenkins GUI (see left picture below) when the upstream **build** job succeed. Now, since I added the “build job” line in “Jenkinsfile”, which will trigger the downstream job too, so the build trigger from Jenkins GUI side should be removed from the “BIM\_API/bim-resource-api-deploy-dev” project (see right picture below). In the “BIM\_API/bim-resource-api-deploy-qa” project, we didn’t have “Build Triggers” setup from Jenkins GUI side, because we manually deploy to QA environment by click the “Build with Parameters” button.

🡪

From the downstream deploy “BIM\_API/bim-resource-api-deploy-dev” project side, to pick up the ‘Input\_Build\_Number’, we need to do 2 things:

1, From Jenkins - select “BIM\_API/bim-resource-api-deploy-dev” – Configure – Under “This project is parameterized” – Add a “String Parameter” with the following values: Name: Input\_Build\_Number, Default Value: Null, Description: Upstream project build number.





2, Modify the “Deploy.Jenkinsfile” line 11, 50, 51, 69, 81, 82, and 97 to take in the ‘Input\_Build\_Number’. The deployment will be triggered by

helm upgrade --wait --recreate-pods --dry-run --debug --install ${args.release\_name} '${args.chart\_dir}' --set environment=${args.env},image.tag=${args.imageTag},appconfig.dbConnection='${args.dbConnection}',namespace=${args.namespace},appName=${args.app\_name} --namespace=${args.namespace}

where “image.tag” decides which docker image inside the ACR will be deployed. Above “helm upgrade” command maybe translated to

helm upgrade --wait --recreate-pods --dry-run --debug --install dev-bim-resource-api app-chart --set environment=dev,image.tag=dev.1.0.0-alpha-000020,appconfig.dbConnection=db\_url,namespace=dev-encelium,appName=bim-resource-api --namespace=dev-encelium

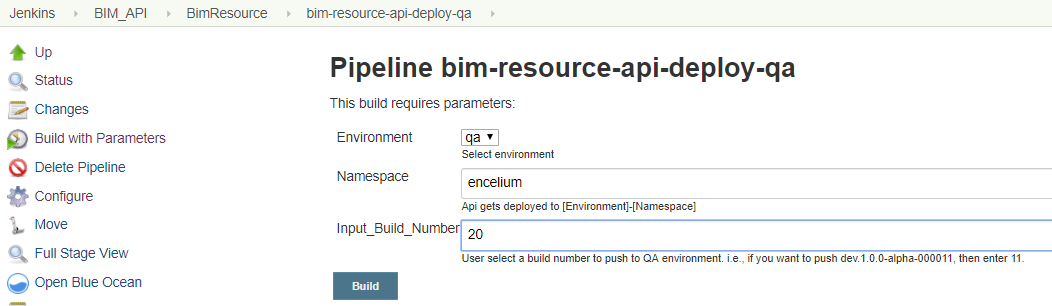
note: “dry run” in the “Deploy.Jenkinsfile” means a rehearsal of a performance or procedure before the real one.

## pick the docker image with right version to manually deploy to QA environment

In the “BIM\_API/bim-resource-api-deploy-qa” project, we didn’t have “Build Triggers” setup from Jenkins GUI side, also the “build.job” from the upstream build project will only trigger a build for “BIM\_API/bim-resource-api-deploy-dev” project but not “BIM\_API/bim-resource-api-deploy-qa” project:

build job: 'BIM\_API/bim-resource-api-deploy-dev', propagate:false, parameters: [[$class: 'StringParameterValue', name: 'Input\_Build\_Number', value: env.BUILD\_NUMBER]]

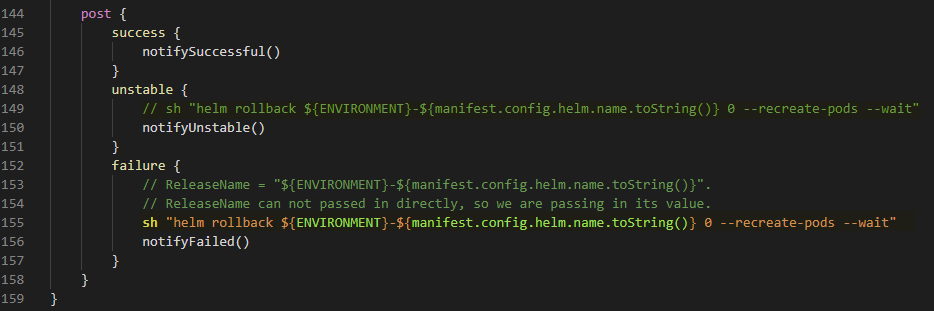
Due to above reasons, we manually deploy to QA environment by click the “Build with Parameters” button. From “BIM\_API/bim-resource-api-deploy-qa” project – Configure, we add a string parameter with exactly the same name as we added for “BIM\_API/bim-resource-api-deploy-dev”project: Under “This project is parameterized” – Add a “String Parameter” with the following values: Name: Input\_Build\_Number, Default Value: Null, Description: User select a build number to push to QA environment. i.e., if you want to push dev.1.0.0-alpha-000011, then enter 11.



“BIM\_API/bim-resource-api-deploy-qa” project also used the “Deploy.Jenkinsfile” to deploy once you click the “Build with Parameters” button and enter the build # you want to deploy inside the “Input\_Build\_Number” field. Because we used the same name “Input\_Build\_Number” for both “BIM\_API/bim-resource-api-deploy-dev” and “BIM\_API/bim-resource-api-deploy-qa” projects in Jenkins, the “Deploy.Jenkinsfile” can work for both projects.

## able to roll back to last release if the deployment failed

We use helm as a package manager to install/deploy applications. Helm just keeps a record of what's being deployed. To roll back, simply call “helm rollback <ReleaseName> 0” in “Deploy.Jenkinsfile” “post – failure” sections, 0 means to roll back to previous version. However, it is not working as expected, we need to add “--recreate-pods --wait” at the end of this command.

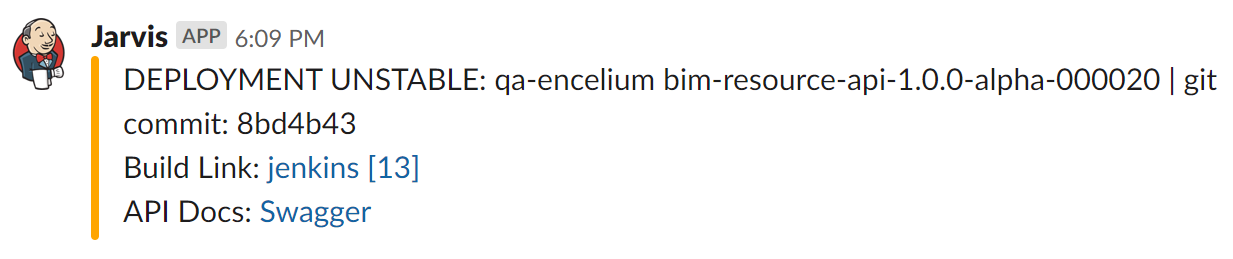
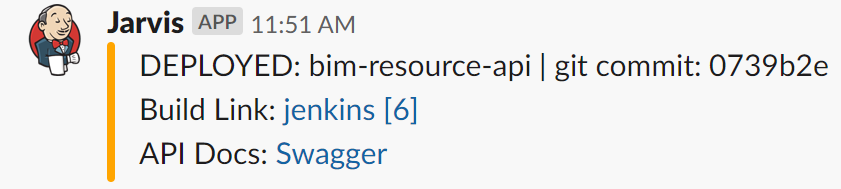


I added line 155 in “Deploy.Jenkinsfile”. I also added line 149 but commented it out, because currently our sanity and regression tests have expected “NOT IMPLEMENTED” failures embedded, which will cause deployment end up with status unstable, which should not rollback at this point of time, but once we get rid of all the “NOT IMPLEMENTED” failures from our tests, then I shall re-enable line 149.

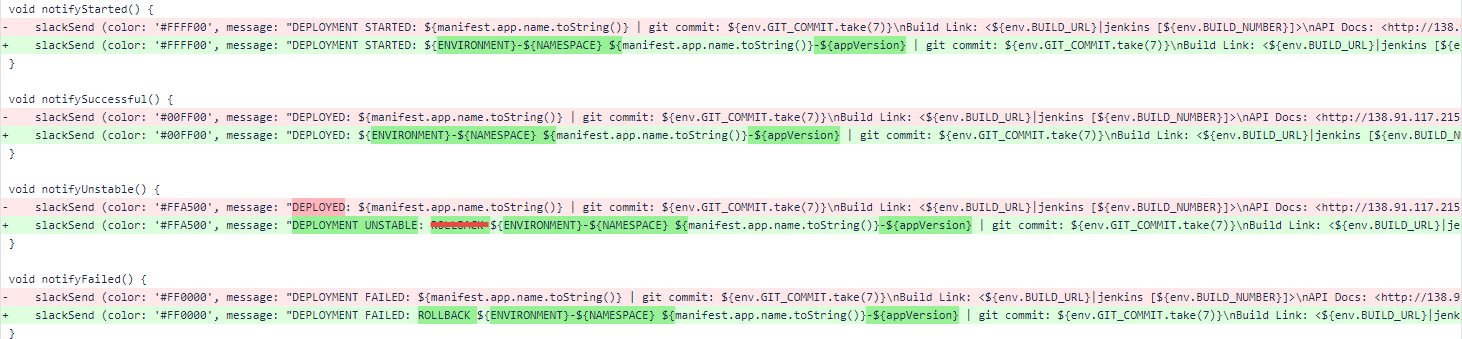
Note: The reason for us not run qa created sanity and regression tests in build project but in deploy project after “helmDeploy” is because our qa test scripts are pointing to real URLs, which only get updated once deployed.

## modify slack deployment alert to contain version information

Before my implementation, our slack alert does not specify which environment it deployed to and with which version (see picture below on the left). After my implementation, it not only specifies which environment it deployed to, but also mentioned the version (see picture above on the right).



The implementation is to modify “Deploy.Jenkinsfile” as follows:



## implement above for all the APIs (this is a side-effect when you use Microservices…)