

Sami Amoura - Description de la chaine d'Intégration Continue du projet battleboat

Rappel du contexte

Battelboat.js est une application web permettant de jouer à un jeu de bataille navale. Celle-ci comprend une IHM (front-end) pour pouvoir jouer des parties ainsi qu'une base de données (backend-end) MySQL afin de pouvoir enregistrer les données liées aux parties. Pour que l'application web soit disponible et accessible il est impératif que la base de données soit déployée et fonctionnelle.

L'objectif de cette pipeline est donc de déployer une infrastructure permettant de tester que l'application soit correctement déployée.

Notre infrastructure sera composée des éléments suivants :

- Source Code Management (SCM) : GitHub
- Outils d'Intégration Continue : Jenkins
- Conteneurisation : Docker
- Provider Cloud : AWS

Nous utiliserons le provider AWS pour déployer notre infrastructure car celui-ci nous permet aisément de disposer de l'ensemble des ressources souhaitées. Nous déploierons nos instances grâce à CloudFormation pour privilégier l'IaC.

Nous utiliserons l'outil Jenkins comme orchestrateur afin de réaliser notre pipeline CI. Effectivement, celui-ci nous permet grâce à sa polyvalence d'intégrer une multitude d'outils nécessaires à la réalisation de notre chaine d'intégration.

Nous installerons et utiliserons les plug-ins suivant :

- GitHub Integration : Afin de pouvoir intégrer notre repository GitHub directement à Jenkins
- docker-build-step : Afin de pouvoir exécuter les commandes Docker via Jenkins
- Docker Compose Build Step : Afin de pouvoir exécuter les commandes docker-compose via Jenkins
- Embeddable Build Status : Afin de pouvoir publier des badges et les montrer à la communauté

Nous utiliserons Docker, afin de pouvoir déployer notre application et ses différents services avec une certaine agilité. De plus, nous utiliserons aussi docker-compose pour faciliter le déploiement et la suppression de notre application après les tests de disponibilité de l'application.

Nous utiliserons DockerHub comme gestionnaire d'images Docker et ainsi stocker notre artefact.

Nous utiliserons GitHub comme SCM afin de pouvoir récupérer le code de notre application mais aussi et surtout afin de pouvoir publier les modifications de notre application et ainsi pouvoir profiter des fonctionnalités liées à l'Intégration Continue (Webhooks, Trigger...) et notamment pour sa compatibilité et sa parfaite intégration avec l'outil Jenkins.

Nous travaillerons sur la branche dev afin de tester notre chaine d'Intégration Continue. Si les tests d'intégration sont concluants alors nous ferons une Pull Request qui sera contrôlée par un manager/Techlead. Si le manager/Techlead valide et effectue la Merge Request sur la branche master alors cela déclenchera automatiquement le pipeline permettant de ré-exécuter les tests mais aussi de pousser notre artefact sur notre gestionnaire d'images.

Configuration de GitHub

Créer un repository sur GitHub.

Il faut maintenant configurer notre SCM GitHub afin que celui-ci puisse avertir notre exécuter de tâches Jenkins qu'il effectue le pipeline lorsque qu'un évènement (push) sera effectué sur le repository.

Se rendre sur notre repository GitHub > Settings > Webhooks > Entrez votre mot de passe car il s'agit d'une action sensible.

Remplir les champs suivants :

- « Payload URL » : adresse_de_votre_serveur_jenkins/**github-webhook/**
- « Which events would you like to trigger your webhooks » : Juste the push event.

Et Validez.

Notre repository est à présent est intégré à notre serveur Jenkins.

Webhooks / Manage webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

Content type

Secret

Which events would you like to trigger this webhook?

☒ Just the push event.

☐ Send me **everything**.

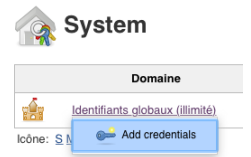
☐ Let me select individual events.

☒ **Active**
We will deliver event details when this hook is triggered.

Configuration de Jenkins

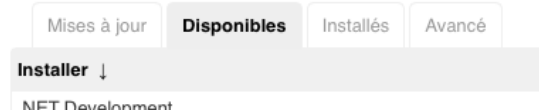
❖ Création des credentials

Se rendre sur Jenkins > Identifiants > System > Cliquez sur « Identifiants globaux (illimité) »
Cliquez sur « Add credentials » et ajoutez-le « login/password » de votre compte DockerHub.



❖ Ajout et téléchargement des plug-ins

Se rendre dans Jenkins > Administrer Jenkins > Gestion des plug-ins
Sur la page suivante cliquez sur « Disponibles »



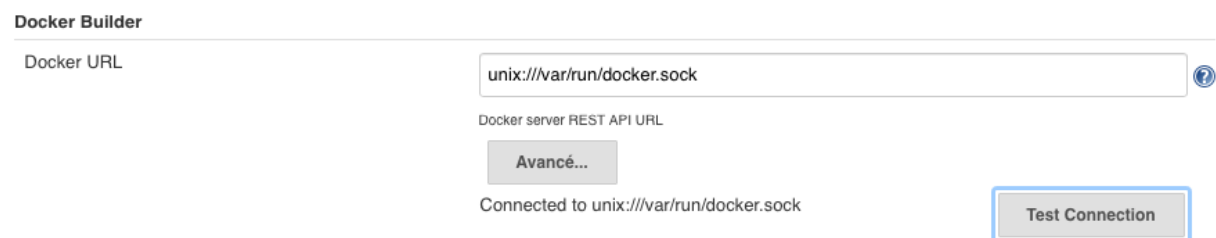
Ajoutez les 4 plug-ins suivants :

- GitHub Integration
- docker-build-step
- Docker Compose Build Step
- Embeddable Build Status

Puis cliquez sur le bouton « Télécharger maintenant et Installer après redémarrage ». Jenkins va télécharger les plug-ins et redémarrer.

❖ Configuration du plug-in docker-build-step

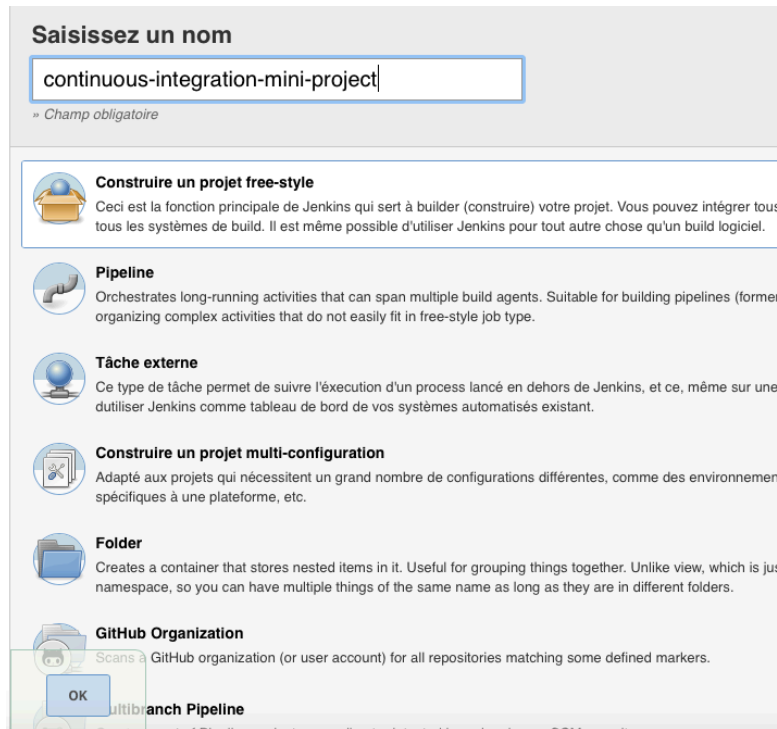
Se rendre dans Jenkins > Administrer Jenkins > Configurer le système > Docker Builder.
Entrez la commande suivante « unix:///var/run/docker.sock » et testez la connexion. Si la connexion est établie le message suivant s'affiche « Connected to unix:///var/run/docker.sock ». Sauvegardez et quittez la page.



❖ Création et configuration du job

Création du job

Sur la page principale cliquez sur « Nouveau Job » puis sur la page qui apparait ; Donnez un nom au projet « continuous-integration-mini-projet », sélectionnez « Construire un mini-projet » et cliquez sur « OK »



Saisissez un nom

continuous-integration-mini-projet

» Champ obligatoire

Construire un projet free-style
Ceci est la fonction principale de Jenkins qui sert à builder (construire) votre projet. Vous pouvez intégrer tous les systèmes de build. Il est même possible d'utiliser Jenkins pour tout autre chose qu'un build logiciel.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (former organizing complex activities that do not easily fit in free-style job type).

Tâche externe
Ce type de tâche permet de suivre l'exécution d'un process lancé en dehors de Jenkins, et ce, même sur une utiliser Jenkins comme tableau de bord de vos systèmes automatisés existant.

Construire un projet multi-configuration
Adapté aux projets qui nécessitent un grand nombre de configurations différentes, comme des environnements spécifiques à une plateforme, etc.

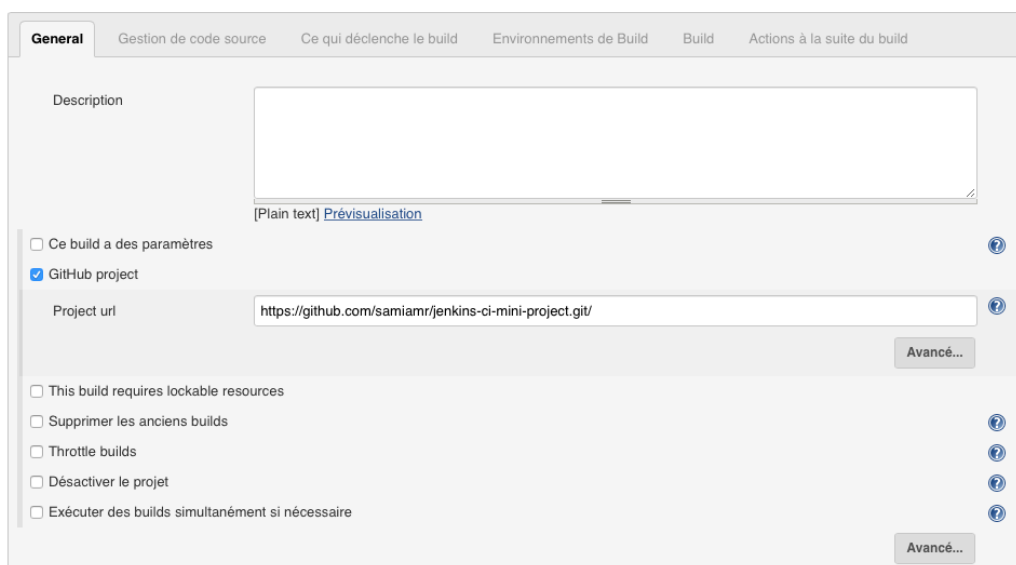
Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just namespace, so you can have multiple things of the same name as long as they are in different folders.

GitHub Organization
Scans a GitHub organization (or user account) for all repositories matching some defined markers.

OK

Choix de notre repository comme espace de travail

Pour configurer notre job, se rendre dans « General », cochez la case « GitHub project » et entrez l'url de votre repos GitHub créé préalablement :



General Gestion de code source Ce qui déclenche le build Environnements de Build Build Actions à la suite du build

Description

[Plain text] [Prévisualisation](#)

☐ Ce build a des paramètres

☒ GitHub project

Project url

[Avancé...](#)

☐ This build requires lockable resources

☐ Supprimer les anciens builds

☐ Throttle builds

☐ Désactiver le projet

☐ Exécuter des builds simultanément si nécessaire

[Avancé...](#)

Choix de la branche qui déclenchera notre trigger

Se rendre dans « Gestion du code source », ajoutez l'URL de votre repository dans le champ « Repository URL » et enfin ajoutez la branche sur laquelle vous souhaitez que le trigger s'active lorsque que puherez votre code.

Dans mon cas de figure j'utilise la branche dev afin d'effectuer les tests et si tout est OK j'effectuerai une pull request à un manager fictif qui pourra merger la branche dev sur la branche master afin de déployer l'artefact Docker sur le DockerHub.

Gestion de code source

☐ Aucune Gestion de code source

☒ Git

Repositories

Repository URL:

Credentials: Ajouter

Avancé...

Add Repository

Branches to build

Branch Specifier (blank for 'any'): X

Add Branch

Navigateur de la base de code

Additional Behaviours Ajouter

☐ Subversion

Choix de l'évènement qui déclenchera notre trigger

Se rendre dans « Ce qui déclenche le build » afin de définir l'évènement qui ce qui déclenchera le build de notre artefact.

Cochez la case « GitHub hook trigger for GITScm polling » afin de définir que notre évènement déclencheur(trigger) du build sera un le push de notre code sur notre repository GitHub.

Ce qui déclenche le build

☐ Déclencher les builds à distance (Par exemple, à partir de scripts)

☐ Construire après le build sur d'autres projets

☐ Construire périodiquement

☐ GitHub Branches

☐ GitHub Pull Requests

☒ GitHub hook trigger for GITScm polling

☐ Scrutation de l'outil de gestion de version

Création du Pipeline

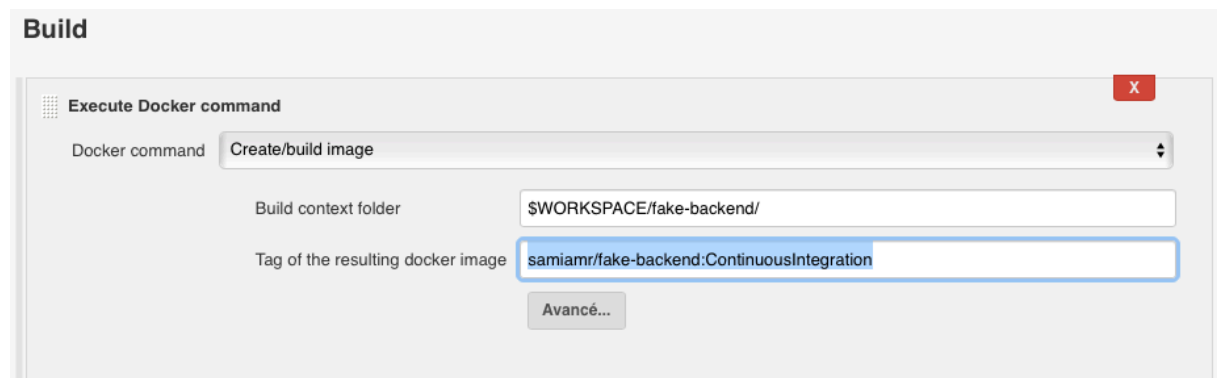
Nous allons maintenant configurer les étapes de notre pipeline CI. Se rendre dans la partie « Build » et « Ajoutez une étape au build »

→ Build de l'image

Choisir « Execute Docker command » : Create build image

Remplir les champs suivants :

- « Build context folder » : `$WORKSPACE/fake-backend/`
- « Tag of the resulting docker image » : `samiarm/fake-backend:ContinuousIntegration`



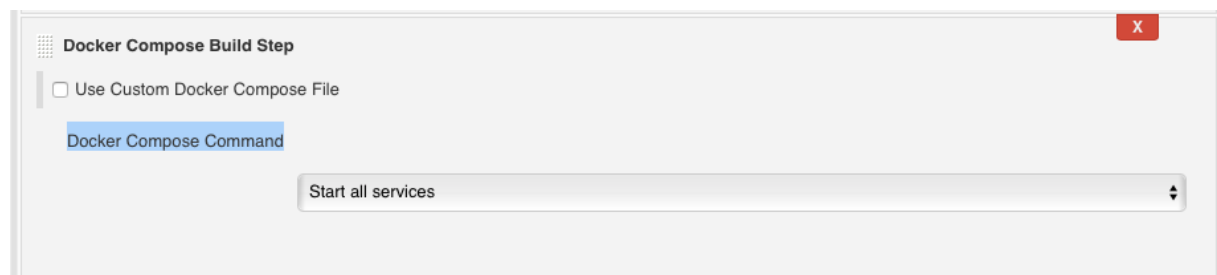
The screenshot shows a configuration window titled 'Build' with a sub-header 'Execute Docker command'. It contains three input fields: 'Docker command' with the value 'Create/build image', 'Build context folder' with the value '\$WORKSPACE/fake-backend/', and 'Tag of the resulting docker image' with the value 'samiarm/fake-backend:ContinuousIntegration'. An 'Avancé...' button is located at the bottom right of the configuration area.

→ Déploiement de nos services (application)

Pour déployer notre application nous utiliserons docker-compose

Choisir « Docker Compose Build Step ».

Remplir le champ « Docker Compose Command » : Start all services



The screenshot shows a configuration window titled 'Docker Compose Build Step'. It has a checkbox labeled 'Use Custom Docker Compose File' which is unchecked. Below it, the 'Docker Compose Command' field is highlighted in blue and contains the text 'Start all services'.

→ Test de l'application

Nous allons maintenant effectuer la partie testing et tester que notre application est déployée et fonctionnelle à l'aide d'un script bash.

Choisir « Exécuter un script shell »

Entrez le script suivant qui permet de tester que l'application est correctement déployée :

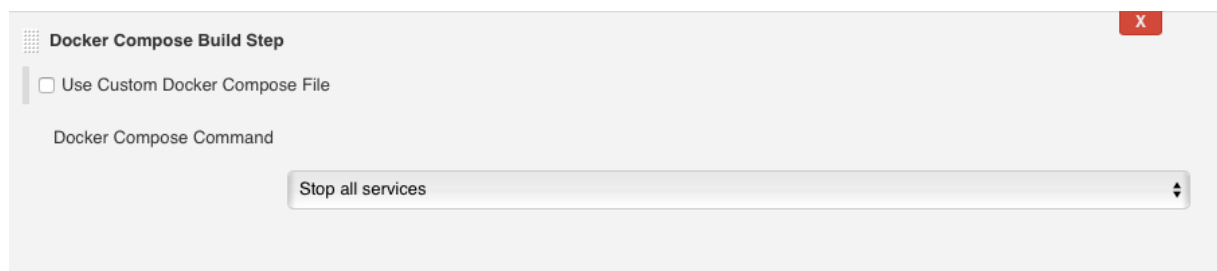
```
#!/bin/bash
sleep 20s
EXTERNAL_HOSTNAME=`curl http://169.254.169.254/latest/meta-data/public-hostname` && if [[
$(curl -s -o /dev/null -w "%{http_code}" $EXTERNAL_HOSTNAME:8181/health) == 200 ]]; then
echo "Application déployée"; exit 0; else echo "Application injoignable"; exit 1; fi
```

→ Clean de l'environnement de build

Une fois le test réalisé, il faut alors nettoyer notre environnement de build. Nous allons stopper l'ensemble des services déployés avec le docker-compose.

Choisir « Docker Compose Build Step »

Remplir le champ « Docker Compose Command » : Stop all services



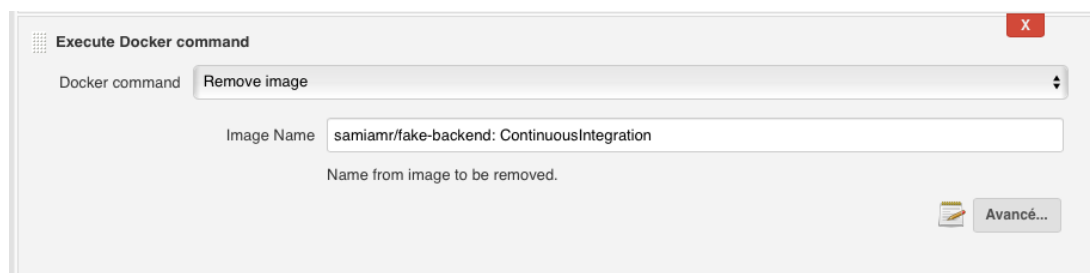
→ Suppression de l'image.

Nous allons maintenant supprimer l'image de notre environnement. Choisir « Execute Docker command » :

Choisir « Execute Docker command » : Remove image

Remplir le champ suivant:

- « Image Name » : samiamr/fake-backend:ContinuousIntegration



Notre pipeline est donc créé. Celui-ci sera exécuté lors d'un push de notre code sur notre repository sur la branche dev.

Si tout est exécuté avec succès nous réaliserons une PR à un manager fictif qui effectuera une MR. Cette MR reproduira un nouveau job similaire à celui-là mais avec une étape supplémentaire qui sera le push de notre artefact sur notre artefactory (DockerHub)

Il faudra donc recréer un job similaire mais avec la modification de la branche qui déclenchera notre trigger cette fois-ci ce sera la branche master :

Se rendre dans « Gestion du code source » et modifiez la branche à « master ».

Gestion de code source

☐ Aucune
☒ Git

Repositories

Repository URL:

Credentials:

Branches to build

Branch Specifier (blank for 'any'):

Navigateur de la base de code:

Additional Behaviours:

Il nous faudra aussi ajouter une nouvelle étape pour le build qui sera effectuée juste après le test de notre application. Cette étape sera le push de notre artefact docker sur le DockerHub.

Choisir « Execute Docker command » : Push image

Remplir les champs suivants :

- « Name of the image to push (repository/image) » : samiamr/fake-backend
- « Tag » : ContinuousIntegration
- « Docker registry URL » : <https://index.docker.io/v2/>
- « Registry credentials » : Mettre les credentials créés préalablement

Execute Docker command

Docker command:

Name of the image to push (repository/image):

Tag:

Registry:

Docker registry URL:

Registry credentials:

