

INF3135

Construction et maintenance de logiciels

Cours 14: Développement continu

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

1 Quiz 4

2 Travail pratique 3

3 Tests

4 Développement continu

5 GitLab-CI

6 Docker

Quiz 4

Quiz 4

Matière

- Chapitre 8: tests
- Labo 10: intégration
- Labo 11: tests

Contenu

- Une question courte sur la gestion de la **mémoire**
- Une question courte sur l'**intégration**
- Une question courte sur les **tests**
- Une question longue sur les **tests**

Travail pratique 3

Travail pratique 3

- **Dépôt:** <https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3>
- **Sujet:** <https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3/-/tree/sujet/sujet>
- Développer un petit **jeu vidéo**
- Inspiré du jeu **Super Hexagon** de Terry Cavanagh
- Avec bibliothèque **SDL2**
- **Coquille** de base disponible
- Peut être fait en **équipe** d'au plus 3
- **Attention!** je vérifie qui fait quoi

Tests

Différents types de tests

- **Compilation**: incluant bibliothèques tierces
- **Intégration**: communication intermodulaire
- **Unitaires**: teste un aspect spécifique
- **Fonctionnels**: en adéquation avec le cahier des charges
- **Non-régression**: pas de perte de fonctionnement ou d'efficacité
- **Configuration**: fonctionne dans des environnements variés
- **Performance**: consomme pas trop de temps ou de mémoire
- **Installation**: s'installe correctement sur un système

Propriétés d'un bon test

- **Juste**: il teste bien ce qu'il faut
- **Robuste**: il ne plante pas
- **Pur**: il est sans effet de bord
- **Reproductible**: il a le même comportement peu importe l'environnement
- **Pertinent**: il augmente notre confiance
- **Non redondant**: il teste quelque chose de distinct d'un autre test
- **Efficace**: il prend un temps raisonnable
- **Automatisable**: il ne nécessite pas d'intervention humaine

Tests shell

- De nombreuses **commandes shell** permettent de tester
- Avec la **sémantique** habituelle (0: succès, $\neq 0$: erreur)

Commandes utiles

- `grep`: recherche un motif
- `diff`: compare le contenu de deux fichiers
- `valgrind`: teste la gestion de la mémoire
- `test` `EXPR`: teste sur des fichiers et sur des valeurs
- → équivalent à `[EXPR]`
- → extension Bash `[[EXPR]]`

Tests internes/externes

Internes

- Basés sur l'**implémentation**
- Notamment des **structures** utilisées (répétitives, conditionnelles)
- **Cas particulier**: assertions (avec `assert.h`)
- **Outils**: graphe de flux, graphe d'appels de fonctions, ...
- **Bibliothèque** spécifique au langage, par exemple **Libtap**

Externes

- Basés sur l'**interface**
- Permet de valider la **documentation**
- **2 niveaux**: module et application
- **Outils**:
 - module → bibliothèques
 - application → le shell, **Bats**

Développement guidé par les tests

1. Ajouter un test ou plusieurs tests

Qui mettent en évidence le comportement souhaité

2. Lancer tous les tests

Les nouveaux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les nouveaux tests devraient réussir, les anciens aussi

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours

Développement continu

Quelques définitions (tirées de Wikipedia)

« **L'intégration continue** est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée »

« La **livraison continue** est une approche d'ingénierie logicielle dans laquelle les équipes produisent des logiciels dans des cycles courts, ce qui permet de le mettre à disposition à n'importe quel moment »

« Le **déploiement continu**, est une approche d'ingénierie logicielle dans laquelle les fonctionnalités logicielles sont livrées fréquemment par le biais de déploiements automatisés. »

« Le **devops** ou **DevOps** est un mouvement en ingénierie informatique et une pratique technique visant à l'unification du développement logiciel (dev) et de l'administration des infrastructures informatiques (ops), notamment l'administration système. »

Chaîne DevOps

- **Planifier** (*plan*): identifier besoins/obstacles, définir politiques
- **Créer** (*create*): coder, programmer, compiler
- **Vérifier** (*verify*): tester à différents niveaux
- **Empaqueter** (*package*): préparer la livraison, dépendances
- **Livrer** (*release*): coordination des livraisons, déploiement
- **Configurer** (*configure*): stockage, infrastructure, environnement
- **Surveiller** (*monitor*): mesures diverses, catalogue des bogues

Outils

Jira, Trello, Github, Bitbucket, GitLab, Jenkins, Travis-CI, Autotools, CMake, GitLab-CI, Docker, ...

Suite

- **GitLab-CI**: développement continu
- **Docker**: manipulation de conteneurs

GitLab-CI

GitLab-CI

- Permet de mettre en place des **pipelines** de traitement
- **Lien:** <https://docs.gitlab.com/ee/ci/>
- **Gratuit** jusqu'à 2000 minutes par mois sur [GitLab.com](https://gitlab.com)

Extrait de la [page d'accueil](#):

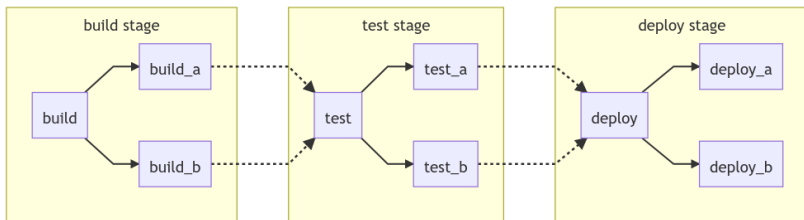
« GitLab CI/CD is a tool built into GitLab for software development through the continuous methodologies:

- Continuous Integration (CI)
- Continuous Delivery (CD)
- Continuous Deployment (CD) »

Gère autant l'**intégration** que la **livraison** et le **déploiement**

Pipeline

- **Ensemble** de tâches à compléter
- Structuré sous forme de **graphe acyclique**
- Possible aussi de relations **hiérarchiques** (parent/enfant)
- Voir **architectures de pipeline** pour plus d'informations
- 3 étapes (*stages*) par défaut: build, test, deploy
- 2 résultats possibles pour chaque tâche: succès (0) ou échec ($\neq 0$)
- Tâches d'une même étape exécutées en **parallèle**
- Tâches d'une étape suivante lancées seulement si **toutes** les tâches de l'étape précédente ont réussi



Mise en place (1/2)

- À l'aide d'un fichier de **configuration** nommé `.gitlab-ci.yml`
- Qui doit respecter la syntaxe **YAML**
- **Exemple:**

```
stages:
  - build
  - test
  - deploy

image: alpine

build_a:
  stage: build
  script:
    - echo "This job builds something."

build_b:
  stage: build
  script:
    - echo "This job builds something else."

test_a:
  stage: test
  script:
    - echo "This job tests something. It will only run when all jobs in the"
    - echo "build stage are complete."
```

Mise en place (2/2)

```
test_b:
  stage: test
  script:
    - echo "This job tests something else. It will only run when all jobs in"
    - echo "the build stage are complete too. It will start at about the"
    - echo "same time as test_a."

deploy_a:
  stage: deploy
  script:
    - echo "This job deploys something. It will only run when all jobs in"
    - echo "the test stage complete."

deploy_b:
  stage: deploy
  script:
    - echo "This job deploys something else. It will only run when all jobs"
    - echo "in the test stage complete. It will start at about the same time"
    - echo "as deploy_a."
```

- Déclarer des **dépendances** plus fines: **needs**
- **Hiérarchiser** des pipelines: **trigger**
- Exécuter un script **avant** ou **après** chaque tâche: **before_script** et **after_script**, ...

Artéfacts

- **Fichier** ou **répertoire** produit lors d'une tâche
- Permet aux tâches de **communiquer** entre elles
- Voir [page sur artéfacts](#) pour plus d'informations
- **Exemple:**

```
test:
  stage: test
  script:
    - make test
  artifacts:
    paths:
      - ./tests/isomap.png
    expire_in: 1 week
```

- Télécharger [tous les artéfacts](#) ou [seulement le fichier](#)

```
# URL de l'archive
https://example.com/<namespace>/<project>/-/jobs/artifacts/<ref>/download?job
=<job_name>
# URL du fichier
https://example.com/<namespace>/<project>/-/jobs/artifacts/<ref>/raw/<
path_to_file>?job=<job_name>
```

Image

- Les tâches sont exécutées dans un « environnement » précis
- On appelle cet « environnement » une **image**
- Spécifié à l'aide du mot-clé **image**
- Par défaut, l'image est `ruby:2.1`
- Par défaut, les images sont récupérées sur **Docker Hub**

Registre de conteneurs

- GitLab supporte aussi un **registre de conteneurs**
- En anglais, *container registry*
- Permet d'héberger des images par **projet**
- Plutôt que de mettre des images **publiques** sur Docker Hub

Docker

Docker

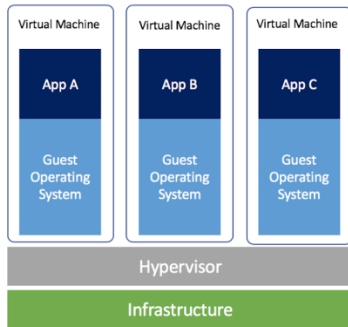
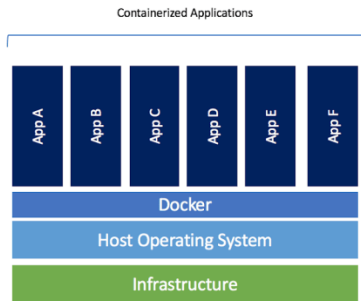
- Site officiel: <https://www.docker.com/>
- Permet de manipuler des **conteneurs**



Extrait de [docker.com](https://www.docker.com/)

« A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. »

Comparaison entre conteneur et machine virtuelle (1/2)



(Source: [blog de Jenny Fong](#))

Comparaison entre conteneur et machine virtuelle (2/2)

Machine virtuelle

- Virtualisation au niveau **matériel**
- Plus **lourd**
- Accès plus **lent**
- Performance **limitée**
- Isolation **complète** et donc plus **sécuritaire**

Conteneurs

- Virtualisation au niveau **logiciel**
- Plus **léger**
- Accès plus **rapide** et meilleur **passage à l'échelle**
- Performance **native**
- Isolation au niveau des **processus** et donc moins **sécuritaire**

Installation

- **Dépend** de la distribution
- **Debian** et dérivée (comme Ubuntu, Mint):

```
# Mise à jour des paquets
$ sudo apt update
# Optionnel: déinstaller anciennes versions
$ sudo apt remove docker docker-engine docker.io containerd runc
# Installation
$ sudo apt install docker.io
$ sudo systemctl start docker.
# Ou: sudo service docker start.
```

- **MacOS:** Docker Desktop pour MacOS
- **Windows:** Docker Desktop pour Windows
- Un peu de configuration à faire pour utiliser en **ligne de commande**

Quelques opérations

- **run**: **lancer** une commande dans un conteneur
- **image**: **gérer** les images
- **images**: **lister** les images
- **build**: **construire** une image
- **login**: se **connecter** à un registre
- **logout**: se **déconnecter** d'un registre
- **pull**: **télécharger** une image
- **push**: **téléverser** une image
- **volume**: **gérer** les volumes

Voir [documentation complète](#) pour plus d'informations

Dockerfile (1/2)

- Décrit comment **construire** des images
- Essentiellement, c'est un **script shell**
- Mais avec plus de **fonctionnalités**:

Quelques commandes

- **FROM**: précise l'image de base
- **RUN**: exécute une commande
- **EXPOSE**: expose un port
- **ENV**: assigne une variable d'environnement
- **WORKDIR**: change le répertoire courant
- **VOLUME**: monte un volume, ...

Dockerfile (2/2)

```
# Précise l'image de départ
FROM ubuntu:18.04
```

```
# Installe des paquets
```

```
RUN apt-get update && apt-get install -y \
    build-essential \
    git \
    graphviz \
    libcairo-dev \
    libjansson-dev \
    pandoc \
    valgrind \
    pkg-config
```

```
# Installe manuellement Libtap
```

```
RUN git clone https://github.com/zorgnax/libtap.git && \
    cd libtap && \
    make && \
    make install && \
    ldconfig
```

```
# Installe manuellement Bats
```

```
WORKDIR /Applications
RUN git clone https://github.com/bats-core/bats-core.git bats-core && \
    bash bats-core/install.sh .
```

```
# Pour trouver Bats de n'importe où
```

```
ENV PATH="${PATH}:/Applications/bin"
```