

La mise en place de l'intégration continue (CI)

Table des matières

I. Intégration continue, principes, bonnes pratiques et bénéfices	3
II. Exercice : Quiz	7
III. Premier pas dans l'intégration continue avec Gitlab	8
A. Intégration avec GitLab	9
IV. Exercice : Quiz	13
V. Essentiel	15
VI. Exercice : Quiz	15
VII. Auto-évaluation	16
A. Exercice	16
B. Test	17
Solutions des exercices	18

I. Intégration continue, principes, bonnes pratiques et bénéfices

Durée : 1 h 30

Prérequis : savoir utiliser GIT, Connaître les principes du DevOps et de l'intégration

Environnement de travail : un ordinateur connecté à Internet

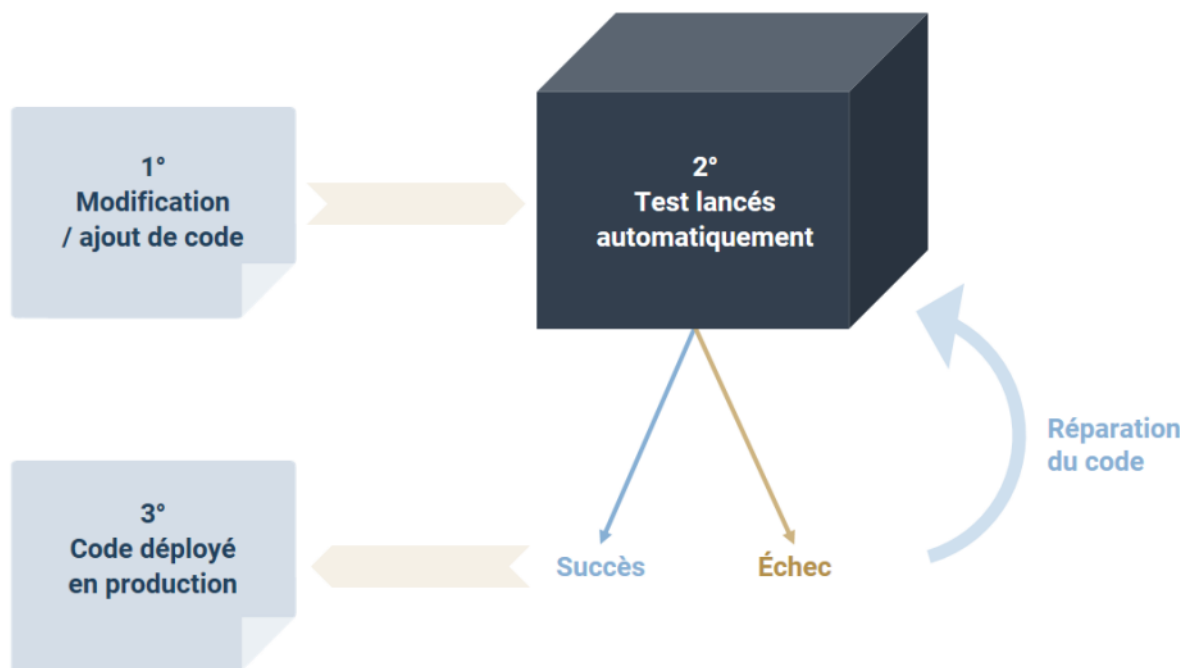
Contexte

L'un des principaux concepts du DevOps est l'intégration continue. L'intégration continue (ou CI, pour Continuous Integration) est une des composantes principales dans un environnement de développement suivant les méthodes agiles. L'objectif de la mise en place de ces pratiques est d'accélérer le développement d'une application et de garantir la qualité du code.

L'Intégration Continue (CI)

L'Intégration Continue (CI) est une bonne pratique agile et DevOps qui visent à permettre aux développeurs d'intégrer le plus fréquemment possible leur modification dans le dépôt de code ou dans la branche principale. L'objectif est de diminuer les chances de vivre un « *enfer de l'intégration* » en attendant le dernier moment et la toute fin d'un sprint ou d'un projet pour fusionner / merger le code produit par tous les développeurs d'une équipe. L'intégration continue visant à automatiser le déploiement, elle permet aux équipes de répondre aux exigences métier plus rapidement, d'améliorer la qualité du code et d'accroître la sécurité des applications.

Ainsi, avec l'intégration continue la phase de tests automatisés est complètement intégrée au flux de déploiement.



Les avantages de l'intégration continue sont de pouvoir :

- Tester immédiatement des modifications,
- Notifier rapidement les développeurs en cas de code manquant ou incompatible,
- Détecter et réparer de manière continue les problèmes d'intégration continue et éviter d'être surpris à la dernière minute,

- Versionner le code afin de toujours disposer d'une application permettant un test, une démonstration ou une distribution.

La fusion

L'intégration continue met en place des mécanismes permettant aux développeurs de merger plus fréquemment leurs développements dans une « *branche partagée ou un tronc* ». Les modifications une fois fusionnées sont validées par la création automatique de l'application (le build) et l'exécution de tests automatisés (généralement des tests unitaires et d'intégration) qui permettent de vérifier qu'aucune des modifications ne provoque de dysfonctionnement au sein de l'application. Il s'agit de tester le maximum des composants du code permettant le fonctionnement de notre application, comme les classes, les fonctions et les différents modules. Si un conflit est détecté entre le code existant et le code que l'on souhaite fusionner, l'intégration continue permet de mettre en évidence et de le résoudre plus facilement, plus fréquemment et plus rapidement.

L'intégration continue va se faire en 5 étapes :

1. Planification du développement,
2. Gestion du code,
3. Tester,
4. Mesurer la qualité du code,
5. Gérer les livrables de l'application.

Certaines de ses étapes se feront automatiquement à l'aide d'outils. Dans la suite de ce module, nous allons voir chacune des étapes de l'intégration continue, ainsi que, pour chaque étape, les outils que vous pouvez utiliser pour la mettre en place.

Voyons maintenant les étapes de l'intégration continue

Fondamental Étape 1 : Planification du développement

Avant même de parler code, l'intégration continue démarre par une parfaite organisation et planification des développements. L'objectif recherché est de découper le projet en une multitude de tâches et de composer un répertoire de tâche (backlog). Les développeurs viendront au fil des semaines s'attribuer des tâches à résoudre, chacune de ses tâches fera l'objet d'un développement en suivant tous les processus de CI jusqu'à se retrouver en production chez le client.

Cette planification est étroitement liée à la méthodologie Scrum. Les tâches réalisées suivent un ordre logique et sont priorisées par le chef de projet ou *product owner*.

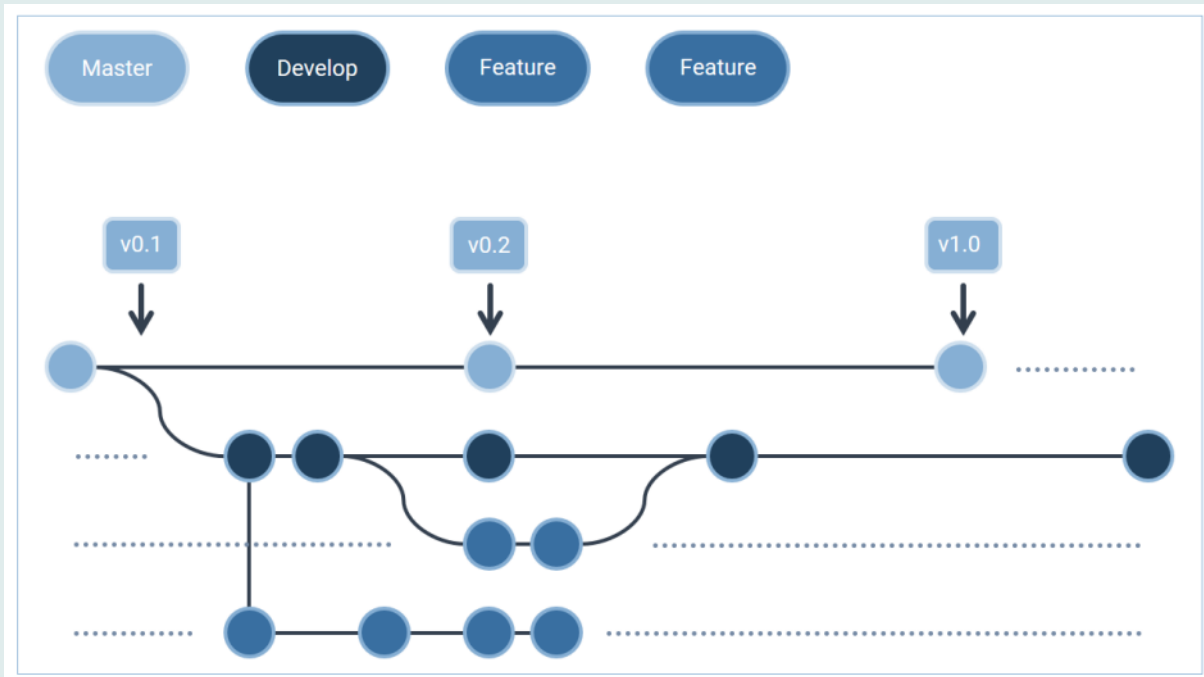
Exemple

Backlog	To Do	In progress	Done
Tâche 6	Tâche 5	Tâche 3	Tâche 1
Tâche 7		Bug 110	Tâche 2
Tâche 8			
Bug 111			
Bug 113			

Un backlog peut très bien être réalisé sur un tableau physique ou logiciel (Trello, Gitlab, etc.).

Fondamental **Étape 2 : Gestion du code****Le contrôle de code source**

Même si cela semble évident, il s'agit d'un des principaux facteurs : tous les membres de l'équipe doivent utiliser la même source (le même dépôt ou repository) lorsqu'ils travaillent sur le code. Le code source se doit d'être disponible à chaque instant sur un dépôt central. Chaque développement doit faire l'objet d'un suivi de révision. Le code doit être compilable à partir d'une récupération fraîche, et ne faire l'objet d'aucune dépendance externe.

**L'orchestrateur**

Afin de concevoir un programme viable à partir du code source, les développeurs doivent le compiler, actualiser les bases de données, gérer les dépendances et déplacer les fichiers au bon endroit. Ce processus de build (construction) est automatisé à l'aide d'un orchestrateur. Dans l'idéal, il faudrait pouvoir exécuter un processus de build à l'aide d'une seule commande.

La première étape gérée par l'orchestrateur sera toujours une étape de compilation du code afin de vérifier qu'aucun nouveau morceau de code ne vient casser la compilation. Cette étape permet de garantir à chaque développeur la possibilité de récupérer un code exécutable.

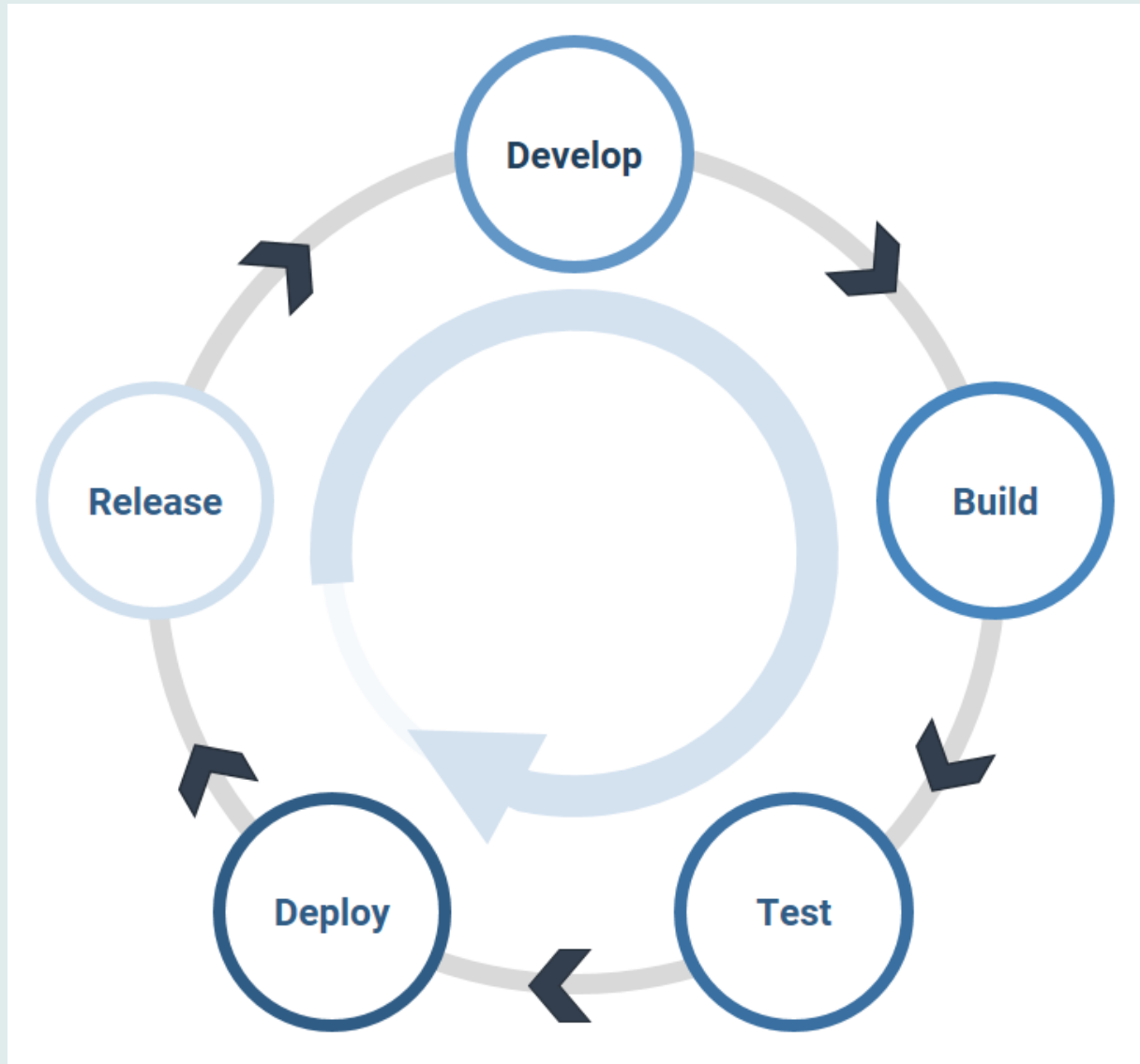
L'orchestration des étapes de votre intégration continue peut se faire grâce à des outils comme Jenkins, TeamCity, Azure DevOps, GitLab CI, Concourse CI, Travis CI ou Bamboo.

Fondamental **Étape 3 : Tester le code**

L'intégration continue permet d'avoir un retour plus rapide sur les développements. Il n'y a donc plus besoin d'attendre plusieurs semaines pour identifier les erreurs et les corriger. En intégrant des mécanismes de test dans le processus de build, l'équipe profite d'une plus grande automatisation et ainsi d'une accélération de l'intégration continue.

En règle générale l'orchestrateur se charge de lancer les tests unitaires à la suite de la compilation. Ces tests doivent s'exécuter de la manière la plus rapide possible, afin d'avoir un feedback (retour d'informations) le plus rapide lui aussi. Pour arriver à ce niveau, il est nécessaire que les tests unitaires n'aient aucune dépendance vis-à-vis de systèmes externes, comme par exemple une base de données, ou même le système de fichiers de la machine.

Ces tests unitaires sont généralement mis en place avec l'aide d'un framework afin de garantir que le code respecte un certain niveau de qualité.



Fondamental Étape 4 : Mesurer la qualité du code

Un code respectant les bonnes pratiques et de qualité supérieure contient moins de bugs, il est donc plus facile à maintenir et à comprendre. Cependant la définition précise d'un code de qualité est assez subjective et peut varier en fonction de sa société, son équipe et même entre membres d'une même équipe.

Bien heureusement, les bonnes pratiques et les normes de codage permettent de donner un cadre commun au sein d'une équipe et d'arbitrer certaines situations. Si les développeurs travaillant sur le même projet appliquent les mêmes normes / pratiques, le code produit sera plus lisible, pour les membres travaillant sur ce projet. De ce fait, le code sera beaucoup plus simple à appréhender pour les nouveaux développeurs arrivant dans l'équipe.

C'est à travers des revues de code ou des exercices comme la programmation en binôme que les développeurs plus expérimentés peuvent juger de la qualité du code produit par son équipe. Néanmoins, des outils de mesure de qualité de code existent et il ne serait pas réaliste de mettre en place des bonnes pratiques sur des projets d'envergure sans ce type d'outil.

Ces outils sont conçus pour analyser le code source de votre application et vérifier s'il respecte certaines règles. Ces règles peuvent englober énormément d'aspects différents comme des normes de codage, des moyennes de complexité du code ou le calcul du nombre de lignes par classe. D'autres se focalisent sur des analyses statistiques plus poussées, ou sur la recherche de bugs.

Exemple Outils mesurant la qualité du code

SonarQube ou GitLab Code Quality.

Fondamental Étape 5 : Gérer les livrables de l'application

Un livrable est un service ou un produit accessible au client. Il s'agit de l'achèvement d'un processus, d'une partie de projet, d'une tâche ou de livraison de la totalité du projet. Un projet est souvent composé de plusieurs livrables. Ces livrables sont appelés artefacts. Ces artefacts doivent être accessibles à toutes les parties prenantes de l'application. Ces artefacts peuvent être livrés directement au client et donc mis en production ou alors continuer un processus autre que les tests unitaires (test de performance, test de bout en bout, etc.).



Pour résumer, voici les grandes étapes de l'intégration continue :

1. La planification du développement avec la méthode Scrum,
2. La compilation et l'intégration du code,
3. Le lancement automatique des tests unitaires, pour vérifier que le code fonctionne comme prévu,
4. La mesure de la qualité du code produit, pour vérifier qu'il sera facilement maintenable sur la durée,
5. La gestion des livrables pour obtenir les artefacts prêts à être déployés en production ou sur l'environnement de test.

Exercice : Quiz

[solution n°1 p.19]

Question 1

Quel est le terme employé par le Product Owner pour définir le répertoire des tâches ?

- ☐ Le frontlog
- ☐ Le backlog
- ☐ Le front-end
- ☐ Le back-end

Question 2

Quel outil va permettre de gérer les étapes 2 à 5 composant notre pipeline de CI ?

- ☐ La compilation
- ☐ Le package
- ☐ Le backlog
- ☐ L'orchestrateur

Question 3

Quelle première étape du build permet à l'orchestrateur de garantir un code utilisable à n'importe quel développeur souhaitant travailler sur le projet ?

- ☐ Le déploiement
- ☐ Le test
- ☐ La compilation
- ☐ Le contrôle qualité

Question 4

Un orchestrateur peut-il gérer un backlog ?

- ☐ Vrai
- ☐ Faux

Question 5

Un artefact peut-il être passé en production ?

- ☐ Vrai
- ☐ Faux

Question 6

La qualité du code doit-elle être revue par un développeur senior seulement ?

- ☐ Vrai
- ☐ Faux

III. Premier pas dans l'intégration continue avec Gitlab

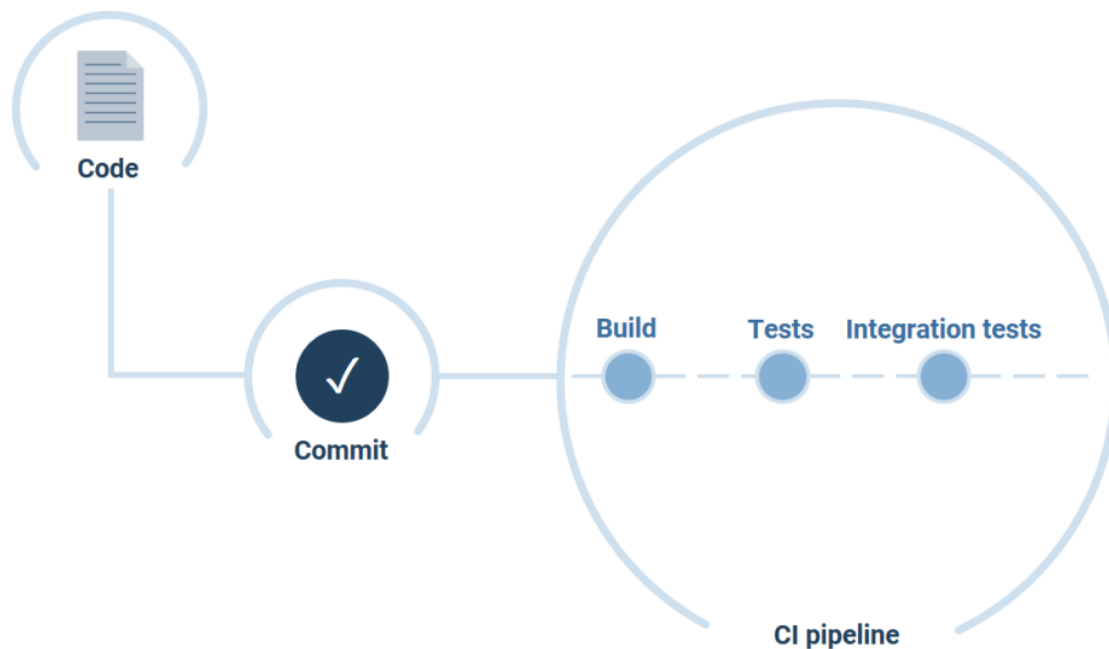
Qu'est-ce que GitLab ?

Gitlab est un des outils incontournables en matière de gestion de projets web. En comparaison avec Github, la palette de fonctionnalités de GitLab se veut nettement plus fournie. En plus de la gestion de dépôts de code source GitLab permet :

- La gestion de projet,
- La planification et la priorisation des tâches,
- La gestion du build,
- L'automatisation des tests logiciel,
- La visualisation du pipeline CI / CD,
- Le monitoring,

- La mise en place de sécurité applicative.

Dans le cadre de la mise en place de l'intégration continue, c'est l'onglet pipeline CI / CD qui nous intéressera le plus.



Environnement de travail

Afin de mettre en place un pipeline d'intégration continue avec GitLab, il vous faudra créer un compte GitLab sur le site officiel¹. Vous pouvez utiliser votre compte GitHub pour vous connecter). Une fois connecté vous pouvez créer un projet « *new project* » - « *create blank project* ».

A. Intégration avec GitLab

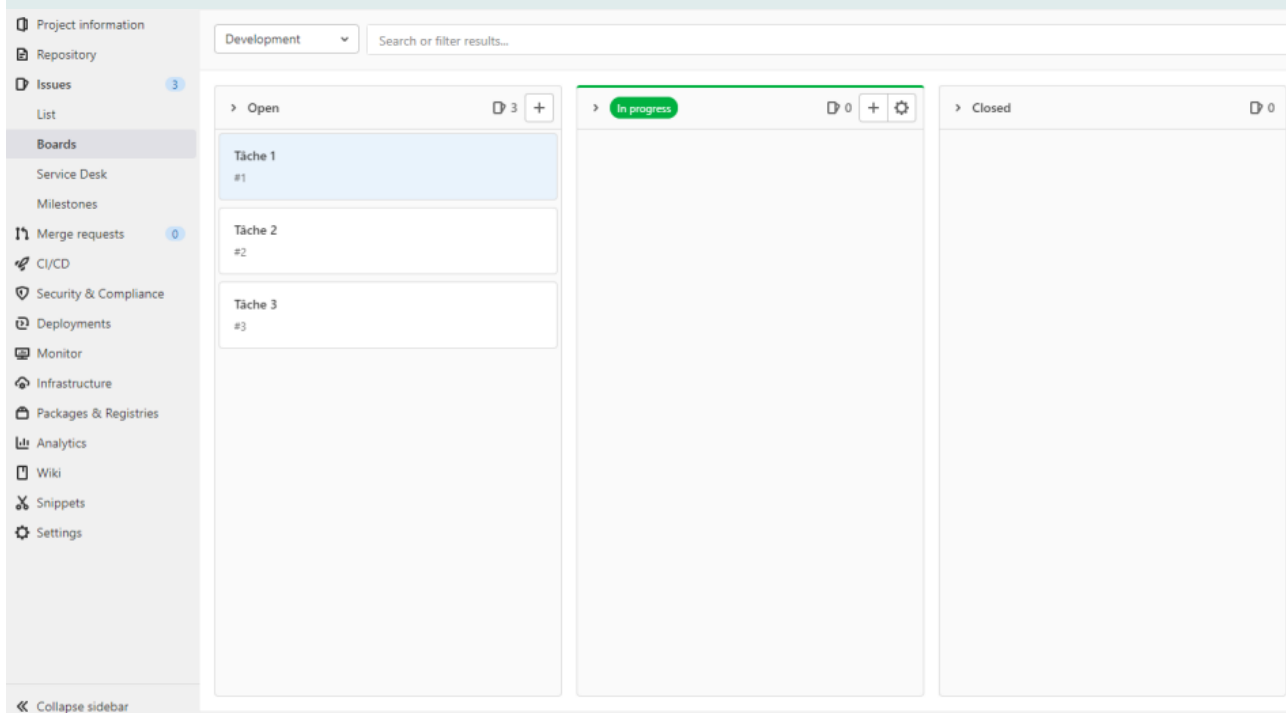
Fondamental Étape 1 : planification du code

Gitlab permet la création d'un backlog directement en ligne. Ce backlog est visible dans la partie Issues - Boards. Afin de rendre le board plus complet, il suffit de lui ajouter des labels.

Exemple : ajouter un label in progress - se rendre dans Project information - Labels

¹ https://gitlab.com/users/sign_in?__cf_chl_jschl_tk__=LX1N107510M510K7aDvE1251Ea1EnVIMAg0nV1F1055106220-0-ganyGzNBtE

Une fois le label créé, retourner dans Issues - Boards et cliquer sur « *Create list* » à droite de l'écran - sélectionner le label et cliquer sur « *Add to board* » votre backlog devrait maintenant ressembler à ça (sans les tâches).



Pour ajouter des tâches, il vous suffit de cliquer sur le bouton + en haut de la colonne Open. C'est grâce à ce type d'outil que l'intégration continue commence, elle permet à chacun de voir rapidement les tâches en attentes, en cours ou finies, on peut assigner certaines tâches à certaines personnes et être notifié des avancées de l'équipe.

Fondamental Étape 2 et 3 : la compilation, l'intégration, et le lancement automatique des tests unitaires

Pour utiliser GitLab CI / CD, vous avez besoin de :

- Un projet d'application hébergé dans un référentiel Git.
- Un fichier appelé `.gitlab-ci.yml`¹ à la racine de votre référentiel, qui contient la configuration CI / CD.

Dans le `.gitlab-ci.yml` vous pouvez définir :

- Les scripts que vous souhaitez exécuter automatiquement ou manuellement,
- Autres fichiers de configuration et modèles que vous souhaitez inclure,
- Dépendances et caches,
- L'emplacement vers lequel déployer votre application.

Dans un premier temps, il faut définir les différents stages du pipeline de CI que l'on souhaite créer. Je vous propose un découpage dans un premier temps en deux stages : Build, Tests.

Ces stages sont lancés séquentiellement et sont composés de jobs. Une étape doit contenir au moins un job, ces derniers étant exécutés en parallèle par défaut.

¹ <https://docs.gitlab.com/ee/ci/yaml/index.html>

Exemple Fichier composé de deux étapes (stages)

Attention, ne pas faire de copier-coller du code suivant dans votre projet cela ne pourra pas fonctionner.

```
1
2 .gitlab-ci.yml
3 <code>
4 images: node:12.1.0
5
6 cache:
7   paths:
8     - node_modules/
9
10 stages:
11   - build
12   - test
13
14 build-job:
15   stage: build
16   script:
17     - echo "Début de l'étape de compilation"
18     - superScriptDeCompilation
19
20 test-job:
21   stage: test
22   script:
23     - echo "Début de l'étape des tests:"
24     - superScriptDeTest
25
26 <code>
```

Explication du contenu de notre fichier :

- *Image* spécifie l'image de l'environnement nécessaire à utiliser(ex : Docker) pour lancer votre pipeline, cette image dépend de votre application.
- *Cache* permet d'éviter de télécharger node_modules à chaque étape.
- *Stages* définit les étapes d'un pipeline (build,test,deploy, etc.).

Le fichier ci-dessus est composé de deux jobs s'exécutant au sein de deux étapes.

Le job build-job appartient à l'étape de build. Cette tâche déclenche deux scripts. Si l'un des scripts échoue, l'étape build retournera une erreur.

Lors de l'étape de test, le pipeline va exécuter les tests unitaires déjà présents au sein du projet appelé par la partie script. L'objectif de cette étape est de s'assurer de lancer les tests écrits par les développeurs. Si un seul de ces tests échoue, le pipeline s'arrête. Il faudra alors remonter le bug dans le backlog et le corriger.

Le pipeline s'exécute à chaque fois que des modifications sont transmises à n'importe quelle branche du projet. Pour voir le pipeline complet, il suffit de cliquer sur le sous-menu Pipelines dans le menu CI / CD.

Fondamental Étape 4 et 5 : mesurez la qualité du code et livrez des packages

Une fois encore tout se joue dans le fichier .gitlab-ci.yml. Ce fichier est un peu le chef d'orchestre de l'intégration continue.

```
1 <code>
2
3 stages:
4   - build
5   - test
6   - quality
```

```

7   - package
8
9
10  build-code:
11    stage: build
12    script:
13      - echo "Début de l'étape de compilation"
14      - superScriptDeCompilation
15
16  test-code:
17    stage: test
18    script:
19      - echo "Début de l'étape des tests:"
20      - superScriptDeTest
21
22  quality-code:
23    stage: quality
24    script:
25      - echo "Début de l'étape contrôle qualité:"
26      - superScriptDeContrôle
27    artifacts: // permet de retourner un fichier contenant les résultats
28      paths:
29        - codequality-results/
30
31  package-code:
32    stage: package
33    script:
34      - echo "Mise en paquet:"
35      - superScriptDeMiseEnPaquet
36
37 <code>

```

Artifacts : définit des dossiers et des fichiers à stocker au sein du pipeline afin de les utiliser dans d'autres jobs. Pour voir le résultat de l'analyse de code, il suffit de naviguer dans le job `code_quality_job`, puis de cliquer sur Browse. Vous aurez alors accès au dossier contenant le résultat de l'analyse de code, et pourrez naviguer au sein de ce fichier, afin de voir les améliorations à apporter au code.

Enfin le processus de packaging nous permettra de pouvoir déployer facilement le même code sur différents environnements. Il sert aussi à figer le code compilé dans un package immuable. De ce fait, nous pouvons facilement redéployer le même code compilé sur n'importe quel autre environnement. Cela assure que le code ne soit pas modifié entre deux environnements, et qu'un code testé soit déployé partout de la même façon.

Nous avons maintenant toutes les étapes nécessaires pour l'intégration continue. Comme prévu, notre code est compilé en continu, testé, analysé puis package, prêt à être déployé sur de nouveaux environnements. Bien entendu chacune de ses étapes doit être complétée avec soin en utilisant les scripts, les images et les outils nécessaires à leur réalisation.

Sachez qu'il existe d'autres outils reprenant les mêmes concepts. Le plus connu et le plus utilisé d'entre eux est Jenkins¹. Avec cet outil, vous pouvez implémenter toutes les étapes précédemment vues. De plus, Jenkins utilise maintenant un fichier de description comme GitLab, qui s'appelle Jenkinsfile.

¹ <https://www.jenkins.io/>

```
.gitlab-ci.yml
1  stages:
2    - test
3    - build
4    - deploy
5
6  test project:
7    stage: test
8    image: node:15
9    script:
10     - yarn install
11     - yarn test
12
```

Attention

Une mauvaise indentation provoquera une erreur. Respecter une indentation de deux espaces.

Exercice : Quiz

[solution n°2 p.20]

Question 1

Puis-je gérer tout mon code Gitlab avec les mêmes commandes que sur Github ?

- ☐ Vrai
- ☐ Faux

Question 2

Les issues sur Gitlab représentent les user stories ?

- ☐ Vrai
- ☐ Faux

Question 3

Le fichier permettant de gérer toute la configuration est le fichier .gitlab-ci.xml.

- ☐ Vrai
- ☐ Faux

Question 4

Vous êtes un nouveau développeur, le projet sur lequel vous devez intervenir contient le fichier .gitlab-ci.yml suivant.

```

1 <code>
2 build:
3   stage: build
4   script:
5     - echo "building..." >> ./result.txt
6
7
8 unit_test:
9   stage: test
10  script:
11    - ls
12    - cat result.txt
13    - echo "unit testing..." >> ./result.txt
14
15 more_test:
16   stage: test
17   script:
18     - ls
19     - cat result.txt
20     - echo "unit testing..." >> ./resultmore.txt
21 artifacts:
22   paths:
23     - resultmore.txt
24   expire_in: 1 week
25
26 deploy:
27   stage: deploy
28   script:
29     - ls
30     - cat result.txt
31
32 </code>

```

Combien d'étapes différentes contient ce pipeline CI Gitlab ?

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4

Question 5

Toujours dans le même contexte, dans quel fichier se trouvent les résultats de la tâche `more_test` ?

- ☐ artefacts
- ☐ echo "unit testing"
- ☐ result.txt
- ☐ resultmore.txt

Question 6

Que pourrait-on mettre en place pour améliorer ce pipeline ?

- ☐ Ajouter une étape de test
- ☐ Ajouter une étape de contrôle qualité du code
- ☐ Ajouter une étape de build
- ☐ Ajouter une étape de déploiement

V. Essentiel

L'intégration continue est la première étape dans la création d'un pipeline CI / CD. Cette première étape est un mélange de pratique DevOps et de méthode agile. Cette association permet au développeur de toujours disposer d'un code source déployable dans un référentiel donné. Le développement continu de petit morceau de code et le grand nombre de fusions dans le code existant, permet de mettre en évidence et de le résoudre plus facilement, plus fréquemment et plus rapidement les erreurs de programmation.

Exercice : Quiz

[solution n°3 p.22]

Question 1

Quelle affirmation est fausse concernant l'intégration continue ?

- ☐ Elle permet aux équipes de répondre aux exigences métier plus rapidement
- ☐ Elle améliore la qualité du code
- ☐ Elle intègre la phase de tests automatisés séparément du flux de déploiement
- ☐ Elle accroît la sécurité des applications

Question 2

L'étape du « *build* » correspond à l'exécution des tests automatisés ?

- ☐ Vrai
- ☐ Faux

Question 3

Quelle est la première étape de la mise en place de l'intégration continue ?

- ☐ Gestion du code
- ☐ Mesurer la qualité du code
- ☐ Planification du développement
- ☐ Gérer les livrables de l'application
- ☐ Tester

Question 4

En règle générale, à quel moment l'orchestrateur se charge de lancer les tests unitaires ?

- ☐ Avant l'étape de compilation
- ☐ Pendant l'étape de compilation
- ☐ À la suite de l'étape de compilation

Question 5

Quelle affirmation est fausse concernant le processus de packaging ?

- ☐ Il permet de pouvoir déployer facilement le même code sur différents environnements
- ☐ Il sert aussi à figer le code compilé dans un package immuable
- ☐ Il permet que le code soit modifié entre deux environnements

VII. Auto-évaluation

A. Exercice

Question

[solution n°4 p.23]

Cet exercice a pour but de démontrer l'importance de la mise en place d'un pipeline CI et de permettre la détection d'une erreur grâce à celui-ci.

Prérequis :

- Avoir un compte Gitlab (pensez à rentrer une carte de paiement afin d'activer le pipeline CI / CD - gratuit jusqu'à 400 pipelines CI / CD par minute donc gratuit pour notre utilisation et bien plus).
- Un éditeur de code ou bloc-notes.
- Utiliser un terminal (Linux terminal, Mac Terminal app, or Windows Git Bash¹).

L'objectif est de créer un dépôt Gitlab, à l'intérieur de ce dépôt sera poussé dans un premier temps un fichier nommé hello.py composé du code suivant.

```
1 <code>
2 # hello.py
3 print('Hello world.')
4 </code>
```

¹ <https://git-scm.com/download/win>

Dans un second temps il faudra créer ou modifier (si créé automatiquement par Gitlab) le fichier `.gitlab-ci.yml` et remplacer son code par le code suivant :

```
1 <code>
2 image: python:latest
3
4 job0:
5   stage: build
6   script:
7     - echo "build phase..."
8     - uname -a
9
10 job1:
11   stage: test
12   script:
13     - echo "test phase..."
14     - python helloworld.py
15
16 job2:
17   stage: deploy
18   script:
19     - echo "deploy phase..."
20 </code>
```

Suite à cette étape votre pipeline sera en status failed, à l'aide du pipeline CI / CD de Gitlab identifier l'erreur et la modifier pour obtenir le status « *passed* ».

B. Test

Exercice 1 : Quiz

[solution n°5 p.25]

Question 1

Que signifie l'acronyme CI que l'on retrouve dans l'approche CI / CD ?

- ☐ Continuous Delivery
- ☐ Continuous Integration
- ☐ Continuous Deployment
- ☐ Continuous Interpretation

Question 2

Vous êtes développeur et vous venez de terminer une modification à apporter sur le projet. Une fois vos modifications poussées dans le dépôt de code, quelle est la prochaine étape afin de respecter un processus de CI ?

- ☐ Déployé le code en production
- ☐ Tester le code
- ☐ Mesurer la qualité du code
- ☐ Planifier un nouveau développement

Question 3

Plus un projet est grand moins l'intégration continue est efficace ?

- ☐ Vrai
- ☐ Faux

Question 4

L'intégration continue permet-elle d'avoir en permanence une version fonctionnelle de l'application ?

- ☐ Vrai
- ☐ Faux

Question 5


Les développeurs doivent-ils être capables de gérer manuellement toutes les étapes de CI ?

- ☐ Vrai
- ☐ Faux

Solutions des exercices


Exercice p. 7 Solution n°1**Question 1**

Quel est le terme employé par le Product Owner pour définir le répertoire des tâches ?

- ☐ Le frontlog
- ☒ Le backlog
- ☐ Le front-end
- ☐ Le back-end
-  Dans un système agile, on utilise bien le backlog.


Question 2

Quel outil va permettre de gérer les étapes 2 à 5 composant notre pipeline de CI ?

- ☐ La compilation
- ☐ Le package
- ☐ Le backlog
- ☒ L'orchestrateur
-  Le processus de build (construction) est automatisé grâce à l'orchestrateur.


Question 3

Quelle première étape du build permet à l'orchestrateur de garantir un code utilisable à n'importe quel développeur souhaitant travailler sur le projet ?

- ☐ Le déploiement
- ☐ Le test
- ☒ La compilation
- ☐ Le contrôle qualité
-  La compilation, associée ensuite à l'intégration du code.

Question 4

Un orchestrateur peut-il gérer un backlog ?


- ☒ Vrai
- ☐ Faux
-  Ce n'est pas sa fonction numéro un, mais des services comme Gitlab permettent d'avoir un backlog connecté à notre processus d'intégration continue, chacune des tâches dans le backlog seront automatiquement reliées à des actions faites par les développeurs. Ainsi on pourra suivre le cycle de vie d'une tâche de sa création dans le backlog jusqu'à sa livraison.

Question 5

Un artefact peut-il être passé en production ?

☒ Vrai

☐ Faux


 Un artefact nous permet de conserver des éléments entre différentes étapes du pipeline.

Question 6

La qualité du code doit-elle être revue par un développeur senior seulement ?

☐ Vrai

☒ Faux

 Il serait impossible sur certain projet de se passer d'un contrôle réalisé par des outils informatiques.


Exercice p. 13 Solution n°2

Question 1

Puis-je gérer tout mon code Gitlab avec les mêmes commandes que sur Github ?

☒ Vrai

☐ Faux


 GitLab est un logiciel libre de gestion du code basé sur git.

Question 2

Les issues sur Gitlab représentent les user stories ?

☒ Vrai

☐ Faux

 Elles permettent de découper les tâches d'un projet en une multitude de petites tâches.

Question 3

Le fichier permettant de gérer toute la configuration est le fichier .gitlab-ci.xml.

☐ Vrai

☒ Faux

 C'est un fichier YAML .yaml.


Question 4

Vous êtes un nouveau développeur, le projet sur lequel vous devez intervenir contient le fichier .gitlab-ci.yml suivant.

```
1 <code>
2 build:
3   stage: build
4   script:
5     - echo "building..." >> ./result.txt
6
7
8 unit_test:
9   stage: test
10  script:
11    - ls
12    - cat result.txt
13    - echo "unit testing..." >> ./result.txt
14
15 more_test:
16   stage: test
17   script:
18     - ls
19     - cat result.txt
20     - echo "unit testing..." >> ./resultmore.txt
21 artifacts:
22   paths:
23     - resultmore.txt
24   expire_in: 1 week
25
26 deploy:
27   stage: deploy
28   script:
29     - ls
30     - cat result.txt
31
32 </code>
```

Combien d'étapes différentes contient ce pipeline CI Gitlab ?


- ☐ 1
- ☐ 2
- ☒ 3
- ☐ 4

 Build, test et deploy. L'étape test est composée de deux jobs différents, mais qui appartiennent à la même étape.

Question 5


Toujours dans le même contexte, dans quel fichier se trouvent les résultats de la tâche more_test ?

- ☐ artefacts
- ☐ echo "unit testing"
- ☐ result.txt
- ☒ resultmore.txt

 Il faut aller dans le fichier resultmore.txt.

Question 6


Que pourrait-on mettre en place pour améliorer ce pipeline ?

- ☐ Ajouter une étape de test
- ☒ Ajouter une étape de contrôle qualité du code
- ☐ Ajouter une étape de build
- ☐ Ajouter une étape de déploiement
-  On pourrait ajouter une étape de contrôle qualité du code pour améliorer ce pipeline.

Exercice p. 15 Solution n°3


Question 1

Quelle affirmation est fausse concernant l'intégration continue ?

- ☐ Elle permet aux équipes de répondre aux exigences métier plus rapidement
- ☐ Elle améliore la qualité du code
- ☒ Elle intègre la phase de tests automatisés séparément du flux de déploiement
- ☐ Elle accroît la sécurité des applications
-  Avec l'intégration continue la phase de tests automatisés est complètement intégrée au flux de déploiement.


Question 2

L'étape du « *build* » correspond à l'exécution des tests automatisés ?

- ☐ Vrai
- ☒ Faux
-  Le « *build* » est la création automatique de l'application.


Question 3

Quelle est la première étape de la mise en place de l'intégration continue ?

- ☐ Gestion du code
- ☐ Mesurer la qualité du code
- ☒ Planification du développement
- ☐ Gérer les livrables de l'application
- ☐ Tester
-  La planification du développement, bien évidemment comme dans bien d'autres situations, on commence par planifier.


Question 4

En règle générale, à quel moment l'orchestrateur se charge de lancer les tests unitaires ?

- ☐ Avant l'étape de compilation
- ☐ Pendant l'étape de compilation
- ☒ À la suite de l'étape de compilation
-  En règle générale l'orchestrateur se charge de lancer les tests unitaires à la suite de la compilation. Ils doivent s'exécuter de la manière la plus rapide possible, afin d'avoir le retour d'information le plus rapide lui aussi.

Question 5

Quelle affirmation est fausse concernant le processus de packaging ?

- ☐ Il permet de pouvoir déployer facilement le même code sur différents environnements
- ☐ Il sert aussi à figer le code compilé dans un package immuable
- ☒ Il permet que le code soit modifié entre deux environnements
-  Cela assure que le code ne soit pas modifié entre deux environnements, et qu'un code testé soit déployé partout de la même façon.

p. 16 Solution n°4

Créer un projet/repo sur Gitlab (Blank project). Sur son ordinateur, utiliser un terminal de commande (ici git bash).

```
git clone https://gitlab.com/votreUserGitlab/ci_python.git
```

```
cd ci_python
```

```
touch hello.py //Créer un fichier
```

```
code hello.py // Ouvrir le fichier avec Visual studio code peut être remplacé par  
clic droit ouvrir avec sur le fichier
```

Coller dans le fichier hello.py le code suivant :

```
1 <code>  
2 # hello.py  
3 print('Hello world.')  
4 </code>
```

```
git add *
```

```
git commit -m "first commit"
```

```
git push
```

Ouvrir .gitlab-ci.yml

Remplacer le code par :

```
1 <code>  
2 image: python:latest  
3  
4 job0:  
5   stage: build  
6   script:  
7     - echo "build phase..."  
8     - uname -a  
9
```


```
10 job1:
11   stage: test
12   script:
13     - echo "test phase..."
14     - python helloworld.py
15
16 job2:
17   stage: deploy
18   script:
19     - echo "deploy phase..."
20 </code>
```

```
git add .gitlab-ci.yml
```

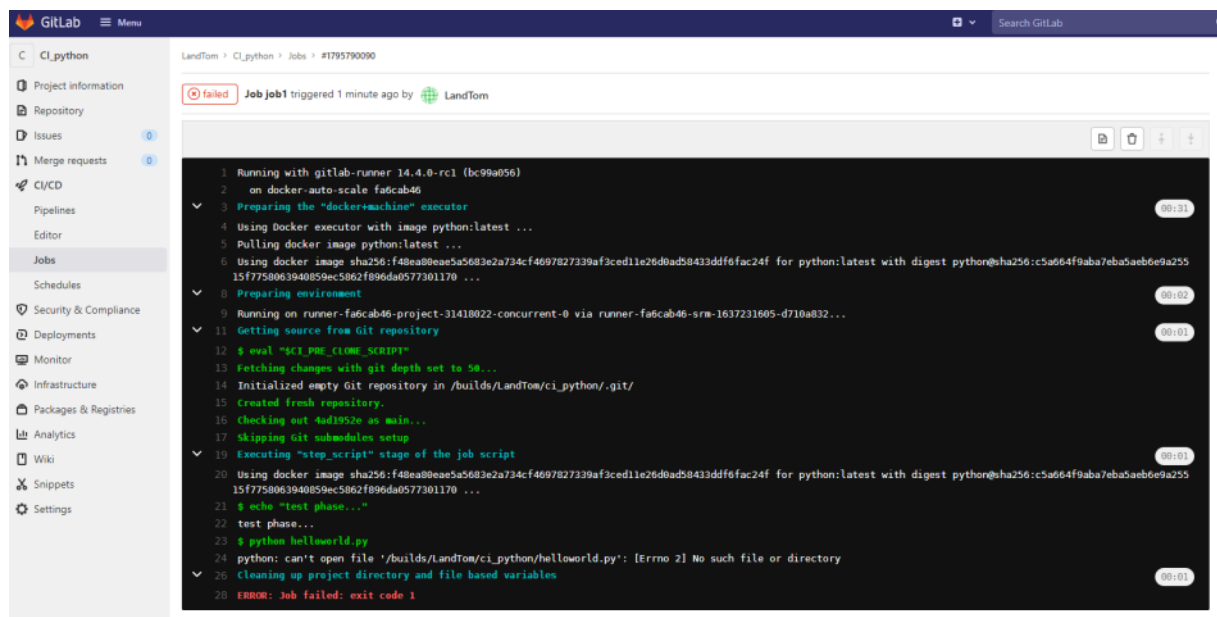
```
git commit -m "commit gitlab-ci"
```

```
git push
```

À ce stade, le pipeline CI / CD devrait ressembler à ça :

Status	Pipeline ID	Triggerer	Commit	Stages	Duration
	#411427832 latest		main → 4ad1952e  commit gitlab-ci	  	00:01:15 7 minutes ago

Cliquer sur « failed » puis sur « job1 » pour arriver sur la fenêtre suivante :



```
1 Running with gitlab-runner 14.4.0-rc1 (bc99a056)
2 on docker-auto-scale fa6cab46
3 Preparing the "docker+machine" executor
4 Using Docker executor with image python:latest ...
5 Pulling docker image python:latest ...
6 Using docker image sha256:f48ea08eae5a5683e2a734cf4697827339af3ced11e26d8ad58433ddf6fac24f for python:latest with digest python@sha256:c5a664f9aba7eba5aeb6e9a25515f7758063940859ec5862f896da0577301170 ...
7 Preparing environment
8 Running on runner-fa6cab46-project-31418022-concurrent-0 via runner-fa6cab46-srm-1637231605-d710a832...
9 Getting source from Git repository
10 $ eval "$CI_PSE_CLONE_SCRIPT"
11 Fetching changes with git depth set to 50...
12 Initialized empty Git repository in /builds/LandTom/ci_python/.git/
13 Created fresh repository.
14 Checking out 4ad1952e as main...
15 Skipping Git submodules setup
16 Executing "step_script" stage of the job script
17 Using docker image sha256:f48ea08eae5a5683e2a734cf4697827339af3ced11e26d8ad58433ddf6fac24f for python:latest with digest python@sha256:c5a664f9aba7eba5aeb6e9a25515f7758063940859ec5862f896da0577301170 ...
18 $ echo "test phase..."
19 test phase...
20 $ python helloworld.py
21 python: can't open file '/builds/LandTom/ci_python/helloworld.py': [Errno 2] No such file or directory
22 Cleaning up project directory and file based variables
23 ERROR: Job failed: exit code 1
```

Identifier l'erreur dans la console : python: can't open file '/builds/USER/ci_python/helloworld.py': [Errno 2] No such file or directory

L'erreur provient du naming de notre premier fichier.









Corrigez l'erreur :

```
git mv hello.py helloworld.py
```

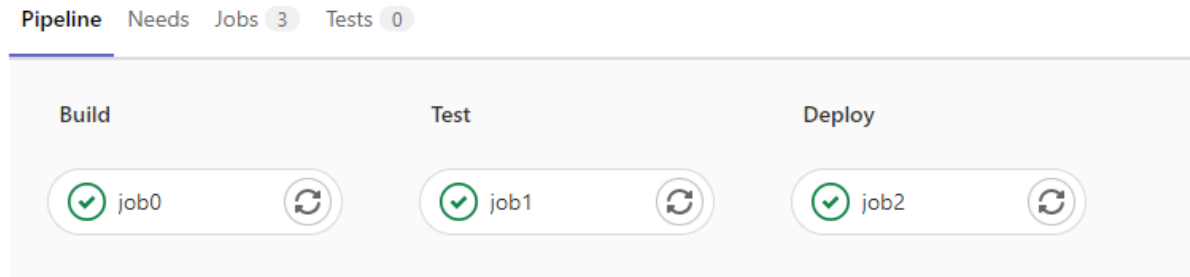
```
git add *
```

```
git commit -m "change name file hello.py to helloworld.py"
```

```
git push
```


	#411438752 latest		P main -> 0da9aed5 change name file hello.py		00:01:57 1 hour ago	
	#411427832		P main -> 4ad1952e commit gitlab-ci		00:01:15 1 hour ago	

Si on retourne sur passed - job1



```

17 Skipping git submodules setup
19 Executing "step_script" stage of the job script
20 Using docker image sha256:f48ea80eae5a5683e2a734cf4697827339af3ced1
20 15f7758063940859ec5862f896da0577301170 ...
21 $ echo "test phase..."
22 test phase...
23 $ python helloworld.py
24 Hello world.
26 Cleaning up project directory and file based variables
28 Job succeeded
  
```

On remarque que l'étape test, qui dans notre cas ouvre simplement le fichier helloworld.py a bien été exécuté et le script `print('Hello world.')` a bien été joué.

Ce léger cas pratique consistait à vous faire mettre le doigt dans l'engrenage du pipeline CI / CD sujet vaste et demandant une expertise sur de nombreux sujets. Sur les projets de grande envergure il est même tout à fait courant que chacune des étapes CI soit gérée par un service différent.

Exercice p. 17 Solution n°5

Question 1

Que signifie l'acronyme CI que l'on retrouve dans l'approche CI / CD ?

- ☐ Continuous Delivery
- ☒ Continuous Integration
- ☐ Continuous Deployment
- ☐ Continuous Interpretation

🔍 « CI » dans CI/CD désigne « *Continuous Integration* » (l'Intégration Continue) alors que « CD » dans CI / CD désigne la « *Distribution Continue* » et / ou le « *déploiement continu* »

Question 2


Vous êtes développeur et vous venez de terminer une modification à apporter sur le projet. Une fois vos modifications poussées dans le dépôt de code, quelle est la prochaine étape afin de respecter un processus de CI ?

☐ Déployé le code en production

☒ Tester le code

☐ Mesurer la qualité du code

☐ Planifier un nouveau développement

 Lorsque le développeur développe une fonctionnalité de l'application, il développe aussi le test qui permet de tester sa fonctionnalité puis la dépose dans le dépôt de code. Le serveur d'intégration va ensuite faire tourner tous les tests pour vérifier qu'aucune régression n'a été fusionnée dans le code source suite à cet ajout.

Question 3

Plus un projet est grand moins l'intégration continue est efficace ?

☐ Vrai

☒ Faux


 Plus un projet est grand plus la mise en place de méthode CI est un facteur important de la viabilité du projet.

Question 4

L'intégration continue permet-elle d'avoir en permanence une version fonctionnelle de l'application ?

☒ Vrai

☐ Faux


 Même si cela ne garantira jamais une absence de bug à 100 %.

Question 5

Les développeurs doivent-ils être capables de gérer manuellement toutes les étapes de CI ?

☐ Vrai

☒ Faux

 Une grande partie de ses étapes seront accomplies automatiquement par des outils, mais il est important de bien comprendre les concepts cachés derrière l'utilisation de ces outils.