

Mettre en place l'intégration et la livraison continue avec la Démarche DevOps

mardi 20 septembre 2022 11:04

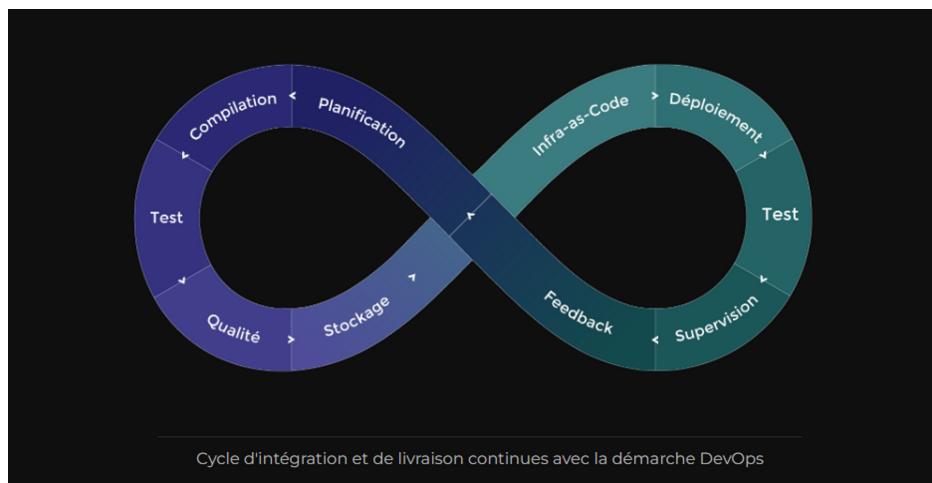
Lien du cours : <https://openclassrooms.com/fr/courses/2035736-mettez-en-place-l-integration-et-la-livraison-continues-avec-la-demarche-devops/6182691-qu'est-ce-que-l-integration-continue>

L'intégration continue et la livraison continue sont de plus en plus prisées en entreprise, surtout dans le cadre de la **démarche DevOps**. C'est parce que ces méthodes de développement et de déploiement de code présentent de nombreux avantages : **fluidité du travail entre les dev et les ops, rapidité d'intégration, flexibilité, rapidité d'itération...**

Qu'est-ce que l'intégration continue ?

L'**intégration et la livraison continues**, ou en anglais **Continuous Integration and Continuous Delivery (CI/CD)** permettent de :

- accélérer le **Time-to-Market** (le temps de développement et de mise en production d'une fonctionnalité)
- réduire les erreurs lors des livraisons
- assurer une continuité de service des applications.



Tout au long de ce cours, nous utiliserons **GitLab CI**, un outil très performant et permettant de mettre en place chacune des étapes du cycle avec un seul et même outil.

Qu'est-ce que l'intégration continue ?

Dans ce chapitre, nous allons introduire et expliquer ce qu'est l'**intégration continue**, et quelles sont les **différentes étapes** à mettre en œuvre lors de la mise en place de celle-ci.

qu'est ce que l'intégration continue ? En quoi diffère-t-elle de la livraison continue ?

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.

Dans une entreprise, il est courant qu'il y ait plusieurs développeurs travaillant sur la même application, et même sur le même code. Avant l'apparition de Git, comme contrôle de code source distribué, il existait d'autres outils comme SVN ou CVS, mais le dépôt était centralisé, ce qui menait à de nombreux problèmes d'intégration du code.

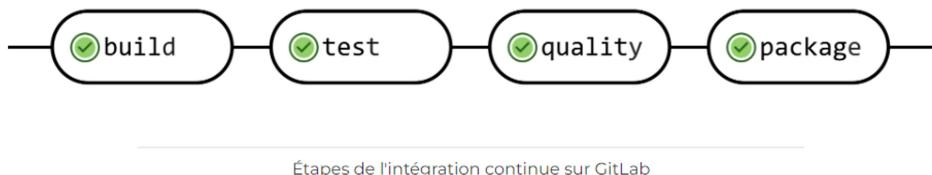
Le principe de l'intégration continue est justement de détecter ces problèmes d'intégration au plus tôt dans le cycle de développement.

Afin de détecter ces problèmes, il est nécessaire de passer par plusieurs étapes que nous allons détailler au fur et à mesure, et afin d'illustrer ces étapes, je vais comparer la CI/CD à une chaîne d'assemblage automobile.

L'intégration continue va se faire en 5 étapes :

- Planifiez votre développement.
- Compilez et intégrez votre code.
- Testez votre code.
- Mesurez la qualité de votre code.
- Gérez les livrables de votre application.

Toutes les étapes se feront sur **GitLab**. Les étapes 2 à 4 seront lancées automatiquement grâce à **GitLab CI**



Étape 1 : Planifiez votre développement

Afin de savoir quoi développer, il est nécessaire d'avoir à disposition un outil permettant la collaboration entre les développeurs. Cet outil permettra notamment de gérer les différentes releases et toutes les fonctionnalités, de garantir la priorité du backlog, etc.

Intervenant tout au long du projet, la collaboration de toute l'équipe est nécessaire pour assurer la planification du projet. Cette planification est étroitement liée à la méthodologie Scrum. Elle a pour but de découper le projet en petites tâches à réaliser par toute l'équipe.

Sur notre chaîne d'assemblage, la planification représente la répartition des tâches de chaque collaborateur, ou encore le brief du matin destiné à définir les tâches à faire durant la journée.

Pour collaborer avec vos équipes, vous pourrez utiliser Jira, GitLab, Confluence, ALM Octane ou encore Pivotal Tracker.

Étape 2 : Compilez et intégrez votre code

- Le contrôle de code source

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. Même s'il existe des **notions de branche**, la création d'une branche doit être évitée le plus possible, privilégiant le **développement sur la branche principale** ; cela évite de maintenir plusieurs versions en parallèle. Ce genre de pratique est appelé **trunk-based development**.

Pour faire du contrôle de source, vous retrouverez les outils **Git**, **Subversion**, **GitHub**, **GitLab**, **Perforce** ou bien **Bitbucket**.

- L'orchestrateur

Ensuite, **toutes les étapes doivent être automatisées par un orchestrateur**, qui saura **reproduire ces étapes** et gérer les dépendances entre elles. De plus, l'utilisation d'un orchestrateur permet de donner **accès à tous**, et à tout moment, à **un tableau de bord** qui donnera l'**état de santé des étapes d'intégration continue**. Ainsi, les développeurs ont au plus tôt la **boucle de feedback** nécessaire, afin de garantir que l'**application soit prête à tout moment**. De plus, l'**orchestrateur permettra d'aller plus loin dans la livraison continue**, comme on le verra dans la suite du cours.

L'orchestrateur est le tapis roulant de notre chaîne d'assemblage, qui va guider toutes les pièces de la voiture petit à petit, afin d'arriver au résultat final.

La **première étape** est de **compiler le code de manière continue**. En effet, sans cette étape, le code est compilé **manuellement** sur le poste du développeur, afin que ce dernier s'assure que son code compile.

Si nous comparons cette étape à notre chaîne d'assemblage automobile, c'est ici que l'on assemble toutes les pièces de la voiture. Malheureusement, comme dit précédemment, le développeur ne s'assure pas que son code permet de bien compiler avec tous les autres développements faits par l'équipe. À la prochaine livraison, un développeur intègre alors manuellement toutes les modifications, une opération qui produit beaucoup de peine et de souffrance.

La **mise en place d'une première étape de compilation dans un processus d'intégration continue** permet justement de **ne plus se soucier si des modifications de code cassent la compilation**. Le développeur doit alors s'assurer de **bien envoyer son code source sur le dépôt central**. En faisant cela, il déclenche une **première étape de compilation, avec toutes les modifications des autres développeurs**. Si la **compilation ne se fait pas**, le code est alors **rejeté**, et le **développeur doit corriger ses erreurs**.

Après cette première étape, le **code devient plus sûr**, et le **dépôt de code source garantit qu'à chaque instant, un développeur récupère un code qui compile**. Dans cette étape, les tests ne sont pas encore exécutés. Le code peut donc être de mauvaise qualité.

Vous pourrez compiler votre code avec **Maven**, **Ant**, **Gradle**, **MSBuild**, **NAnt**, **Gulp** ou encore **Grunt**.

Étape 3 : Testez votre code

Dans cette étape, l'**orchestrateur se charge de lancer les tests unitaires à la suite de la compilation**. Ces tests unitaires, généralement avec un **framework associé**, garantissent que le **code respecte un certain niveau de qualité**.

Les tests unitaires permettent de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme.

Plus il y a de tests unitaires, plus le code est garanti sûr. Évidemment, l'**orchestrateur ne peut lancer que les tests qui ont été codés par les développeurs**, et ne peut pas inventer de nouveaux cas de tests.

Ces tests doivent s'exécuter de la manière la **plus rapide possible**, afin d'avoir **un feedback 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.

Les **tests unitaires** apportent 3 atouts à la production :

- **trouver les erreurs plus facilement.** Les tests sont exécutés durant tout le développement, permettant de visualiser si le code fraîchement écrit correspond au besoin
- **sécuriser la maintenance.** Lors d'une **modification** d'un programme, les tests unitaires signalent les éventuelles régressions. En effet, certains tests peuvent échouer à la suite d'une modification, il faut donc soit réécrire le test pour le faire correspondre aux nouvelles attentes, soit corriger l'erreur se situant dans le code
- **documenter le code.** Les tests unitaires peuvent servir de **complément à la documentation** ; il est très utile de lire les tests pour comprendre comment s'utilise une méthode. De plus, il est possible que la documentation ne soit plus à jour, mais les tests, eux, correspondent à la réalité de l'application.

L'ensemble des tests unitaires doivent être relancés après une modification du code, afin de vérifier qu'il n'y ait pas de régressions (l'apparition de nouveaux dysfonctionnements).

La multiplicité des test unitaires oblige à les **maintenir** dans le temps, **au fur et à mesure que le développement avance**.

Pour implémenter et exécuter vos tests unitaires, vous retrouverez des outils comme **JUnit**, **NUnit** ou encore **XUnit**.

Toujours sur notre chaîne d'assemblage, les tests constituent l'assurance qualité de la voiture, à chaque étape d'assemblage de celle-ci.

Étape 4 : Mesurez la qualité de votre code

Maintenant que les tests unitaires sont écrits et exécutés, nous commençons à avoir une meilleure qualité de code, et à être rassurés sur la fiabilité et la robustesse de l'application. Grâce à la compilation et aux tests unitaires, nous pouvons maintenant **mesurer la qualité du code**. Tout ceci permet aux développeurs de **maintenir dans le temps un code de très bonne qualité**, alertant l'équipe en cas de dérive des bonnes pratiques de tests.

L'étape de qualité de code est **différente** de l'étape de test, car cette **étape de qualité** assure que le **code sera maintenable et évolutif au fur et à mesure de son cycle de vie**, alors que les tests servent à garantir que le code **implémente bien les fonctionnalités demandées, et ne contient pas (ou peu) de bugs**.

Lors de l'étape de qualité de code, nous cherchons à assurer la plus petite **dette technique** possible de notre application :

La dette technique est le temps nécessaire à la correction de bugs ou à l'ajout de nouvelles fonctionnalités, lorsque nous ne respectons pas les règles de coding. La dette est exprimée en heures de correction. Plus cette dette est élevée, plus le code sera difficile à maintenir et à faire évoluer.

L'étape de **qualité de code** mesure aussi d'**autres métriques**, comme le **nombre de vulnérabilités** au sein du code, la **couverture de test**, mais aussi les **code smells** (qui sont des mauvaises pratiques à ne pas implémenter), la **complexité cyclomatique (complexité du code applicatif)** ou la **duplication de code**. C'est le rôle du développeur de respecter les normes définies et de corriger au fur et à mesure son code.

Afin de **renforcer la qualité du code** et de **ne pas autoriser le déploiement d'un code de mauvaise qualité**, nous pouvons implémenter un **arrêt complet du pipeline d'intégration continue**, si le code n'atteint pas la qualité requise.

Les outils : la qualité du code peut être évaluée grâce à **SonarQube**, **Cast** ou **GitLab Code Quality**.

La qualité de code constitue le banc d'essai de notre chaîne d'assemblage, où l'on teste différents paramètres comme la qualité du freinage, le moteur, la résistance aux chocs, etc.

Étape 5 : Gérez les livrables de votre application

Le code, une fois compilé, doit être **déployé dans un dépôt de livrables**, et versionné. Les binaires produits sont appelés **artefacts**. Ces **artefacts** doivent être accessibles à toutes les parties prenantes de l'application, afin de pouvoir **les déployer et lancer les tests** autres qu'unitaires (**test de performance, test de bout en bout, etc.**). Ces **artefacts** sont disponibles dans un **stockage, centralisé et organisé, de données**. Ce peut être **une ou plusieurs bases de données** où les **artefacts** sont localisés en vue de leur distribution sur le réseau, ou bien un endroit directement accessible aux utilisateurs.

La mise à disposition des artefacts peut être faite par **Nexus**, **Artifactory**, **GitLab repository**, **Quay**, **Docker Hub**.

Sur notre chaîne d'assemblage, la gestion des livrables est l'entrepôt des voitures finies. Il est tout à fait possible de prendre une de ces voitures afin de lancer des tests complémentaires, ou alors de la vendre, ce qui correspondrait à la mise en production.

Utilisez GitLab pour mettre en place un pipeline CI/CD

Dans la suite de ce cours, nous ferons le choix de GitLab. Cet outil a l'avantage d'avoir toutes les briques nécessaires à la mise en place de l'intégration continue, sans rentrer dans des étapes complexes de mise en place des outils, ainsi que les connexions associées.

Afin de pouvoir illustrer l'intégration continue, nous allons travailler sur un projet open source : la mise en place du site web d'une clinique vétérinaire via Spring Boot. Toutes les étapes de mise en place de l'intégration continue seront entièrement détaillées dans la suite de cette partie.

En résumé :

voici les grandes étapes de l'**intégration continue** :

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

Planifier votre développement

Nous allons nous servir de **GitLab** comme **outil de planification de nos développements**. La méthodologie DevOps veut que la planification de tout le **backlog** de l'application et des **fonctionnalités** à développer soit **effectuée en amont**, et **suivie tout au long de la chaîne de CI/CD**.

Dans GitLab, l'issue est la notion centrale pour définir n'importe quoi : que ce soit une epic, une feature ou même un bug. La distinction se fait via des labels, que l'on positionne sur l'issue

Créez votre projet GitLab

Dans un premier temps, nous allons créer un projet dans GitLab. Pour cela, nous allons utiliser la plateforme <https://gitlab.com/>, afin d'éviter d'installer notre propre plateforme GitLab en local.

Une fois connecté, vous arrivez sur une page listant tous vos projets. Si vous venez de créer le compte, normalement cette page sera vide.

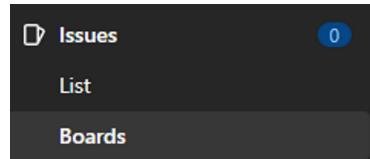
Comme expliqué dans le premier chapitre, nous allons nous baser sur le projet PetClinic tout au long du cours. Dans ce chapitre, nous allons commencer à planifier le développement de fonctionnalités pour ce projet.

Pour **créer un nouveau projet** dans GitLab, une fois connecté, cliquez sur le bouton New Project. Sur la nouvelle page, il faut entrer un nouveau nom de projet. Dans le champ Project Name, nous allons entrer le nom **spring-petclinic-microservices**. Notez au passage que GitLab remplit automatiquement le champ **Project Slug**. Il est **impératif de passer le projet en Public** afin de bénéficier de toutes les fonctionnalités abordées dans ce cours. Une fois tous les champs remplis, l'écran devrait ressembler à cela :

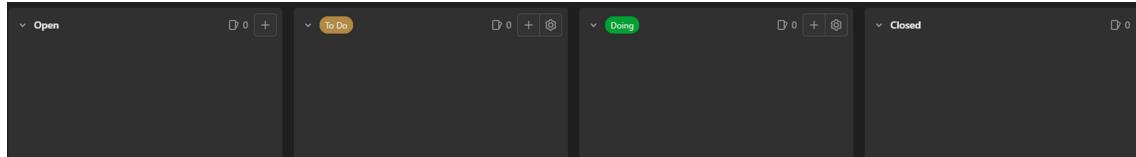
Création d'un projet sur GitLab

Appuyez maintenant sur le bouton **Create Project**. Vous arrivez alors sur la page du projet fraîchement créé.

Avant de récupérer le code source de l'application, et de l'intégrer dans GitLab, **nous allons mettre en place les différentes briques nécessaires au bon déroulement du projet**. Tout d'abord, nous allons naviguer dans la partie Issues, afin d'**ajouter les colonnes nécessaires à notre projet**. Pour ce faire, il suffit d'aller sur le menu à gauche, survoler le menu Issues, et cliquer sur le lien Boards.

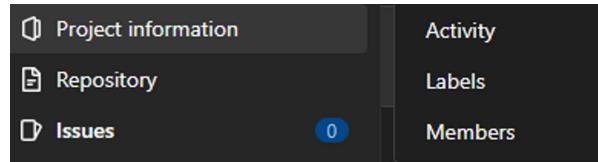


Sur cette page, GitLab nous propose d'ajouter les listes par défaut via le bouton **Add default lists**. GitLab crée alors deux nouvelles colonnes, To Do et Doing :



Personnellement, j'ai du créer les label à la main avant de pouvoir créer les colonnes

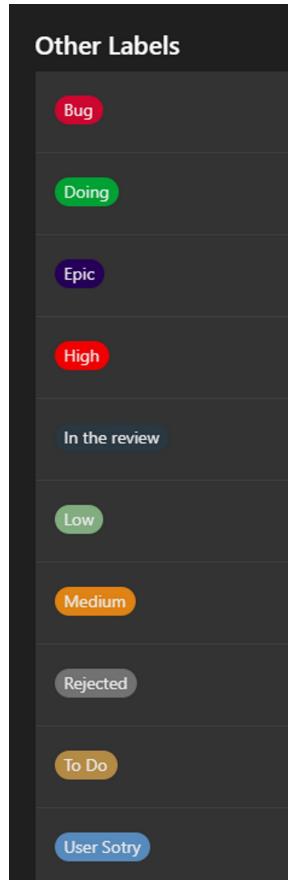
Ce board sera notre **board par défaut durant tout notre projet**, afin de voir son avancement. Nous allons ensuite créer les **différents labels** que nous avons vus précédemment, afin de pouvoir créer et catégoriser les issues que nous allons ouvrir. Pour ce faire, nous allons aller dans le sous-menu Labels du menu Issues. Sur la nouvelle page, il faut alors cliquer sur le **bouton New Label** pour créer une nouvelle catégorie d'issue. *Notez que GitLab a déjà créé les deux labels To Do et Doing pour nous.*



Sur la page de création de nouveaux labels, nous allons ajouter tous les labels nécessaires à la catégorisation des issues :

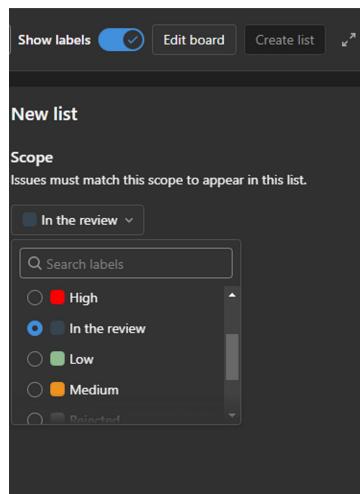
The screenshot shows the 'New Label' creation form. It includes fields for 'Title' (a blue input field), 'Description' (an empty input field), and 'Background color' (a color picker set to #428BCA). Below the color picker is a message: 'Choose any color. Or you can choose one of the suggested colors below.' A horizontal color palette with various hex codes is displayed. At the bottom left is a 'Create label' button, and at the bottom right is a 'Cancel' button. The status bar at the bottom says 'Création d'un nouveau label'.

Les catégories à créer sont : **Epic, User Story, Bug, Ready, Rejected, High, Medium, Low, In Review**. Une fois les labels créés, la liste devrait ressembler à ceci :

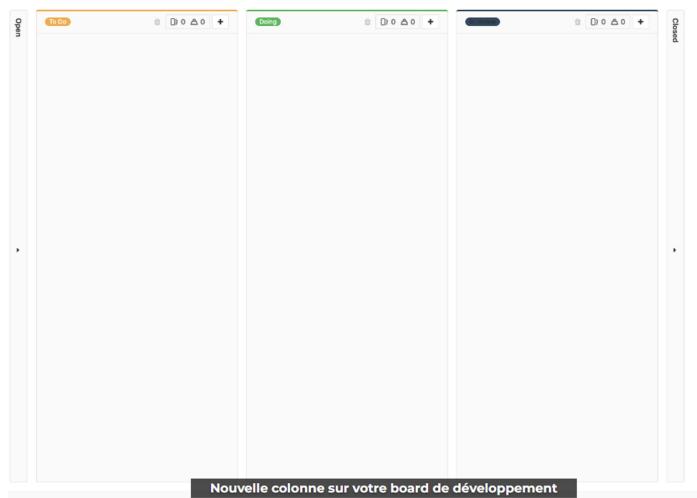


Vous noterez la **création de plusieurs labels** que je n'ai pas détaillés : **High, Medium, Low, Ready, Rejected et In Review**. Ces **labels** sont utilisés pour **catégoriser plusieurs issues** et pour **créer un nouveau board**, que nous allons appeler **Product Backlog**.

Pour ce faire, il faut **retourner dans le menu Board**. Tout d'abord, dans la page courante, cliquez sur Add List et sélectionnez le label In Review pour ajouter une nouvelle colonne dans le board courant.

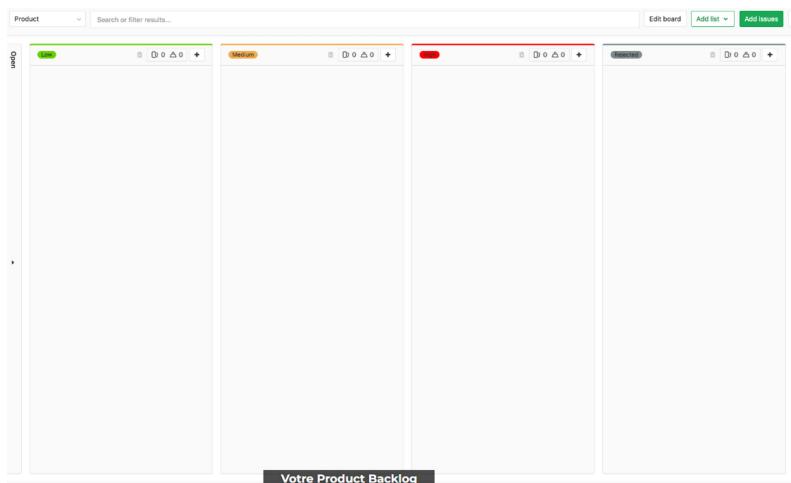


Le nouveau board doit ressembler à cela (pour des questions de lisibilité, j'ai replié les colonnes Open et Closed dans la capture d'écran suivante) :



Ensuite, sur la page, en haut à gauche, il y a une liste déroulante contenant tous les boards créés. Pour l'instant, il n'y en a qu'un seul, celui de développement qui est actuellement affiché.

Dans cette liste déroulante, sélectionnez **Create New Board**, puis donnez-lui le nom de **Product**. Ce board sera notre **Product Backlog** qui listera toutes les **epics**, ainsi que les **user stories** associées et leurs états respectifs :



Nous avons maintenant tous les outils nécessaires afin de commencer notre travail sur le projet. Le **workflow de développement** se déroule comme suit :

- Le **product owner** travaille avec les **utilisateurs finaux** afin de définir un **Product Backlog**. Ce **Product Backlog** est constitué de **différentes epics**.
- Ces **epics** doivent **faire apparaître plusieurs critères**, comme par exemple des **wireframes** ou autres écrans définissant l'application. Une **epic** en **high** passe en **sprint backlog** lorsque son périmètre est délimité avec un label **Ready**, et le développement est confirmé avec le client. Ce n'est qu'à ce moment que l'**epic** est discutée avec l'équipe et **décomposée en user stories**, décrivant la **feature** à développer.
- Les **user stories** doivent respecter la **Definition of Ready** définie plus tôt, lors du cycle de développement. Une **user story** n'est **ready** que si elle a bien été écrite, et **contient tous les éléments** nécessaires à son **développement** durant le **sprint**.
- Une fois que le **sprint backlog** a été **décomposé en user stories**, nous pouvons alors **commencer la planification du sprint backlog**.

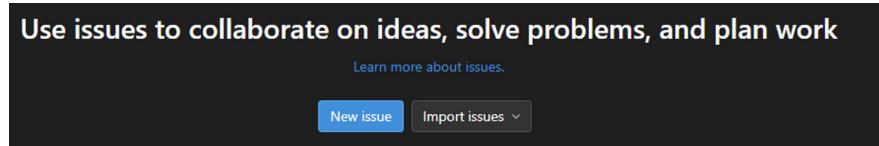
Pour ce cours, nous allons créer deux issues :

- une **epic**

- une user story.

Créer votre première epic

Pour créer une **epic**, le plus simple est d'aller dans le **sous-menu List du menu Issues**, et de cliquer sur le bouton **New Issue**.



Sur la nouvelle page, il faut alors renseigner les champs nécessaires. Commençons par renseigner le titre, ainsi que la description de l'issue :

A screenshot of the Jira 'New Issue' creation page. The title field contains '[EPIC] Gestion des vétérinaires'. The type dropdown is set to 'Issue'. The description field contains the following text:

Dans le projet *PetClinic*, il faut ajouter la gestion des vétérinaires.
Implémentation de :

- Crédation des vétérinaires
- Recherche des vétérinaires
- Mise à jour des vétérinaires
- Suppression des vétérinaires

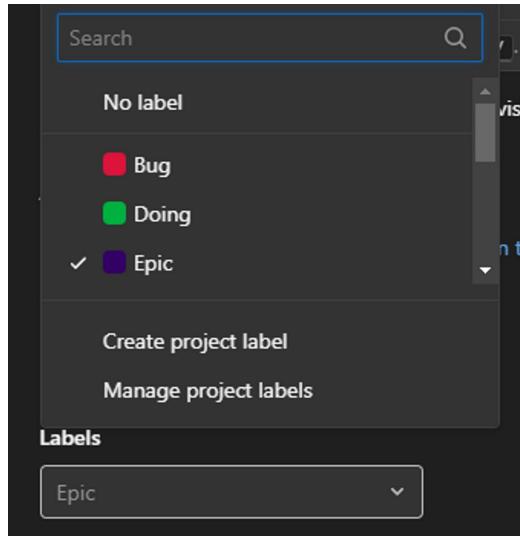
Tâches à faire :

- [x] Crédation des vétérinaires
- [x] Recherche des vétérinaires
- [x] Mise à jour des vétérinaires
- [] Suppression des vétérinaires
- [] Supprimer les vétérinaires
- [x] Suppression de la base de données

Supports Markdown. For quick actions, type / .

This issue is confidential and should only be visible to team members with at least Reporter access.

Ajoutez lui le label "**Epic**" :



Et enregistrez en cliquant sur le bouton Submit Issue.

Créez votre première user story

La deuxième issue sera de type user story. Ajoutez une nouvelle issue en cliquant sur le bouton New Issue :

Si vous revenez sur la liste des issues, vous avez maintenant deux issues créées :

The screenshot shows the Jira search interface with the following details:

- [User story] Suppression du vétérinaire**: Issue #2, created 1 minute ago by Vel, labeled as a User Story.
- [EPIC] Gestion des vétérinaires**: Issue #1, created 4 minutes ago by Vel, labeled as an Epic.

Organisez votre backlog

Nous allons maintenant associer la **user story** fraîchement créée à l'**epic** que l'on a créée auparavant. Pour ce faire, il suffit d'aller sur la description de l'epic, de cliquer sur le bouton +, au niveau du menu **Related Issues**, d'ajouter le numéro de la user story (dans mon cas, #5), et de cliquer sur Add.

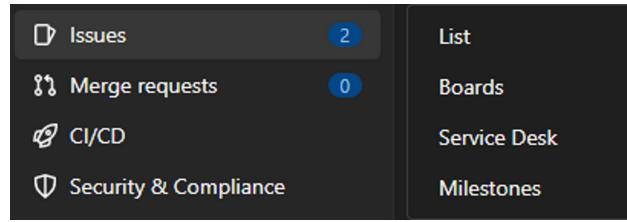
Si nous retournons maintenant sur le menu Boards, nous allons voir que sur les deux boards créés (Development et Product), dans la partie Open, nous avons nos deux issues créées précédemment :

Votre user story associée à votre epic

Cependant, **nous ne voulons voir ici que les issues associées à chacun des boards (user stories pour le board Development, et epic pour le board Product)**. Pour ce faire, il suffit de cliquer sur le bouton **Edit Board**, et d'ajouter uniquement les labels qui nous intéressent :

La dernière chose à faire est de **créer un sprint**, et de voir le **burndown chart associé**. Pour ce faire, il faut aller dans le menu **Milestones**, et cliquer sur le bouton **New Milestone**.

Ensuite, il suffit de remplir les informations du sprint, et de cliquer sur le bouton **Create Milestone** :



Use milestones to track issues and merge requests over a fixed period of time

Organize issues and merge requests into a cohesive group, and set optional start and due dates. [Learn more.](#)

[New milestone](#)

The central part of the page features a dark background illustration of a path with purple dots and a red flag. Below the illustration, a text block explains what milestones are used for, followed by a link to learn more and a blue button to create a new milestone.

New Milestone

Title

Start Date Due Date
Clear start date Clear due date

Description

Write Preview

Le sprint de Final Fantasy tu connais

Supports Markdown

[Create milestone](#) [Cancel](#)

A detailed screenshot of the 'New Milestone' creation form. It includes fields for title, start date, due date, a rich text editor for the description, and a button to create the milestone.

The screenshot shows the GitLab interface for a project named 'spring-petclinic-microservices'. A specific milestone, 'Sprint 1', is selected. The burndown chart shows 100% complete progress from March 10 to April 30. Below the chart, there are sections for 'Issues', 'Merge Requests', 'Participants', and 'Labels'. A note at the bottom states: 'The tabs below will be removed in a future version. Learn more about issue boards, to keep track of issues in multiple lists, using labels, assignees, and milestones. If you're missing something from issue boards, please create an issue on GitLab's issue tracker.'

Afin d'associer des issues au sprint en cours, il suffit d'**aller sur chacune des issues que l'on souhaite associer au sprint, et de modifier leur propriété sur le menu à droite**. Dans cet exemple, j'ai modifié le milestone, le time tracking (en commentant l'issue avec les commandes /estimate et /spend), la due date, le weight, et je me suis assigné l'issue :

The screenshot shows the 'Sprint 1' backlog board. It has three columns: 'Unstarted Issues (open and unassigned)', 'Ongoing Issues (open and assigned)', and 'Completed Issues (closed)'. The 'Ongoing Issues' column contains one item: '[EPIC] Gestion des vétérinaires'.

En résumé :

Dans ce chapitre, vous avez **planifié le développement de votre application grâce à GitLab**. Vous avez créé votre **backlog et vos issues (epic et user stories)**, que vous avez **organisées** afin d'**assigner les tâches**.

Nous avons donc **rempli la première étape de l'intégration continue** :

- Planifiez votre développement.
- Compilez et intégrez votre code.
- Testez votre code.
- Mesurez la qualité de votre code.
- Gérez les livrables de votre application.

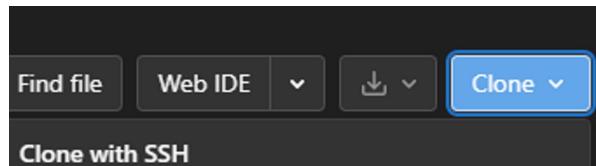
Intégrez votre code en continu

*Dans ce chapitre, nous allons utiliser **GitLab** afin d'**implémenter les différentes étapes de l'intégration continue** pour garantir que notre code fonctionne à tout moment. Vous allez commencer par implémenter les étapes de compilation et de tests.*

Clonez votre projet sur votre poste

Pour pouvoir commencer à travailler sur le projet, afin de récupérer le code source, nous allons dans un premier temps **cloner le repository Git**. Pour ce faire, retournez sur la page d'accueil du projet où se trouvent toutes les instructions nécessaires au clonage du projet.

Cliquez en haut à droite sur le bouton bleu Clone, et copiez la deuxième ligne de la pop-up Clone with HTTPS.



Prenez ensuite une console d'ordinateur, représentant votre poste de développeur, et clonez le repository en tapant la commande suivante :

- git clone [https://gitlab.com/\[votre-nom-d-utilisateur\]/spring-petclinic-microservices.git](https://gitlab.com/[votre-nom-d-utilisateur]/spring-petclinic-microservices.git)
- cd spring-petclinic-microservices

```
aresv@MSI MINGW64 ~/Documents/projetGitLab
$ git clone https://gitlab.com/VelStack/spring-petclinic-microservices.git
Cloning into 'spring-petclinic-microservices'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.

aresv@MSI MINGW64 ~/Documents/projetGitLab
$ cd spring-petclinic-microservices/

aresv@MSI MINGW64 ~/Documents/projetGitLab/spring-petclinic-microservices (main)
$
```

Une fois le clone fait, nous nous retrouvons avec un dossier vide où la **branche master est associée à notre repository GitLab**. Nous allons **ajouter une branche upstream sur GitHub** et **puller** les dernières modifications GitHub. Les commandes à taper sont les suivantes :

- git remote add upstream <https://github.com/spring-petclinic/spring-petclinic-microservices.git>
- git pull upstream master
- git push origin master

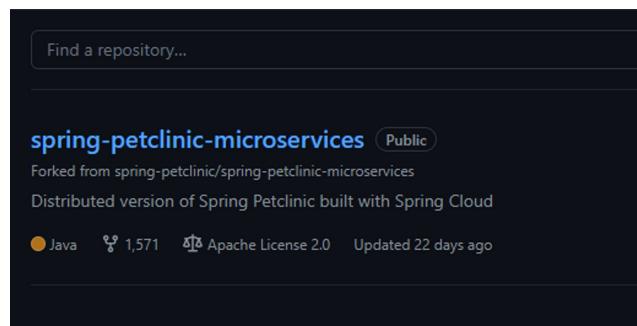
```
aresv@MSI MINGW64 ~/Documents/projetGitLab/spring-petclinic-microservices (main)
$ git remote add upstream https://github.com/spring-petclinic/spring-petclinic-microservices.git

aresv@MSI MINGW64 ~/Documents/projetGitLab/spring-petclinic-microservices (main)
$ git pull upstream master
remote: Enumerating objects: 10440, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 10440 (delta 7), reused 12 (delta 5), pack-reused 10423
Receiving objects: 100% (10440/10440), 7.16 MiB | 11.01 MiB/s, done.
Resolving deltas: 100% (4109/4109), done.
From https://github.com/spring-petclinic/spring-petclinic-microservices
 * branch            master      -> FETCH_HEAD
 * [new branch]      master      -> upstream/master
fatal: refusing to merge unrelated histories
```

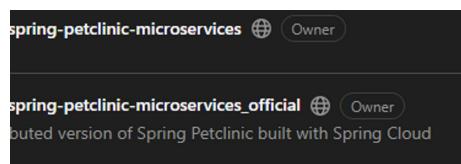
```
→ spring-petclinic-microservices git:(master) git push origin master
Enumerating objects: 9536, done.
Counting objects: 100% (9536/9536), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4077/4077), done.
Writing objects: 100% (9536/9536), 6.99 MiB | 2.79 MiB/s, done.
Total 9536 (delta 3751), reused 9536 (delta 3751)
remote: Resolving deltas: 100% (3751/3751), done.
To https://gitlab.com/openclassrooms-devops/spring-petclinic-microservices.git
 * [new branch]      master -> master
```

Normalement, si les commandes précédentes ont été exécutées, le dossier Git devrait contenir le code source du projet PetClinic, ainsi que toutes les modifications associées.

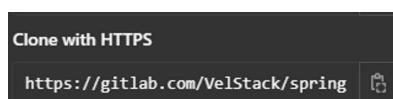
Personnellement, je n'ai pas réussi à procéder comme dans le cours car un changement de nom de branche avait été effectué : pour répondre aux prérogatives du cours, j'ai du importer un nouveau projet avec le lien de mon fork GitHub du repository officiel.



J'ai importé le repository dans GitLab sous forme de nouveau projet



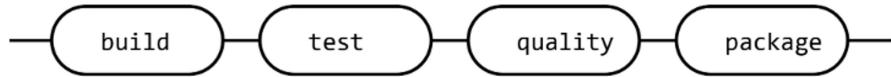
Que j'ai ensuite cloné en local sur mon poste



```
aresv@MSI MINGW64 ~/Documents/projetGitLab
$ git clone https://gitlab.com/VelStack/spring-petclinic-microservices_official.git
Cloning into 'spring-petclinic-microservices_official'...
remote: Enumerating objects: 10446, done.
remote: Total 10446 (delta 0), reused 0 (delta 0), pack-reused 10446
Receiving objects: 100% (10446/10446), 7.17 MiB | 3.09 MiB/s, done.
Resolving deltas: 100% (4112/4112), done.
```

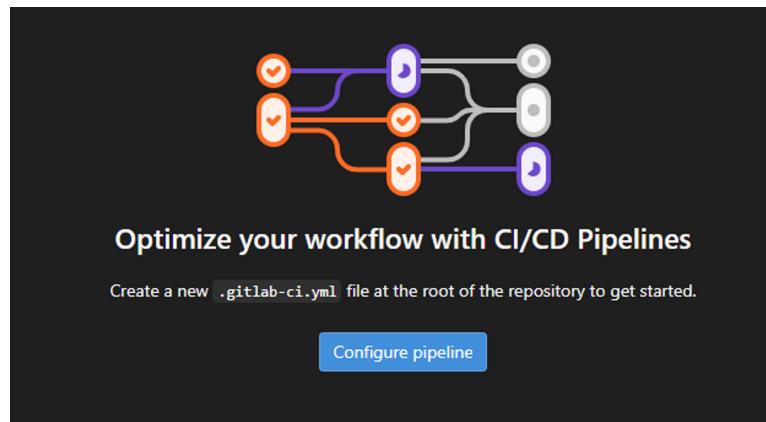
Activez l'intégration continue sur votre projet avec GitLab

Nous allons maintenant ajouter un **pipeline d'intégration continue**, afin d'implémenter les différentes étapes que nous avons vues précédemment. Les étapes de ce pipeline seront lancées successivement, lors de chaque nouveau push du code sur le repo. Voici à quoi ressemblera le pipeline :



Les étapes que nous allons mettre en place

Pour activer l'**intégration continue** sur GitLab, le plus simple est de cliquer sur le bouton **Set up CI/CD** sur la page d'accueil du projet. Cette commande va créer le fichier `gitlab-ci.yml` dans votre projet. C'est sur ce fichier que vous décrirez tout votre pipeline CI/CD avec la syntaxe YAML.



Et d'ajouter les lignes suivantes dans le fichier :

```

1 stages:
2   - build
3   - test
4
5 cache:
6   paths:
7     - .m2/repository
8   key: "$CI_JOB_NAME"
9
10 build_job:
11   stage: build
12   script:
13     - ./mvnw compile
14     | -Dhttps.protocols=TLSv1.2
15     | -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
16     | -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=warn
17     | -Dorg.slf4j.simpleLogger.showDateTime=true
18     | -Djava.awt.headless=true
19     | --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
20   image: openjdk:8-alpine
21
22 test_job:
23   stage: test
24   script:
25     - ./mvnw test
26     | -Dhttps.protocols=TLSv1.2
27     | -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
28     | -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=warn
29     | -Dorg.slf4j.simpleLogger.showDateTime=true
30     | -Djava.awt.headless=true
31     | --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
32   image: openjdk:8-alpine

```

```

stages:
- build
- test

cache:
paths:
- .m2/repository
key: "$CI_JOB_NAME"

build_job:
stage: build
script:
- ./mvnw compile
-Dhttps.protocols=TLSv1.2
-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=warn

```

```

-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
image: openjdk:8-alpine

test_job:
stage: test
script:
- ./mvnw test
-Dhttps.protocols=TLSv1.2
-Dmaven.repo.local=${CI_PROJECT_DIR}/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
image: openjdk:8-alpine

```

Ce fichier est la **pierre angulaire** de l'implémentation d'un pipeline dans **GitLab**. Ce fichier s'appelle **.gitlab-ci.yml**, et c'est ici que nous allons définir notre pipeline. Dans cet exemple, nous avons implémenté deux étapes :

- l'étape de **compilation** avec la tâche **build_job**
- l'étape des **tests** avec la tâche **test_job**.

Découvrons le fichier bloc par bloc !

Définissez les étapes du pipeline

Dans un premier temps, je définis les étapes de mon pipeline avec le mot clé **stages**. Ce mot clé permet de définir l'**ordre des étapes**. Ici, la première étape va être le **build**, et ensuite les **tests**.

Ça veut dire que les **tâches** (ou jobs en anglais) associées à l'étape **build** (ici **build_job**) vont s'exécuter en **premier**, puis les tâches de l'étape **test** (ici **test_job**).

```

stages:
- build
- test

```

Accélérez les étapes avec le cache

Le **deuxième bloc**, avec le mot clé **cache**, est ici utilisé pour **accélérer toutes nos étapes**. Effectivement, dans le cas d'une compilation Java avec Maven (notre cas), cette **compilation récupère beaucoup de dépendances et de librairies externes**. Ces librairies sont stockées dans le répertoire **.m2**.

Grâce à l'utilisation du mot clé **cache** et de la variable prédéfinie de **GitLab \$CI_JOB_NAME**, ce répertoire est **commun à tous les jobs du pipeline**.

```

cache:
paths:
- .m2/repository
key: "$CI_JOB_NAME"

```

Définissez les jobs à effectuer

Ensuite, je déclare **deux jobs**, correspondant chacun à une des étapes de notre pipeline d'intégration continue. Dans ces deux jobs, nous voyons que nous avons **trois différentes lignes**. Découvrons à quoi ces lignes correspondent :

- **stage** : c'est le nom de l'étape qui va apparaître dans notre pipeline d'intégration continue. Cela correspond aussi au stage auquel sera exécuté le job

- **script** : ce sont les lignes de script à lancer afin d'exécuter l'étape. Ici, nous lançons le **script Maven** suivant son lifecycle. Dans la partie **build**, nous lançons la **compilation** ; et dans la partie **test**, nous lançons les tests de l'application. D'autres options sont définies afin d'accélérer le temps de traitement de ces lignes. Le script va alors télécharger Maven, l'outil de compilation, toutes les dépendances de l'application, et lancer la compilation du projet

- **image** : c'est l'image Docker qui va être lancée par GitLab afin d'exécuter les lignes de script que nous avons définies. Ici, l'image **openjdk:8-alpine**, qui contient déjà Java 8, va être lancée afin de pouvoir compiler le projet. Une fois le fichier sauvegardé, le pipeline de build se lance, et vous devriez voir les différentes étapes se lancer (ici, l'étape de build et l'étape de test).

```
build_job:
  stage: build
  script:
    - ./mvnw compile
    -Dhttps.protocols=TLSv1.2
    -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
    -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
    -Dorg.slf4j.simpleLogger.showDateTime=true
    -Djava.awt.headless=true
    --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
  image: openjdk:8-alpine
```

Lors de l'étape de test, le pipeline va exécuter les tests unitaires déjà présents au sein du projet. 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.

Si les tests échouent, le rôle du développeur est alors de :

- soit corriger le test qu'il a écrit, car la fonctionnalité a évolué et le test ne correspond plus
- soit faire évoluer le code, car le test a détecté un bug.

```
test_job:
  stage: test
  script:
    - ./mvnw test
    -Dhttps.protocols=TLSv1.2
    -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
    -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
    -Dorg.slf4j.simpleLogger.showDateTime=true
    -Djava.awt.headless=true
    --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
  image: openjdk:8-alpine
```

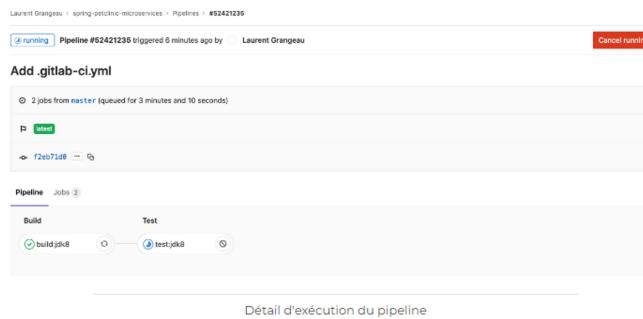
Lancez votre pipeline CI/CD

Pour voir le pipeline complet, il suffit de cliquer sur le sous-menu Pipelines dans le menu CI/CD.

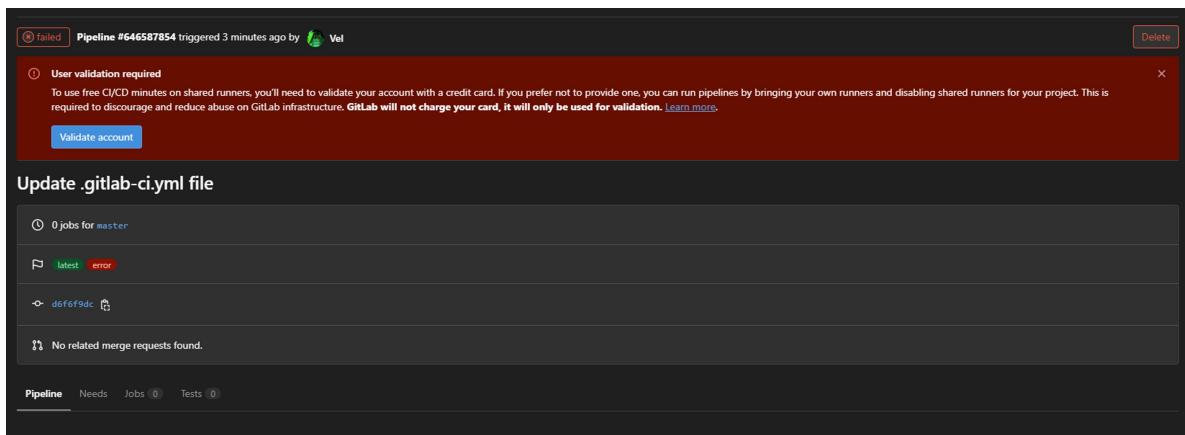
The screenshot shows the GitLab CI/CD pipeline interface. At the top, there are navigation links: 'Tout' (All), 'Achevé' (Completed), 'Branches', 'Mots clés' (Keywords), 'Effacer les caches des coureurs' (Delete runner caches), 'Cl charpie' (Delete runner), and 'Exécuter le pipeline' (Run pipeline). Below this is a search bar labeled 'Filtrer les canalisations' (Filter pipelines) with a magnifying glass icon and a button 'Afficher l'ID du pipeline' (Show pipeline ID) with a dropdown arrow. A table below lists the pipeline details:

| Statut | Pipeline | Déclencheur | Étapes |
|--|---|-------------|--------|
| ✗ manqué il y a 1 minute | Mettre à jour le fichier .gitlab-ci.yml #646587854 ➔ maître ➔ d6f6f9dc ➔ dernier Erreur | | |

En cliquant sur le statut **running** du pipeline, nous avons **plus de détails sur ce pipeline**, les jobs associés ainsi que leurs **status**.



Pour nous



Il demande une vérification de la carte de crédit

Et si l'application contient un bug ?

Afin de **démontrer un workflow typique de CI/CD**, nous allons introduire un **bug** dans l'application. Ainsi, nous verrons **comment le pipeline de CI/CD le détecte**, et **le corrige**. Dans un premier temps, **nous allons récupérer le fichier que nous venons de créer** avec la commande `git pull`:

```
aresv@MSI MINGW64 ~/documents/projetGitLab/spring-petclinic-microservices_official (master)
$ git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 1.69 KiB | 247.00 KiB/s, done.
From https://gitlab.com/VelStack/spring-petclinic-microservices_official
  a05f4fb..d6f6f9d  master      -> origin/master
Updating a05f4fb..d6f6f9d
Fast-forward
  .gitlab-ci.yml | 50 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
  1 file changed, 50 insertions(+)
  create mode 100644 .gitlab-ci.yml
```

Le dossier courant devrait avoir le nouveau fichier `.gitlab-ci.yml`. Nous allons **éditer** un fichier afin d'introduire un bug. Dans un premier temps, **créez une nouvelle branche** qui va accueillir nos modifications avec la commande `git checkout -b refactor-customers`:

```

aresv@MSI MINGW64 ~/documents/projetGitLab/spring-petclinic-microservices_official (master)
$ git checkout -b refactor-customers
Switched to a new branch 'refactor-customers'

aresv@MSI MINGW64 ~/documents/projetGitLab/spring-petclinic-microservices_official (refactor-customers)
$
```

Ensuite, éditez le fichier Java :

"**spring-petclinic-customers-service/src/main/java/org/springframework/samples/petclinic/customers/CustomersServiceApplication.java**".

Supprimez par exemple le " ; " situé à la fin de la ligne package org.springframework.samples.petclinic.customers et **commitez** les changements dans Git avec les commandes :

- **git add spring-petclinic-customers-service/src/main/java/org/springframework/samples/petclinic/customers/CustomersServiceApplication.java**
- **git commit -m "Refactorisation du code des clients"**
- **git push origin refactor-customers**

```

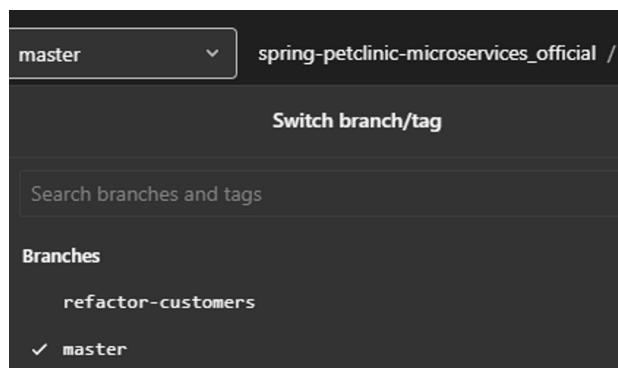
aresv@MSI MINGW64 ~/documents/projetGitLab/spring-petclinic-microservices_official (refactor-customers)
$ git add spring-petclinic-customers-service/src/main/java/org/springframework/samples/petclinic/customers/CustomersServiceApplication.java

aresv@MSI MINGW64 ~/documents/projetGitLab/spring-petclinic-microservices_official (refactor-customers)
$ git commit -m "Refactorisation du code des clients"
[refactor-customers 2797919] Refactorisation du code des clients
 1 file changed, 1 insertion(+), 1 deletion(-)

aresv@MSI MINGW64 ~/documents/projetGitLab/spring-petclinic-microservices_official (refactor-customers)
$ git push origin refactor-customers
Enumerating objects: 23, done.
Counting objects: 100% (23/23), done.
Delta compression using up to 16 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (12/12), 898 bytes | 898.00 KiB/s, done.
Total 12 (delta 2), reused 0 (delta 0), pack-reused 0
remote:
remote: To create a merge request for refactor-customers, visit:
remote:   https://gitlab.com/VelStack/spring-petclinic-microservices_official/-/merge_requests/new?merge_request%5Bsou
e_branch%5D=refactor-customers

```

Vous devriez maintenant avoir deux branches visibles sur la page d'accueil du projet.



Puisque vous avez introduit un bug, le pipeline de build est maintenant en échec :

En cliquant sur la croix rouge dans la colonne Stage, nous avons le détail de l'erreur (ici, le bug que nous avons introduit) :

Refactorisation du code des clients

-o parent `d6f6f9dc` ↗ refactor-customers

No related merge requests found

Pipeline #646602693 failed

Changes 1 **Pipelines** 1

Showing 1 changed file ▾ with 1 addition and 1 deletion

Hide whitespace changes Inline Side-by-side

spring-petclinic-customers-service/src/main/java/org/springframework/samples/petclinic/customers/CustomersServiceApplication.java

```

@@ -13,7 +13,7 @@
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
- package org.springframework.samples.petclinic.customers;
+ package org.springframework.samples.petclinic.customers;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

```

Gérez le fix du bug sur GitLab

Pour **enregistrer le bug**, nous pouvons créer un nouveau **Bug** directement en cliquant sur le bouton **New Issue**. Il faut alors remplir les champs adéquats comme **l'assignee**, le **milestone**, les **labels (Bug et User Story)** ou la due date :

Title (required)
Job Failed #646602693

Add description templates to help your contributors communicate effectively!

Type ⓘ
Issue

Description

Write Preview
Job [#646602693] (https://gitlab.com/VelStack/spring-petclinic-microservices_official/-/commit/27979194aa4c06226a2b04e0ec6a9a4461e4fd11) failed for 27979194aa4c06226a2b04e0ec6a9a4461e4fd11

Supports [Markdown](#). For quick actions, type `/`.

This issue is confidential and should only be visible to team members with at least Reporter access.

Assignee
Vel

Milestone
Sprint 1

Labels
Labels

Due date
2022-09-21

Vel > spring-petclinic-microservices_official > Issues > #1

Open Issue created just now by  Vel Owner

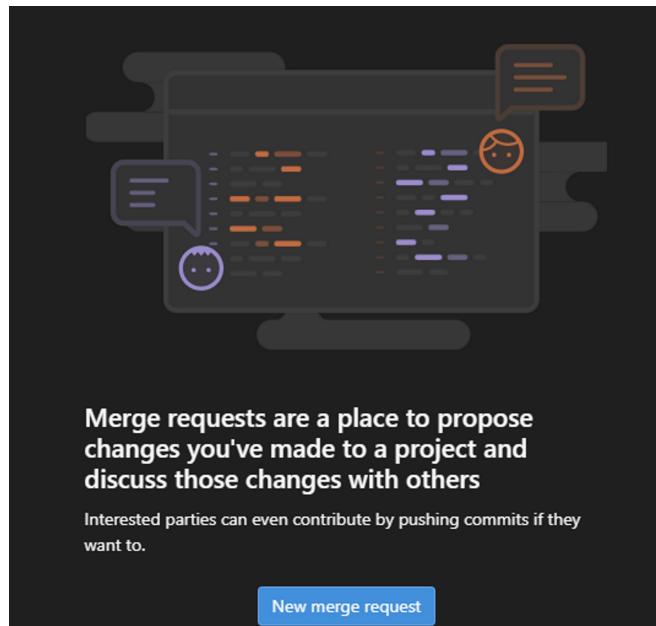
Jod Failed #646602693

Job #646602693 failed for 27979194

Une fois l'issue complétée, nous avons tous les détails du bug à corriger et le job en échec est automatiquement récupéré :

Si nous revenons sur le board **Development**, nous nous apercevons que l'issue créée apparaît dans la colonne **Open**. Nous pouvons alors la **déplacer** dans la colonne **Doing**, car nous allons la **corriger**.

Pour corriger ce **bug automatiquement**, nous allons **créer une merge request**, c'est-à-dire demander à **committer les changements sur notre branche dans la branche principale master**. Pour ce faire, il faut aller dans le menu **Merge Requests** et cliquer sur **New Merge Request**.



Dans la nouvelle page, il faut choisir la branche que nous voulons merger dans la branche principale (ici, la branche monbug) et cliquer sur "Compare branches and continue" :

New Merge Request

Source branch: laurentgrangeau/spring-petclinic-microservices... monbug

Target branch: laurentgrangeau/spring-petclinic-microservices... master

Mon premier bug
Laurent Grangeau authored 1 hour ago

Add .gitlab-ci.yml
Laurent Grangeau authored 1 hour ago

Compare branches and continue

Au prochain écran, il faut remplir les champs de façon adéquate, et cliquer sur **Submit Merge Request**. Pour le champ **Description**, la syntaxe est "Closes" et le numéro du bug, ici, le 6 :

- **Title : WIP: Mon premier bug**
- **Description : Closes #6**
- **Assignee : Assign to me**
- **Milestone : Sprint 1**

New Merge Request

From monbug into master

Change branches

Title: WIP: Mon premier bug

Remove the **WIP:** prefix from the title to allow this Work In Progress merge request to be merged when it's ready.
Add [description templates](#) to help your contributors communicate effectively!

Description

Write Preview

Closes #6 |

Markdown and quick actions are supported

Attach a file

Description de mon bug

ATTENTION : dans l'interface GitLab, l'option "Delete source branch when merge request is accepted." est cochée de base, prudence !!!

Merge options

Delete source branch when merge request is accepted.

Squash commits when merge request is accepted. [?](#)

Une fois la merge request créée, nous avons tous les détails de celle-ci :

The screenshot shows a GitLab merge request for a branch named 'monbug' into 'master'. The title is 'WIP : Mon premier bug'. It includes a note 'Closes #6'. A pipeline status bar indicates a failure for pipeline #52426298. Approval status shows 'No approval required'. Below the pipeline, there's a button to 'Merge' or 'Resolve WIP status'. The merge request has 0 likes and 0 dislikes. A discussion section shows a mention from Laurent Grangeau. A text input field is labeled 'Description de mon bug'.

De mon côté : pour l'exemple j'ai laissé l'option "suppression de la branche source" après avoir corrigé mon code

Corrigez votre bug

Il est temps de corriger notre bug. Nous allons éditer le fichier que nous avons modifié. Pour cela, nous allons réintroduire le ";" manquant, puis commiter le code corrigé sur la branche `monbug`.

- `git add spring-petclinic-customers-service/src/main/java/org/springframework/samples/petclinic/customers/CustomersServiceApplication.java`
- `git commit -m "Correction du bug clients"`
- `git push origin monbug`

Vous allez constater qu'une fois le code pushé sur la branche `monbug`, **le pipeline de build se lance** afin de vérifier que le code que nous avons envoyé **fonctionne**.

Une fois que le **pipeline s'est terminé** en succès, nous pouvons alors enlever le statut **Work in progress** en cliquant sur le bouton **Resolve WIP status**, pour ensuite cliquer sur le bouton **Merge** :

Open Opened 20 minutes ago by Laurent Grangeau

Close merge request

Mon premier bug

Closes #6

Edited 3 minutes ago by Laurent Grangeau

The screenshot shows a merge request for a branch named 'monbug' into the 'master' branch. The pipeline for this merge request has passed. There are merge options available, including 'Merge', 'Delete source branch', and 'Squash commits'. A note indicates that 2 commits and 1 merge commit will be added to the master branch. The merge request is closed and linked to issue #6. Below the merge request, there are social sharing icons for thumbs up, thumbs down, and a share button.

Bug résolu, mergez les branches !

Le pipeline se lance alors une dernière fois pour vérifier que le code mergé ne casse pas la compilation.

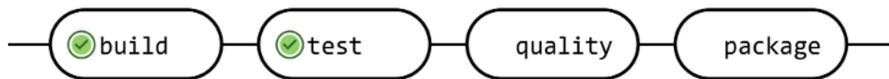
Enfin, si nous revenons sur le board Development, nous voyons que le bug est automatiquement fermé, suite à notre merge request.

En résumé

Dans ce chapitre, nous avons implémenté les premières étapes de notre pipeline d'intégration continue grâce à GitLab.

Pour cela, nous avons utilisé le fichier **gitlab-ci.yml** que nous avons configuré pour qu'il lance :

- une étape de **build de l'application avec Maven** ;
- une étape de **test de l'application**, en lançant les tests développés précédemment par les développeurs.



Les 2 premières étapes du pipeline sont fonctionnelles ✓

Garantissez la qualité de votre code

Dans ce dernier chapitre de la partie CI, nous allons voir les deux dernières étapes du **pipeline d'intégration continue** : la **qualité de code** et la **gestion des livrables**.

Mesurez la qualité de votre code

Commençons par l'analyse de **code statique**, afin de contrôler la **qualité du code**.

Nous allons donc modifier le **pipeline de code**, afin d'ajouter cette analyse de code. Il faut alors modifier le fichier **.gitlab-ci.yml** afin qu'il ressemble à ceci :

```
stages:
- build
- test
- quality

cache:
paths:
- .m2/repository
key: "$CI_JOB_NAME"

build_job:
stage: build
script:
- ./mvnw compile
-Dhttps.protocols=TLSv1.2
-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARNING
-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
image: openjdk:8-alpine

test_job:
stage: test
script:
- ./mvnw test
-Dhttps.protocols=TLSv1.2
-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARNING
-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
image: openjdk:8-alpine

code_quality_job:
stage: quality
image: docker:stable
allow_failure: true
services:
- docker:stable-dind
script:
- mkdir codequality-results
- docker run
--env CODECLIMATE_CODE="$PWD"
--volume "$PWD":/code
--volume /var/run/docker.sock:/var/run/docker.sock
--volume /tmp/cc:/tmp/cc
codeclimate/codeclimate analyze -f html > ./codequality-results/index.html
artifacts:
paths:
- codequality-results/
```

Ne pas oublier de rajouter la variable "quality" dans la catégorie stages :

```
stages:
- build
- test
- quality
```

Ici, j'ai ajouté une étape supplémentaire de qualité de code. L'étape (**le stage**) **quality** est défini dans le bloc **stages**. Cela veut dire que le **pipeline** va ajouter un **job de qualité** à la suite de la **compilation** et des **tests**. Ce job, je l'ajoute à la fin du fichier et il est appelé **code_quality_job**.

```

code_quality_job:
  stage: quality
  image: docker:stable
  allow_failure: true
  services:
    - docker:stable-dind
  script:
    - mkdir codequality-results
    - docker run
      --env CODECLIMATE_CODE="$PWD"
      --volume "$PWD":/code
      --volume /var/run/docker.sock:/var/run/docker.sock
      --volume /tmp/cc:/tmp/cc
      codeclimate/codeclimate analyze -f html > ./codequality-results/index.html
  artifacts:
    paths:
      - codequality-results/

```

Dans ce job, nous retrouvons les **3 lignes** des étapes précédentes, plus d'autres lignes :

- **allow_failure** : cette ligne **autorise l'échec de l'étape de qualité**. Comme ce n'est pas une **étape critique**, nous nous permettons **d'autoriser l'échec** et de laisser le pipeline continuer

allow_failure: true

- **services** : ce nouveau mot clé de **GitLab** permet de démarrer un **démon Docker**, pour que l'exécution de notre programme d'analyse de code puisse se faire. **Cette ligne nous permet d'exécuter Docker au sein de l'image, afin d'exécuter l'analyse de code**

services:
- docker:stable-dind

- **script** : cette ligne est un peu plus compliquée que les lignes de script précédentes. **La première ligne va créer un dossier codequality-results/ qui contiendra le résultat de l'analyse de code. La deuxième ligne monte le code à l'intérieur d'une image Docker via le dossier /code, et lance l'analyse via le programme codequality.** Le résultat sera exporté dans le dossier **codequality-results**

```

script:
  - mkdir codequality-results
  - docker run
    --env CODECLIMATE_CODE="$PWD"
    --volume "$PWD":/code
    --volume /var/run/docker.sock:/var/run/docker.sock
    --volume /tmp/cc:/tmp/cc
    codeclimate/codeclimate analyze -f html > ./codequality-results/index.html

```

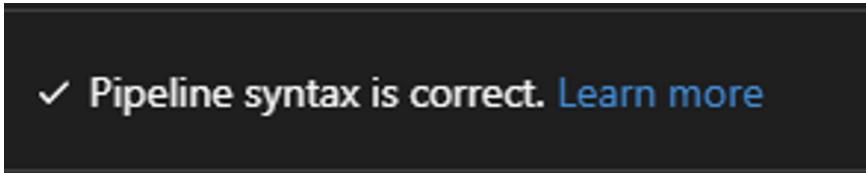
- **artifact** : cette ligne est un prérequis de **GitLab** si nous voulons voir notre **évolution de qualité**. Le dossier **codequality-results/** sera stocké au sein de **GitLab** afin de pouvoir voir le résultat de l'**analyse du scan**. Ce résultat sera disponible et visible au sein du job **code_quality_job**.

```
66     artifacts:
67       paths:
68         - codequality-results/
```

Tout le script est exécuté au sein de l'image **docker:stable**. Cette image permet de **démarrer le programme d'analyse de code**.

Une fois le fichier **commité sous Git** et envoyé sur **GitLab** via les commandes suivantes, le **pipeline d'intégration continue** va alors se **mettre à jour et lancer une compilation**, suivie d'une **analyse statique du code**.

*Il va aussi afficher un petit encart dans **GitLab** concernant la syntaxe :*



✓ Pipeline syntax is correct. [Learn more](#)

Le code est alors analysé par **GitLab**, et le **rapport généré stocké au niveau des artefacts**. 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.

Ça y est, l'étape de qualité est implémentée !



D'autres outils existent pour voir la qualité d'un code de développement. Le plus connu est **SonarQube**, qui permet d'afficher des rapports de qualité, l'évolution de ceux-ci, ainsi qu'une détection partielle des erreurs.

Packagez votre application pour la déployer

La prochaine étape après la **qualité du code** est le **packaging de l'application**, afin de pouvoir la déployer plus facilement. Pour ce projet, nous allons choisir **Docker** comme **programme de packaging**.

GitLab vient avec une **registry Docker** incluse, ce qui nous permet de **stocker ces images** au sein de **GitLab**. Pour pouvoir packager nos **images Docker**, il est nécessaire d'**ajouter une nouvelle étape à notre pipeline d'intégration continue**. Nous allons une nouvelle fois **modifier le fichier .gitlab-ci.yml** pour ajouter cette nouvelle étape. Le fichier final ressemblera alors à ceci :

```
stages:
- build
- test
- quality
- package
```

```
cache:
paths:
- .m2/repository
```

```

key: "$CI_JOB_NAME"

build_job:
stage: build
script:
- ./mvnw compile
-Dhttps.protocols=TLSv1.2
-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
image: openjdk:8-alpine

test_job:
stage: test
script:
- ./mvnw test
-Dhttps.protocols=TLSv1.2
-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
image: openjdk:8-alpine

code_quality_job:
stage: quality
image: docker:stable
allow_failure: true
services:
- docker:stable-dind
script:
- mkdir codequality-results
- docker run
--env CODECLIMATE_CODE="$PWD"
--volume "$PWD":/code
--volume /var/run/docker.sock:/var/run/docker.sock
--volume /tmp/cc:/tmp/cc
codeclimate/codeclimate analyze -f html > ./codequality-results/index.html
artifacts:
paths:
- codequality-results/

package_job:
stage: package
services:
- docker:stable-dind
variables:
DOCKER_HOST: tcp://docker:2375
script:
- apk add --no-cache docker
- docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
- ./mvnw install -PbuildDocker -DskipTests=true -DpushImage
-Dhttps.protocols=TLSv1.2
-Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
image: openjdk:8-alpine

```

J'ajoute dans le fichier la dernière étape de mon pipeline d'intégration continue, l'**étape (le stage) package**, qui s'exécute à la suite de l'**étape quality**. J'ajoute ensuite le **job package_job** associé à cette étape de qualité.

```

stages:
- build
- test
- quality
- package

```

Ce job supplémentaire **compile le projet et l'encapsule dans un conteneur**. Il est ensuite poussé sur la **registry** de **GitLab**. Nous retrouvons toutes les lignes que nous avons vues précédemment. La partie script lance cependant quelques commandes supplémentaires :

- tout d'abord, nous installons le **client Docker** dans l'image **openjdk:8-alpine** afin de pouvoir lancer les commandes propres à Docker
- ensuite, nous nous connectons sur la **registry interne de GitLab** afin de **pouvoir pousser les images Docker de façon sécurisée**

- enfin, nous lançons la commande **Maven de création de l'image Docker**.

```
package_job:
  stage: package
  services:
    - docker:stable-dind
  variables:
    DOCKER_HOST: tcp://docker:2375
  script:
    - apk add --no-cache docker
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
    - ./mvnw install -PbuildDocker -DskipTests=true -DpushImage
      -Dhttps.protocols=TLSv1.2
      -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
      -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
      -Dorg.slf4j.simpleLogger.showDateTime=true
      -Djava.awt.headless=true
      --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true
  image: openjdk:8-alpine
```

Ce processus nous permettra, dans la livraison continue, 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. Les images Docker ainsi packagées se retrouvent sur la page de la **registry** :

Container Registry

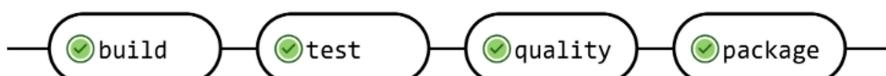
With the Docker Container Registry integrated into GitLab, every project can have its own space to store its Docker images.

Learn more about [Container Registry](#).

| | |
|---|--|
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-admin-server | |
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-api-gateway | |
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-config-server | |
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-customers-service | |
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-discovery-server | |
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-hystrix-dashboard | |
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-vets-service | |
| > laurentgrangeau/spring-petclinic-microservices/spring-petclinic-visits-service | |

Registry GitLab de votre projet, avec les packages du pipeline

Nous avons maintenant toutes les étapes nécessaires pour l'intégration continue. Comme prévu, notre code est compilé en continu, testé, analysé puis packagé, prêt à être déployé sur de nouveaux environnements.



Toutes les étapes de l'intégration continue sont implémentées

Les autres outils de l'intégration continue

Vous avez donc vu tout au long de cette partie comment mettre en place l'**intégration continue** avec **GitLab**, mais sachez qu'il existe d'autres outils reprenant les mêmes concepts. Le plus connu et le plus utilisé d'entre eux est **Jenkins**. 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**.

Le principe est strictement le même que **GitLab** : il s'agit d'un **fichier de description du pipeline d'intégration continue** qui va contenir toutes les étapes à lancer, afin de garantir que le code compile et soit de qualité à tout moment. Cependant, il est nécessaire d'installer **Jenkins** dans votre entreprise, de le configurer, de le maintenir et de le mettre à jour, ce qui peut s'avérer long et fastidieux.

Pour ceux qui ne voudraient pas passer leur temps à maintenir ce genre d'outils, il existe aussi d'autres outils en mode **Software-as-a-Service (SaaS)**, où la **maintenance** et l'**évolution** sont **garanties par le fournisseur**. Ces outils peuvent être mieux adaptés pour mettre en place rapidement et sans effort un pipeline d'intégration continue. Les outils les plus connus dans cet écosystème sont **Travis CI** et **CircleCI**.

Le gros avantage de ces outils est que la maintenance n'est pas à la charge de l'équipe, mais du fournisseur. De plus, ces outils peuvent se **connecter automatiquement sur Github.com** pour la plupart, ce qui évite aussi les configurations longues et fastidieuses des différents outils.

Enfin, GitHub a sorti une beta de son nouveau service à destination des développeurs, afin de pouvoir implémenter rapidement des **pipelines d'intégration continue** : **GitHub Actions**. Le principe est toujours le même : un fichier **.workflow** permet de **créer un pipeline**, afin de **compiler et déployer du code** sur **n'importe quelle plateforme**. L'avantage principal de **GitHub Actions** est que ce dernier est directement intégré dans **GitHub**.

En résumé :

Dans ce chapitre, nous avons vu **comment ajouter les étapes de mesure de la qualité de votre code ainsi que de packaging de votre application, pour sauvegarder les artefacts prêts à être déployés**.

Qu'est-ce que la livraison continue ?

Dans cette partie, nous allons voir comment **livrer votre code en continu** pour le mettre en production **rapidement**, et de manière **fiable**.

La **livraison continue** est la suite logique de l'**intégration continue**. Dans l'**intégration continue**, nous cherchons à ce que le code compile bien, mais aussi qu'il soit **fonctionnel en production** et de **qualité**, en lançant le plus régulièrement possible les **tests unitaires**. Mais il existe d'autres types de tests, tout aussi importants, pour garantir la qualité du code. **Ces tests ne peuvent cependant pas être lancés sans avoir un environnement déployé**.

Attention, **la livraison continue ne doit pas être confondue avec le déploiement continu**, qui est la suite logique de la livraison continue. Ces deux disciplines ont comme objectif de déployer une application en production. La différence se trouve dans l'**automatisation du déploiement en production**. La livraison continue s'arrête avant la production, et la mise en production reste un acte manuel (que ce soit avec un outil, ou automatisé via un clic de bouton, ou bien manuellement). La mise en production est soumise alors à la validation d'un être humain.

Le **déploiement continu**, quant à lui, est l'**extension de la livraison continue** : le déploiement se fait de manière **automatisée** par un **pipeline**. Toutes les étapes de compilation, tests unitaires et autres tests automatisés doivent être alors au **vert** avant de procéder au déploiement.

La **livraison continue** est une discipline où l'application est **construite** de manière à pouvoir être **mise en production** à **n'importe quel moment**.

Pour atteindre la mise en œuvre de la **livraison continue** sur une **application**, il est nécessaire de mettre en place

plusieurs étapes supplémentaires au sein de notre **pipeline**. Elles sont au nombre de 5 :

- **La codification de l'infrastructure avec l'Infrastructure-as-Code.**
- **Le déploiement de votre application.**
- **Le test de votre application en environnement de test.**
- **La supervision de l'application.**
- La mise en place de **notifications d'alerte**.

[Étape 1 : Codifiez votre infrastructure avec l'Infrastructure-as-Code](#)

Infrastructure-as-Code : L'**Infrastructure-as-Code** est une pratique qui consiste à décrire une **infrastructure avec du code**. Ce code est alors **stocké avec le code de l'application**, et **fait partie intégrante de cette dernière**.

Les avantages sont nombreux :

- **possibilité de créer des environnements à la demande**
- **création d'environnement en quelques minutes, contre plusieurs semaines dans une entreprise classique**
- **pilotage de l'infrastructure grâce au pipeline de livraison continue**
- **connaissance des logiciels installés sur la plateforme, grâce à l'outillage**
- **montée de version des environnements automatisés.**

Les outils principaux de l'**Infrastructure-as-Code** sont **Docker**, **Chef**, **Puppet**, **Ansible** et **Terraform**.

[Étape 2 : Déployez votre application](#)

L'étape la plus importante de la livraison continue est le **déploiement du package** que nous avons précédemment créé lors de l'**intégration continue**. Les avantages d'utiliser un outil pour automatiser le déploiement de l'application sont nombreux :

- **cela permet à l'équipe de se concentrer sur le développement, là où elle a sa valeur à ajouter**
- **n'importe qui dans l'équipe peut déployer des logiciels**
- **les déploiements deviennent beaucoup moins sujets aux erreurs et beaucoup plus reproductibles**
- **déployer sur un nouvel environnement est facile**
- **les déploiements peuvent être très fréquents.**

Pour pouvoir déployer les artefacts précédemment créés, vous pourrez utiliser **Spinnaker**, **XLDeploy** ou **UrbanCode**.

[Étape 3 : Testez votre application](#)

C'est dans cette étape que nous allons **ajouter d'autres types de tests**, plus **pertinents** et plus **fonctionnels**, afin

de garantir que l'application fonctionne comme nous l'avons estimé.

L'avantage de tester à ce stade du pipeline est que l'**application tourne sur un environnement de test, presque identique à celui de la production**. Son **comportement** sera donc le plus **fidèle** possible à celui qu'elle aura en **production**.

Ces tests peuvent être de différents types :

- **Test d'acceptance** : Les **tests d'acceptance** sont des tests **formels** exécutés pour **vérifier si un système satisfait à ses exigences opérationnelles**. Ils **exigent que l'application entière soit opérationnelle et se concentrent sur la réPLICATION DES COMPORTEMENTS DES UTILISATEURS**. Mais ils peuvent aussi **aller plus loin en mesurant la performance du système**, et rejeter les changements si certains objectifs ne sont pas atteints.

Ces tests peuvent être **automatisés**, mais aussi manuels, avec une équipe de test dédiée qui regardera si le logiciel correspond au besoin.

Pour lancer des **tests d'acceptance**, vous pourrez utiliser Confluence, **FitNesse** ou **Ranorex**.

- **Test de performance** : Les **tests de performance** vérifient le **comportement du système lorsqu'il est soumis à une charge importante**. Ces tests ne sont pas fonctionnels et peuvent prendre **différentes formes** pour comprendre la **fiabilité**, la **stabilité** et la **disponibilité** de la plateforme. Par exemple, il peut s'agir d'observer les temps de réponse lors de l'exécution d'un grand nombre de requêtes, ou de voir comment le système se comporte avec une quantité importante de données.

Les **tests de performance** sont par nature **assez coûteux** à mettre en œuvre et à exécuter, mais ils peuvent vous aider à comprendre si de nouveaux changements vont dégrader votre système.

Pour faire des tests de performance, vous pourrez utiliser **JMeter**, **Apache Bench** ou **Gatling**.

- **Smoke test** : Les **smoke tests** sont des **tests de base** qui vérifient les **fonctionnalités de base de l'application**. Ils sont conçus pour être **rapides** à exécuter, et leur but est de vous donner l'**assurance que les principales caractéristiques de votre système fonctionnent** comme prévu. Ils peuvent être utiles juste après une nouvelle build, pour décider si vous pouvez ou non exécuter des tests plus coûteux, ou juste après un déploiement pour s'assurer que l'application fonctionne correctement dans le nouvel environnement déployé.

Par exemple, les smoke tests peuvent s'assurer que la base de données répond et est correctement configurée, mais aussi que les différents composants sont présents et envoient des données correctes, comme des API qui devraient répondre un code HTTP 200, ou une page web qui devrait s'afficher.

Pour s'assurer du bon fonctionnement de l'application, vous pourrez utiliser **Selenium**, **SoapUI** ou **Cypress**.

Étape 4 : Supervisez le comportement de votre application

Le **monitoring**, ou **supervision**, intervient une fois que notre application est déployée sur un environnement, que ce soit un environnement de staging, de test, de démonstration ou bien l'environnement de production lui-même.

Le principe est de **récupérer certaines métriques qui ont du sens pour ceux qui interviennent sur l'application**. Cela peut être par exemple le **nombre de connexions HTTP**, le **nombre de requêtes à la base de données**, le **temps de réponse de certaines pages** ; mais aussi des **métriques plus orientées métier**, comme le chiffre d'affaires généré, ou le nombre de personnes inscrites sur l'application.

Pour avoir un monitoring de vos applications, vous pourrez utiliser la suite **Elastic**, **Prometheus** ou **Graylog**.

Les **métriques peuvent être aussi sur la partie livraison en elle-même, ou sur le processus de développement**. Par exemple, l'équipe peut mesurer le **nombre de déploiements qu'elle effectue par jour**, ou encore deux autres indicateurs qui sont importants afin de voir la performance de l'équipe sur la correction d'erreurs qui peuvent survenir en production :

- **Le Mean-Time-Between-Failure** : Le **Mean-Time-Between-Failure** (ou **MTBF**) est le **temps moyen qui sépare deux erreurs en production**. Plus ce temps est élevé, plus le système est stable et fiable, notamment du fait de la qualité des tests qui sont joués lors de la livraison continue.
- **Le Mean-Time-To-Recover** : Le **Mean-Time-To-Recover** (ou **MTTR**) est le **temps moyen de correction entre deux erreurs de production**. Plus ce temps est faible, plus l'équipe est apte à détecter des erreurs et à les corriger rapidement.

Des outils comme **Dynatrace**, **Sysdig** ou **New Relic** permettent d'avoir ces métriques.

Étape 5 : Mettez en place des notifications d'alertes

La première version d'une **nouvelle fonctionnalité** ou d'un **nouveau produit** ne couvre souvent pas entièrement les **besoins** des clients. Même lorsque l'équipe passe des semaines ou des mois à construire quelque chose, le produit final est souvent voué à manquer des fonctionnalités importantes. C'est le principe du **Minimum Viable Product (MVP)** en **Agile**.

Il arrive donc très souvent de livrer des logiciels incomplets ou buggés, si l'équipe veut aller assez vite. Au lieu de vouloir éviter cela, il est nécessaire d'adopter l'idée de livrer des petites pièces de valeur.

En **livrant plus vite**, nous pouvons **réparer les bugs tant que les livraisons restent petites**, et que nous savons ce qui a été modifié dans l'application. Quand les développements grossissent, ils deviennent plus difficiles à gérer et à remanier. Un feedback rapide, grâce aux tests en production et au monitoring, permet d'intervenir et de corriger le problème dès que possible. Il nous permet d'apprendre des clients, et des erreurs, au bon moment.

Une fois le déploiement fini et les différents tests effectués, il est nécessaire d'avoir un **feedback rapide de l'utilisation du logiciel**. En effet, si le **déploiement de la nouvelle version du logiciel apporte des bugs malgré les différents tests effectués**, il faut alors les détecter le plus rapidement possible, afin de pouvoir proposer une **nouvelle correction au logiciel**.

Pour avoir un feedback rapide de vos déploiements, vous pourrez tout simplement utiliser **Slack**, **Trello** ou **Twitter**.

En résumé :

- La **codification de l'infrastructure** avec l'**Infrastructure-as-Code**, et notamment en **construisant des images avec Docker**.
- Le **déploiement des images de votre application**.
- Le **test de votre application en environnement de test**.
- La **supervision de l'application**, grâce à **plusieurs métriques suivies automatiquement**.
- La **mise en place de notifications d'alerte automatiques** grâce à **Twitter**, **Slack** ou **Trello**.

Codifiez votre infrastructure

Dans ce chapitre, nous allons voir comment l'**Infrastructure-as-Code** permet de **faciliter le déploiement des applications** grâce aux fameux **conteneurs**.

Construisez les images de votre application avec Docker

Notre application PetClinic est construite à partir de fichiers, nommés **dockerfiles**, déjà présents dans le contrôle de code source. Ces **dockerfiles** sont présents dans le **répertoire Docker** du projet et contiennent les lignes suivantes :

| Name | Last commit |
|--------------|---|
| .. | |
| └ grafana | Fix Grafana dashboard with new HTTP status code |
| └ prometheus | introducing micrometer/prometheus metric collection |
| └ Dockerfile | Use Spring Boot layered jars to optimize Docker images (#155) |

```
FROM openjdk:8-jre-alpine

VOLUME /tmp

ARG DOCKERIZE_VERSION
ARG ARTIFACT_NAME
ARG EXPOSED_PORT

ENV SPRING_PROFILES_ACTIVE docker

ADD https://github.com/jwilder/dockerize/releases/download/\${DOCKERIZE\_VERSION}/dockerize-alpine-linux-amd64-\${DOCKERIZE\_VERSION}.tar.gz
dockerize.tar.gz
RUN tar xzf dockerize.tar.gz
RUN chmod +x dockerize

ADD ${ARTIFACT_NAME}.jar /app.jar

RUN touch /app.jar

EXPOSE ${EXPOSED_PORT}

ENTRYPOINT ["java", "-XX:+UnlockExperimentalVMOptions", "-XX:+UseCGroupMemoryLimitForHeap", "-Djava.security.egd=file:/dev/.urandom", "-jar", "/app.jar"]
```

Ma version du cours :

Dockerfile 972 bytes

Open in Web IDE | Replace | Delete |

```

1 FROM openjdk:11-jre as builder
2 WORKDIR application
3 ARG ARTIFACT_NAME
4 COPY ${ARTIFACT_NAME}.jar application.jar
5 RUN java -Djarmode=layertools -jar application.jar extract
6
7 # Download dockerize and cache that layer
8 ARG DOCKERIZE_VERSION
9 RUN wget -O dockerize.tar.gz https://github.com/jwilder/dockerize/releases/download/${DOCKERIZE_VERSION}/dockerize-alpine-linux-amd64-${DOCKERIZE_VERSION}.tar.gz
10 RUN tar xzf dockerize.tar.gz
11 RUN chmod +x dockerize
12
13
14 # wget is not installed on adoptopenjdk:11-jre-hotspot
15 FROM adoptopenjdk:11-jre-hotspot
16
17 WORKDIR application
18
19 # Dockerize
20 COPY --from=builder application/dockerize ./
21
22 ARG EXPOSED_PORT
23 EXPOSE ${EXPOSED_PORT}
24
25 ENV SPRING_PROFILES_ACTIVE docker
26
27 COPY --from=builder application/dependencies/ .
28 COPY --from=builder application/spring-boot-loader/ .
29 COPY --from=builder application/snapshot-dependencies/ .
30 COPY --from=builder application/application/ .
31 ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]

```

```

FROM openjdk:11-jre as builder
WORKDIR application
ARG ARTIFACT_NAME
COPY ${ARTIFACT_NAME}.jar application.jar
RUN java -Djarmode=layertools -jar application.jar extract

# Download dockerize and cache that layer
ARG DOCKERIZE_VERSION
RUN wget -O dockerize.tar.gz https://github.com/jwilder/dockerize/releases/download/\${DOCKERIZE\_VERSION}/dockerize-alpine-linux-amd64-\${DOCKERIZE\_VERSION}.tar.gz
RUN tar xzf dockerize.tar.gz
RUN chmod +x dockerize

# wget is not installed on adoptopenjdk:11-jre-hotspot
FROM adoptopenjdk:11-jre-hotspot

WORKDIR application

# Dockerize
COPY --from=builder application/dockerize ./

ARG EXPOSED_PORT
EXPOSE ${EXPOSED_PORT}

ENV SPRING_PROFILES_ACTIVE docker

COPY --from=builder application/dependencies/ .
COPY --from=builder application/spring-boot-loader/ .
COPY --from=builder application/snapshot-dependencies/ .
COPY --from=builder application/application/ .
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]

```

Ce **dockerfile** part d'une image qui contient déjà la **version 8 de Java** et copie tous les .jar des différents projets, afin de construire les différentes images associées. Enfin, la dernière étape de ce **dockerfile** est de lancer la commande Java avec le jar associé.

Dans cette partie du cours, vous allez modifier le **dockerfile** pour voir l'impact de l'**Infrastructure-as-Code** sur votre **pipeline de déploiement**. Pour ce faire, ouvrez le **fichierDockerfile** présent dans le dossier Docker, afin de modifier la **version du runtime de Java** en version **openjdk:13-alpine**

Write Preview changes

```
Y master docker/Dockerfile
```

```
1 FROM openjdk:13 alpine
2 VOLUME /tmp
3 ARG DOCKERIZE_VERSION
4 ARG ARTIFACT_NAME
5 ARG EXPOSED_PORT
6 ENV SPRING_PROFILES_ACTIVE docker
7
8 ADD https://github.com/jwilder/dockerize/releases/download/${DOCKERIZE_VERSION}/dockerize-alpine-linux-amd64-${DOCKERIZE_VERSION}.tar.gz dockerize.tar.gz
9 RUN tar xzf dockerize.tar.gz
10 RUN chmod +x dockerize
11 ADD ${ARTIFACT_NAME}.jar /app.jar
12 RUN touch /app.jar
13 EXPOSE ${EXPOSED_PORT}
14 ENTRYPOINT ["java", "-XX:+UnlockExperimentalVMOptions", "-XX:+UseCGroupMemoryLimitForHeap", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app.jar"]
```

Commit message

Target Branch

Commit changes Cancel

Changez la version de l'openjdk dans le dockerfile

Avant de pousser ce fichier sur Git, vous allez modifier la version de release de chaque pom.xml présent dans le projet, pour incrémenter le numéro de version des images Docker créées, et ainsi ne pas écraser les versions précédemment buildées :

Write Preview changes

```
Y master spring-petclinic-admin
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>org.springframework.samples.petclinic.admin</groupId>
8   <artifactId>spring-petclinic-admin-server</artifactId>
9   <packaging>jar</packaging>
10  <description>Spring Boot Admin server</description>
11
12  <parent>
13    <groupId>org.springframework.samples</groupId>
14    <artifactId>spring-petclinic-microservices</artifactId>
15    <version>2.0.7</version>
16  </parent>
17
18  <properties>
19    <spring-boot-admin.version>2.0.3</spring-boot-admin.version>
20    <docker.image.exposed.ports>9090</docker.image.exposed.ports>
21    <docker.image.dockerfile.dir>${basedir}./..</docker.image.dockerfile.dir>
22  </properties>
23
24  <dependencies>
25    <!-- Spring Boot -->
26    <dependency>
27      <groupId>org.springframework.boot</groupId>
28      <artifactId>spring-boot-starter</artifactId>
29    </dependency>
30    <dependency>
31      <groupId>org.springframework.cloud</groupId>
```

Commit message

Target Branch

Commit changes Cancel

Changez la version des images Docker créées dans les fichiers pom.xml

Il y a plusieurs pom.xml où il faut ajouter le bon numéro de version ! Personnellement j'ai eu quasi 10 fichiers à modifier

Une fois ces modifications faites, poussez les fichiers sur Git :

- **git add**
- **git commit -m "Modification de la version de Java et incrémentation du numéro de version"**
- **git push origin master**

Le pipeline d'intégration continue devrait se lancer :

The screenshot shows a pipeline interface with the following details:

- Pipeline Status:** Pipeline #54827325 triggered 13 minutes ago by Laurent Grangeau. A "Cancel running" button is visible.
- Add environment:** Signed-off-by: Laurent Grangeau <laurent.grangeau@gmail.com>
- Jobs:** 5 jobs from codequality (queued for 6 minutes). One job is labeled "Issue" and has a hash "965a10ce".
- Pipeline Stages:** Pipeline Jobs 5. Stages include Build, Test, Quality, Package, and Deploy. The Build stage contains "buildjdk8", the Test stage contains "testjdk8", the Quality stage contains "code_quality", the Package stage contains "package", and the Deploy stage contains "deploy-staging".

Le pipeline se lance à nouveau

Ou avec GitLab et sans le pipeline malheureusement

The screenshot shows a merge request interface with the following details:

- Status:** Checking if merge request can be merged...
- Details:**
 - 9 commits and 1 merge commit will be added to master.
 - Source branch will be deleted.
- Votes:** 0 likes, 0 dislikes.
- Actions:** Sort or filter.

The screenshot shows a merge request interface with the following details:

- Status:** Ready to merge!
- Options:**
 - Delete source branch
 - Squash commits
 - Edit commit message
- Summary:** 9 commits and 1 merge commit will be added to master.
- Buttons:** Merge..., Sort or filter.

The screenshot shows a merge request interface with the following details:

- Status:** Merged by Vel just now
- Actions:** Revert, Cherry-pick
- Summary:**
 - Changes merged into master with 0e12e41b.
 - Deleted the source branch.

```

Dockerfile 964 bytes

1 FROM openjdk:13-alpine
2 WORKDIR application
3 ARG ARTIFACT_NAME
4 COPY ${ARTIFACT_NAME}.jar application.jar
5 RUN java -Djarmode=layer-tools -jar application.jar extract

```

Et la **registry Docker** devrait contenir les nouvelles images buildées grâce au pipeline.

Vous venez de voir à quel point l'**Infrastructure-as-Code** est pratique pour tester rapidement le changement de version d'un **framework**, ou le **changement de version d'un middleware** comme **Apache** ou **IIS**. En ne changeant que quelques lignes, nous pouvons alors relancer tout le pipeline, afin de voir s'il y a un impact sur le code applicatif

Déployez votre application avec Docker Compose

L'**Infrastructure-as-Code** ne s'arrête pas là. Dans le cas de **Docker**, toute l'application peut être **déployée** grâce au fichier **docker-compose.yml** qui contient toute la **définition de l'application**, la relation entre les images Docker et le sens de démarrage de celles-ci.

Le fichier **docker-compose.yml** présent dans le **repository Git** définit des **images en dur**. Vous allez remplacer le nom des images par les nouvelles images que vous venez de créer.

Remplacez alors toutes les lignes contenant `mszarlinski/` par votre nom de registry (chez moi, `registry.gitlab.com/laurentgrangeau/`). De plus, **ajoutez aussi en bout de ligne le numéro de version de l'image que vous venez de créer :2.0.7**

Le fichier **docker-compose.yml** devrait ressembler à ceci (le fichier est volontairement tronqué)

```

1 version: '2'
2
3 volumes:
4   graf-data:
5
6 services:
7   config-server:
8     image: registry.gitlab.com/laurentgrangeau/spring-petclinic-microservices/spring-petclinic-
9       config-server:2.0.7
10    container_name: config-server
11    mem_limit: 512M
12    ports:
13      - 8888:8888
14
15   discovery-server:
16     image: registry.gitlab.com/laurentgrangeau/spring-petclinic-microservices/spring-petclinic-
17       discovery-server:2.0.7
18     container_name: discovery-server
19     mem_limit: 512M
20     depends_on:
21       - config-server
22     entrypoint: ["/.dockerize","-wait=tcp://config-server:8888","-timeout=120s","","","java", "-
23       XX:+UnlockExperimentalVMOptions", "-XX:+UseCGroupMemoryLimitForHeap", "-
24       Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
25     ports:
26       - 8761:8761
27
28 ...

```

Et voilà, vos fichiers Docker et Docker Compose sont prêts à être lancés par votre pipeline de livraison continue.

En résumé

Dans ce chapitre, vous avez simplement **modifié** les fichiers **Dockerfile**, **pom.xml** et **Docker Compose**, pour voir que l'**Infrastructure-as-Code** fonctionne bien sur ce projet. Il vous donne donc un bon exemple de l'intérêt de l'**Infrastructure-as-Code** et de **Docker**, pour **déployer facilement votre application** !

Déployez et testez votre code sur différents environnements

Dans cette partie, vous allez **déployer** les **images** précédemment **buildées** sur un **environnement Docker**, grâce au **pipeline de livraison continue**. Puis, nous lancerons les tests grâce à la méthode **Canary Release**.

Préparez votre environnement de travail

Afin de vous faciliter la tâche et de ne pas installer des dépendances inutiles, je vous conseille de créer un environnement sur le site **Play-With-Docker**. Ce site va vous permettre de créer une **infrastructure Docker** rapidement. Rendez-vous sur le site, et connectez-vous avec vos identifiants **Docker Hub**.

Une fois connecté, **une session de 4 heures est créée** afin de vous permettre de déployer vos images. Sur la page d'accueil, cliquez sur l'icône et sélectionnez le template **3 Managers and 2 Workers**.

Cela va vous créer un cluster **Docker Swarm**, nécessaire au déploiement des images. Une fois le cluster créé, vous allez récupérer l'URL de l'environnement. Il suffit de copier l'URL présente dans la case SSH.

SSH
ssh ip172-18-0-60-ccngfja3icp000fsfddg@direct.labs.play-w

Cette URL sera utilisée pour **configurer l'environnement de déploiement** dans le fichier **.gitlab-ci.yml**. Maintenant, modifiez ce fichier pour ajouter deux nouvelles lignes. **La première ligne à ajouter est au niveau de variables**. Cette nouvelle variable va contenir l'**URL copiée précédemment** (ip172-18-0-51-bihm1906chi000b37l6g chez moi) :

```
variables:  
| PWD: ip172-18-0-60-ccngfja3icp000fsfddg
```

La deuxième ligne est à ajouter **juste après l'étape package**. Cette étape supplémentaire sera le **déploiement des images sur un environnement de staging** :

```

deploy_staging_job:
  stage: deploy
  image: docker:stable
  script:
    - apk add --no-cache openssh-client py-pip python-dev libffi-dev openssl-dev gcc libc-dev make
    - pip install docker-compose
    - export DOCKER_HOST=tcp://$PLAYWD.direct.labs.play-with-docker.com:2375
    - docker-compose down
    - docker-compose up -d
  environment:
    name: staging
    url: http://$PLAYWD-8080.direct.labs.play-with-docker.com

```

La syntaxe est la même que les précédentes étapes. Dans la partie **script**, nous avons ajouté la copie du fichier **docker-compose.yml**, ainsi que le dossier docker. Enfin, nous démarrons le projet grâce à Docker Compose.

Il faut ensuite rajouter une nouvelle étape dans la partie "stages" : **deploy**

```

stages:
  - build
  - test
  - quality
  - package
  - deploy

```

Si tout s'est bien passé, vous devriez voir apparaître dans vos environnements (Opérations > Environnements), le nouvel environnement Staging.

| Available | Stopped |
|-----------|---------|
| 1 | 0 |

| Environment | Deployment | Job |
|-------------|----------------------|--------------------------|
| staging | #17 by [circle icon] | deploy_staging #18984... |

Vous pouvez alors cliquer sur le lien "Open live environment" sur la droite de cet environnement, afin de voir l'application déployée.



Maintenant que l'environnement **Staging** est déployé, il est possible de **lancer des tests impossibles à lancer lors de la phase d'intégration continue**. Dans ce cours, nous allons lancer un **test de performance**, afin de mesurer les temps de réponse de l'application. Pour ce faire, vous allez utiliser **Apache Benchmark** pour simuler de la charge sur le serveur.

Il faut alors **ajouter** de nouvelles lignes dans le fichier **.gitlab-ci.yml**, afin de lancer les tests de performance sur le nouvel environnement :

```
performance_job:
  stage: performance
  image: docker:git
  variables:
    URL: http://\$PLAYWD-8080.direct.labs.play-with-docker.com/
  services:
    - docker:stable-dind
  script:
    - apk add --no-cache curl
    - x=1; while [[ "$(curl -s -o /dev/null -w "%{http_code}" http://\$PLAYWD-8080.direct.labs.play-with-docker.com/) != "200" || $x -le 60 ]]; do sleep 5; echo $(( x++ )); done || false
    - mkdir gitlab-exporter
    - wget -O ./gitlab-exporter/index.js https://gitlab.com/gitlab-org/gl-performance/raw/master/index.js
    - mkdir sitespeed-results
    - docker run --shm-size=1g --rm -v "$(pwd)":/sitespeedio/sitespeed.io:6.3.1 --plugins.add ./gitlab-exporter --outputFolder sitespeed-results $URL
    - mv sitespeed-results/data/performance.json performance.json
  artifacts:
    paths:
      - sitespeed-results/
  reports:
    performance: performance.json
```

```
performance_job:
  stage: performance
  image: docker:git
  variables:
    URL: http://\$PLAYWD-8080.direct.labs.play-with-docker.com/
  services:
    - docker:stable-dind
  script:
    - apk add --no-cache curl
    - x=1; while [[ "$(curl -s -o /dev/null -w "%{http_code}" http://\$PLAYWD-8080.direct.labs.play-with-docker.com/) != "200" || $x -le 60 ]]; do sleep 5; echo $(( x++ )); done || false
    - mkdir gitlab-exporter
    - wget -O ./gitlab-exporter/index.js https://gitlab.com/gitlab-org/gl-performance/raw/master/index.js
    - mkdir sitespeed-results
    - docker run --shm-size=1g --rm -v "$(pwd)":/sitespeedio/sitespeed.io:6.3.1 --plugins.add ./gitlab-exporter --outputFolder sitespeed-results $URL
    - mv sitespeed-results/data/performance.json performance.json
  artifacts:
    paths:
      - sitespeed-results/
  reports:
    performance: performance.json
```

Dans ce nouveau bloc, la **syntaxe reste la même**. Nous récupérons dans un premier temps l'utilitaire de test de performance dans le bloc **script**. Nous lançons ensuite une application qui va se charger de tester notre site et d'en extraire des **métriques de performance**. Ces métriques sont ensuite uploadées sur **GitLab** afin d'être accessibles.

Ensuite, modifiez aussi le début du fichier afin d'ajouter une nouvelle ligne dans le bloc **stages** :

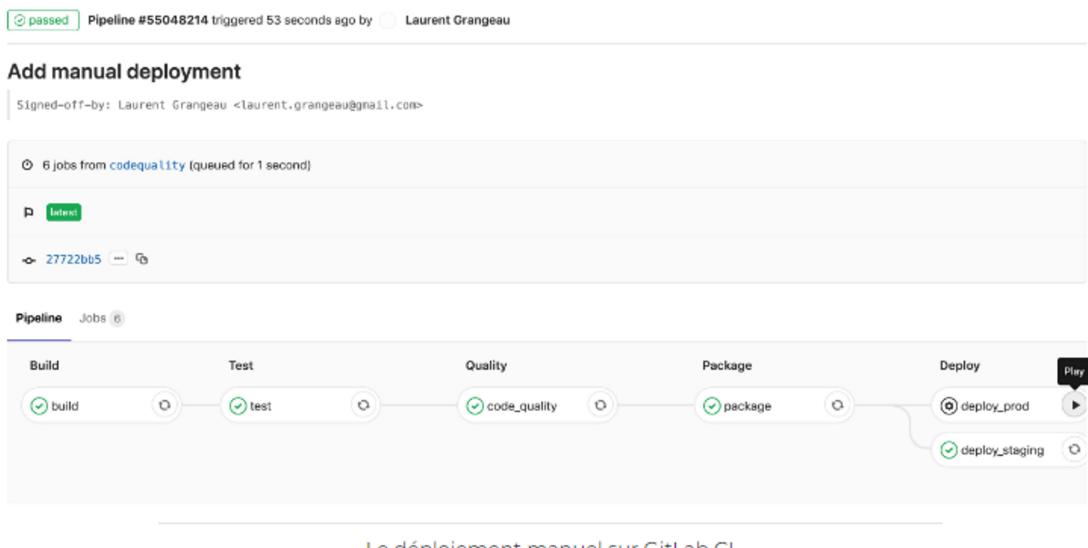
```
stages:
  - build
  - test
  - quality
  - package
  - deploy
  - performance
```

Enfin, une fois l'environnement de staging déployé et testé, il ne reste plus qu'à déployer l'application sur l'environnement de production. Pour cela, vous allez une nouvelle fois modifier le fichier **.gitlab-ci.yml** afin d'ajouter l'étape de mise en production :

```
9      - docker-compose up -d
10 environment:
11   name: prod
12   url: http://\$PLAYWD-8080.direct.labs.play-with-docker.com
13   when: manual
```

Dans cette étape, nous ajoutons le mot clé **when: manual** afin de ne déployer en production qu'avec l'intervention d'un être humain. La validation est requise afin de savoir s'il existe des erreurs lors du déploiement sur staging. Si des erreurs existent, il n'y aura alors pas de mise en production.

Sur votre **pipeline de livraison continue**, le **déploiement manuel** est symbolisé par l'icône ➤ à côté de l'étape **deploy_prod** :



Ces erreurs seront analysées lors de la prochaine étape : le **monitoring**.

Enfin, une technique largement utilisée lors de l'utilisation de la **livraison continue** est le **Canary Release**. Le principe du Canary Release est le même que dans les mines de charbon. À l'époque, les mineurs de charbon qui descendait à la mine plaçaient un canari devant eux, au bout d'une perche dans une cage. Si le canari mourait, cela voulait dire que l'air était non respirable et les mineurs avaient le temps de rebrousser chemin afin d'éviter un sort fatal.

Le principe est le même dans le déploiement : **une partie seulement des utilisateurs vont être redirigés vers la nouvelle version de production**, et si quelque chose se passe mal, il n'y aura uniquement qu'**une petite partie des utilisateurs qui sera impactée**. Pour le mettre en place sur notre projet, modifiez le fichier `.gitlab-ci.yml` en ajoutant un nouveau bloc **canary** :

```
canary_job:
stage: canary
image: docker:stable
script:
- apk add --no-cache openssh-client py-pip python-dev libffi-dev openssl-dev gcc libc-dev make
- pip install docker-compose
- export DOCKER_HOST=tcp://$PLAYWD.direct.labs.play-with-docker.com:2375
- docker-compose down
- docker-compose up -d
environment:
name: prod
url: http://$PLAYWD-8080.direct.labs.play-with-docker.com
when: manual
only:
- master
```

```
canary_job:
stage: canary
image: docker:stable
script:
- apk add --no-cache openssh-client py-pip python-dev libffi-dev openssl-dev gcc libc-dev make
- pip install docker-compose
- export DOCKER_HOST=tcp://$PLAYWD.direct.labs.play-with-docker.com:2375
- docker-compose down
- docker-compose up -d
environment:
name: prod
url: http://$PLAYWD-8080.direct.labs.play-with-docker.com
when: manual
only:
- master
```

Le principe ici est exactement le même que la **production**, la différence étant que le **déploiement en canary est décorrélé de la production**.

Ensuite, modifiez le début du fichier afin que dans le bloc stages soit ajoutée l'étape canary :

```
stages:  
  - build  
  - test  
  - quality  
  - package  
  - canary  
  - deploy  
  - performance
```

Nous avons maintenant un environnement qui se déploie en parallèle de la production, et qui contient uniquement une sous-partie des utilisateurs. Cet environnement sera très utile afin de faire des analyses en temps réel du comportement de l'application, et voir s'il n'y a pas d'erreurs.

Nous avons maintenant un pipeline complet de livraison continue, de la compilation du projet au déploiement sur un environnement de staging, une possibilité de déploiement en production via l'intervention d'une personne de l'équipe d'ops, par exemple, et un environnement Canary qui contient un sous-ensemble des utilisateurs, afin de voir comment se comporte l'application.

En résumé :

Dans ce chapitre, vous avez créé votre environnement de staging sur **GitLab**, afin d'y effectuer vos **tests avant le déploiement en production que vous pourrez faire manuellement via GitLab**. Vous avez également mis en place une étape de test de type **Canary Release**.

Monitez votre application

Afin de savoir si le déploiement d'un système s'est bien déroulé, il est nécessaire de le **moniturer**, ou le **superviser**. Cela permet de prendre des décisions plus rapides, comme le **rollback automatique** d'une application si celle-ci ne fonctionne pas.

Les deux dernières étapes de notre **pipeline de livraison continue** sont donc le **monitoring** de l'application, afin de savoir si celle-ci fonctionne correctement, ou présente des erreurs, ainsi que l'activation de notifications en cas de problème, pour avoir un **feedback rapide**.

Supervisez votre application avec Prometheus

Dans cette section, vous allez ajouter dans **GitLab** la récupération des **métriques de l'application**. Lors du déploiement de l'application, des métriques sont exposées par **Prometheus**, qui est un des composants de notre **stack technique**.

Le serveur **Prometheus** est accessible sur le port **9091**, auquel vous pouvez accéder en cliquant sur le numéro de **port 9091** qui est apparu sur le site **Play-with-Docker** :



Ce serveur Protheameus récupère énormément de **métriques de l'application**, des **systèmes sous-jacents**, ainsi que des **images Docker**, comme le **nombre de connexions par seconde**, le **nombre de connexions total**, etc. Il expose aussi des **métriques applicatives**, comme le **nombre d'animaux créés ou mis à jour**.

Pour récupérer ces métriques dans **GitLab**, il suffit d'aller dans le menu **Settings**, **Integrations**, puis **Prometheus**.

Ensuite, il faut activer l'intégration avec **Prometheus**, et le configurer. Cochez la case **Active**, renseignez l'URL du serveur Prometheus ; dans mon cas :<http://ip172-18-0-8-biiabu86chi000em9j9g-9091.direct.labs.play-with-docker.com/> et sauvegardez :

Nous allons ensuite définir une **métrique** que nous allons suivre, afin de voir si l'**application** à un problème. Cliquez alors sur **New Metric** et ajoutez les informations suivantes :

Laurent Grangeau > spring-petclinic-microservices > Settings > Integrations > Prometheus > New metric

New metric

Name: Requests/second
Used as a title for the chart

Type: Response

Query: http_server_requests_seconds_count
PromQL query is valid

Y-axis label: Requests/second
Label of the y-axis (usually the unit). The x-axis always represents time.

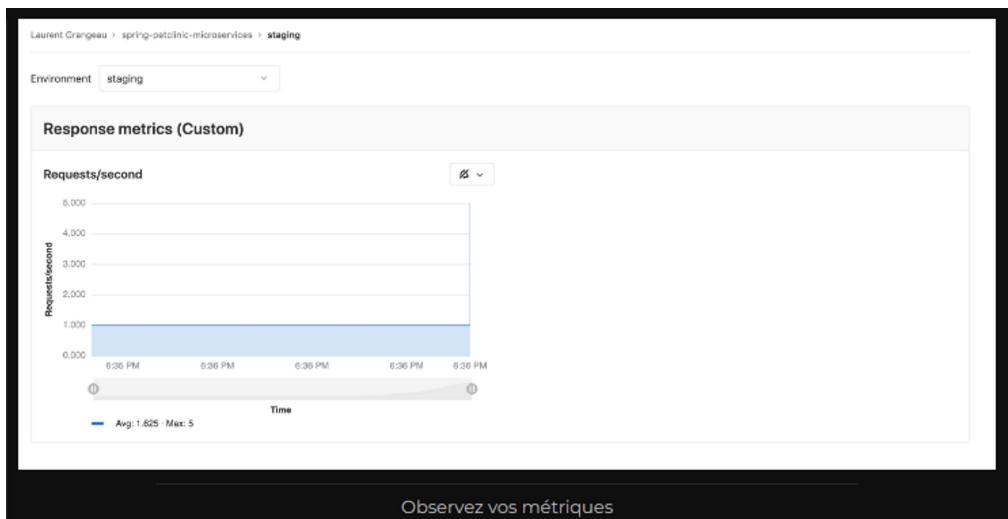
Unit label: req/sec

Legend label (optional): e.g. HTTP requests
Used if the query returns a single series, if it returns multiple series, their legend labels will be picked up from the response.

Create metric **Cancel**

Configurez une nouvelle métrique sur Prometheus

GitLab va alors récupérer la métrique `http_server_requests_seconds_count` depuis le serveur Prometheus, et l'ajouter dans sa base de données interne. Suite à cela, nous pouvons alors voir les graphes de ces métriques dans le menu **Operations**, puis **Metrics** où nous avons l'**évolution des connexions HTTP au fur et à mesure du temps** :



Ces métriques sont très utiles pour prendre des décisions sur le déploiement en production. Ici, nous voyons que les connexions HTTP se font bien, et nous sommes donc confiants sur la mise en production.

D'autres types de métriques sont aussi accessibles via **GitLab**, afin de **prendre des décisions et voir la productivité de l'équipe**.

Par exemple, lors du précédent chapitre, vous avez déployé un environnement **Canary** afin d'analyser le comportement de l'application. Pour voir comment cet environnement se comporte et si celui-ci est viable, allez dans le menu **Operations**, puis **Environments**. Vous devriez voir votre nouvel environnement **canary** et voir les métriques associées :

The screenshot shows the GitLab Cycle Analytics interface. At the top, there are two tabs: "Available 3" and "Stopped 1". On the right, there is a green button labeled "New environment". Below this, a table lists environments: "production" (status: "production-canary (pod 0) Finished"), "Last deployment": "#26 by [user icon]", "Job": "canary #14790396", "Commit": "fa2d2368 [user icon] Update server.rb", and "Updated": "12 minutes ago". To the right of the table are several icons: a play button, a refresh, a copy, a link, and a "Re-deploy" button. Below the table, a progress bar indicates "100%" completion with a series of colored squares. A message "Complete" is displayed below the progress bar. At the bottom, a link "Les métriques associées à Canary" is visible.

Pour voir la performance et la productivité de l'équipe, **GitLab** intègre aussi des métriques concernant le **code**, les **issues**, ou encore le **temps d'exécution des tests**. Ces différentes métriques sont disponibles dans le menu **Project**, sous-menu **Cycle Analytics**.

The screenshot shows the "Recent Project Activity" section of the Cycle Analytics interface. It displays three key metrics: "3 New Issues", "43 Commits", and "7 Deploys". A dropdown menu shows "Last 30 days". Below this, a table provides a breakdown of activity by stage:

| Stage | Median | Related Jobs | Total Time |
|------------|-----------------|--|------------|
| Issue | Not enough data | Total test time for all commits/merges | |
| Plan | about 1 hour | test;jdk8 · #179781426 ↗ monbug ← 92708b95 19 days ago | 3 mins |
| Code | Not enough data | build;jdk8 · #179781424 ↗ monbug ← 92708b95 19 days ago | 1 min |
| Test | 5 minutes | test;jdk8 · #179772673 ↗ 6-job-failed-179747636 ← f2eb7108 19 days ago | 3 mins |
| Review | 21 minutes | build;jdk8 · #179772672 ↗ 6-job-failed-179747636 ← f2eb7108 19 days ago | 1 min |
| Staging | Not enough data | | |
| Production | Not enough data | | |

At the bottom, a link "Métriques Cycle Analytics" is visible.

Les métriques les plus **intéressantes** sont celles qui fournissent des **indicateurs sur la vitesse et la productivité de l'équipe**. Par exemple, il est possible de **voir le temps entre la création d'une issue et sa résolution dans la rubrique Review**.

Il est possible de voir aussi le **temps de déploiement sur les différents environnements**. Plus cette valeur est petite, plus il est facile de déployer sur les environnements. Par exemple, **dans l'exemple ci-dessous, le temps de déploiement sur l'environnement de Staging est de 3 minutes en moyenne. Avec un temps de déploiement aussi court, il est facile de déployer une correction en production assez rapidement**.

Laurent Grangeau > spring-petclinic-microservices > Cycle Analytics

Recent Project Activity

| New Issues | 1 Commit | 7 Deploy | Last 30 days |
|------------|----------|----------|--------------|
| - | 1 | 7 | Last 30 days |

| Stage | Median | Related Deployed Jobs | Total Time |
|------------|-----------------|---|------------|
| Issue | Not enough data | From merge request merge until deploy to production | |
| Plan | Not enough data | #179781425 ⚡ nonbug → 92708b95 about 1 month ago by Laurent Grangeau | 3 mins |
| Code | Not enough data | #179781424 ⚡ nonbug → 92708b95 about 1 month ago by Laurent Grangeau | 1 min |
| Test | Not enough data | #179772673 ⚡ 6-job-failed-179747636 → f2eb71d0 about 1 month ago by Laurent Grangeau | 3 mins |
| Review | Not enough data | #179772672 ⚡ 6-job-failed-179747636 → f2eb71d0 about 1 month ago by Laurent Grangeau | 1 min |
| Staging | Not enough data | | |
| Production | Not enough data | | |

Métrique sur le temps de déploiement

Il y a aussi d'autres métriques qui existent, concernant le **code**. Par exemple, vous pouvez voir le **nombre de commits**, ainsi que les **differents contributeurs** dans le menu **Repository**, sous-menu **Contributors**.

Laurent Grangeau > spring-petclinic-microservices > Contributors

master History

January 9, 2013 – April 2, 2019

Commits to master, excluding merge commits. Limited to 6,000 commits.

Mic
204 commits
[misry@vmware.com](#)

Antoine Rey
97 commits
[antoine.rey@gmail.com](#)

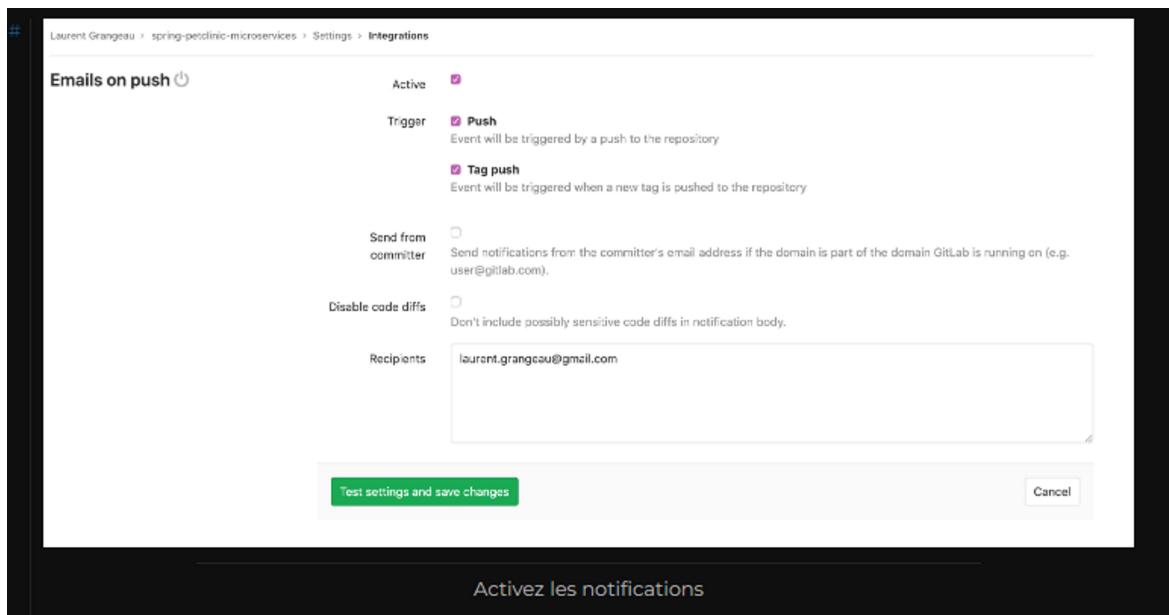
Métriques sur les contributeurs

Il est aussi intéressant de voir le **nombre de commits par jours**, pour évaluer le temps de travail de chaque **développeur**. Cette métrique est disponible dans le même menu, sous-menu Charts.

Mettez en place des notifications Slack

La dernière étape est l'étape de **feedback rapide**. Cette étape est celle qui va nous permettre de faire le lien entre la production (**ops**) et les développeurs (**dev**). C'est une étape qui donne de la **visibilité** aux **développeurs** sur des problèmes qu'il peut y avoir en production. **Plus rapide est la détection des problèmes, plus rapide est leur correction.**

Cette étape est le lien final qui permet d'avoir notre **amélioration continue** durant tout le **cycle de vie** de l'application. **GitLab** permet l'**intégration avec beaucoup d'applications tierces**. L'intégration la plus simple est l'intégration **email**. Afin d'intégrer **GitLab** avec l'email, allez dans le menu **Settings**, sous-menu **Integrations**. Dans ce menu, choisissez "**Email on push**". Sur le prochain écran, cochez la case Active, renseignez votre mail et cliquez sur "Test settings and save changes".



Vous allez maintenant être alerté des différents **commits** qu'il pourrait y avoir sur le **contrôle de code source**. Mais l'**intégration n'est que partielle avec l'email**. Le mieux est d'intégrer un outil comme **Slack** qui prendra toutes les notifications de **GitLab**, et les affichera dans un **channel** dédié à votre application. Pour intégrer Slack, il suffit d'aller dans le menu **Integrations**, et de choisir **Slack Notifications**.

This service sends notifications about projects events to Slack channels.

To set up this service:

- Add an Incoming webhook in your Slack team. The default channel can be overridden for each event.
- Paste the Webhook URL into the field below.
- Select events below to enable notifications. The Channel name and Username fields are optional.

Active

Push
Channel name (e.g. general)
Event will be triggered by a push to the repository

Issue
Channel name (e.g. general)
Event will be triggered when an issue is created/updated/closed

Confidential issue
Channel name (e.g. general)
Event will be triggered when a confidential issue is created/updated/closed

Merge request
Channel name (e.g. general)
Event will be triggered when a merge request is created/updated/merged

Note
Channel name (e.g. general)
Event will be triggered when someone adds a comment

Confidential note
Channel name (e.g. general)

Tag push
Channel name (e.g. general)
Event will be triggered when a new tag is pushed to the repository

Pipeline
Channel name (e.g. general)
Event will be triggered when a pipeline status changes

Wiki page
Channel name (e.g. general)

Activez les notifications Slack

De cette page, vous allez être **invité à créer un webhook Slack** afin de l'intégrer dans votre repository GitLab.

Choisissez un channel dédié à votre application afin de recevoir tous les messages associés. Attention, les messages sont nombreux. Je vous conseille de ne pas l'ajouter dans le channel Général.

Incoming WebHooks

Send data into Slack in real-time.

Incoming Webhooks are a simple way to post messages from external sources into Slack. They make use of normal HTTP requests with a JSON payload, which includes the message and a few other optional details described later.

Message Attachments can also be used in Incoming Webhooks to display richly-formatted messages that stand out from regular chat messages.

New to Slack Integrations?
Check out our Getting Started guide to familiarize yourself with the most common types of integrations, and tips to keep in mind while building your own. You can also register as a developer to let us know what you're working on, and to receive future updates to our APIs.

Post to Channel
Start by choosing a channel where your Incoming Webhook will post messages to.
#gitlab
 New channel created

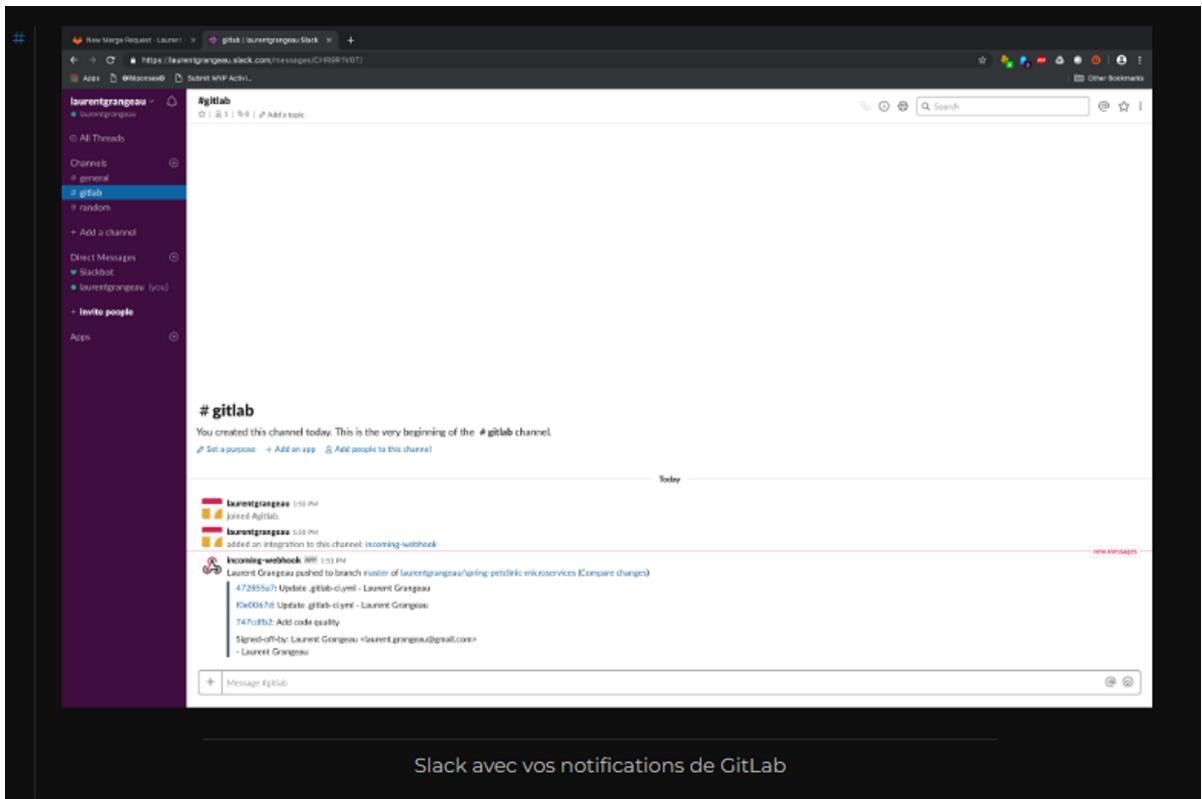
Add Incoming WebHooks integration

By creating an incoming webhook, you agree to the Slack API Terms of Service.

Configurez le channel Slack où recevoir les notifications

Une fois le webhook créé, copiez l'URL délivrée par Slack afin de la coller dans GitLab.

Lorsque l'intégration avec Slack est finie, vous recevrez tous les messages des événements GitLab dans le channel associé.



En résumé :

Dans ce chapitre, vous avez mis en place la **supervision de plusieurs métriques de votre application en production** grâce à **Prometheus** et **GitLab**. Puis, vous avez configuré **GitLab** pour que des notifications d'alerte vous soient automatiquement envoyées sur **Slack** en cas de problème.

Nous avons donc rempli les deux dernières étapes de la **livraison continue** :

- La codification de l'infrastructure avec l'Infrastructure-as-Code.**
- Le déploiement de votre application.**
- Le test de votre application en environnement de test.**
- La supervision de l'application.**
- La mise en place de notifications d'alerte.**

Ce dernier chapitre conclut notre cours sur l'intégration continue et la livraison continue. Vous avez vu tout le long de ce cours comment mettre en place les différentes étapes nécessaires, de la compilation au déploiement en production, en passant par les différents tests possibles et la création d'environnements. Nous avons aussi vu comment gérer la qualité de notre application, afin de maîtriser la dette technique générée par les différents développement et développeurs de notre application.

Question 1

Vous avez assisté à une conférence sur Docker et vous voulez tester si les conteneurs peuvent répondre à votre besoin. Vous devez convaincre votre chef du bien-fondé des conteneurs, et des bénéfices que ceux-ci peuvent apporter.

Quels sont les bénéfices apportés par Docker dans le cadre de la livraison continue ?

Attention, plusieurs réponses sont possibles.

- Les conteneurs assurent que le code produit au sein d'une image fonctionnera, quel que soit son environnement.
- Les conteneurs remplacent les machines virtuelles.
- ✓ Les conteneurs peuvent contenir l'application, ainsi que tout ce qui est nécessaire pour la faire fonctionner.
- ✓ Les conteneurs permettent de fluidifier le travail entre les développeurs et les opérations.

Les conteneurs permettent de simplifier la mise en production, car ils contiennent tout le code de l'application, ainsi que toutes ses dépendances. Ainsi, ils peuvent être facilement transportés du poste de développeur à la production, ils permettent d'être clair sur les dépendances de l'application et garantissent qu'elle fonctionnera aussi bien d'un côté que de l'autre.

Cependant, les conteneurs fonctionnent au niveau applicatif, ils ont besoin d'un système d'exploitation pour tourner. Il ne remplacent donc pas les machines virtuelles ! Ce sont simplement un autre niveau de virtualisation, complémentaire des VM.

Question 2

Votre chef d'équipe de production se plaint de la lenteur pour obtenir des environnements de test pour son application. Ces environnements peuvent parfois prendre jusqu'à plusieurs semaines avant d'être créés. Conséquence de ça : les environnements ne sont jamais libérés et ils engendrent des coûts importants. De plus, l'installation de middlewares est souvent difficile.

Votre idée est de mettre en place de l'**Infrastructure-as-Code**. Quels avantages allez-vous présenter à votre chef d'équipe pour défendre cette solution ?

Attention, plusieurs réponses sont possibles.

- ✓ Les infrastructures peuvent être créées et détruites autant de fois que nécessaire, réduisant fortement les coûts.
- ✓ La création d'environnement est un processus automatique, permettant de n'oublier aucune étape lors de la création.
- ✓ Le code peut être versionné, et garantit l'historique d'installation des environnements.
- L'**Infrastructure-as-Code** amène à une architecture élastique, permettant de doubler la puissance de l'application.

L'**Infrastructure-as-Code** est une manière de coder et versionner, via du code, toute l'infrastructure nécessaire à l'exécution de l'application. Elle permet d'être pilotée par le processus d'intégration et livraison continues. Comme la structure de l'environnement est codé, cela permet de n'oublier aucune étape lors de sa création.

Par contre, l'**Infrastructure-as-Code** n'apporte pas d'élasticité dans l'application. Cette élasticité n'est obtenue que lors du déploiement d'une application sur le cloud.

Question 3

Lors de la création d'un pipeline de livraison continue, plusieurs types de tests peuvent être lancés. Quels sont les tests qui sont lancés lors de la phase de livraison continue ?

Attention, plusieurs réponses sont possibles.

- ✓ Les tests d'acceptance
- ✗ Les tests fonctionnels
 - Les tests unitaires
- ✓ Les tests de performance

Attention, les tests unitaires et fonctionnels sont des tests effectués sur le code juste après sa compilation. Ils ne nécessitent pas d'être sur un environnement proche de la production, ils sont donc effectués lors de l'intégration continue !

Question 4

Votre équipe se plaint que le pipeline de CI/CD mis en place pour leur projet ne fonctionne pas. Le déploiement ne s'effectue pas correctement, et l'équipe voudrait déclencher le déploiement manuellement, à la suite de tests de performance. Voici le code fourni par l'équipe pour le déploiement en production :

```
yaml
1 deploy_prod_job:
2   stage: deploy
3   image: docker:stable
4   script:
5     - scp -r target.project.host.me:/var/www/html
6     - ssh www-data@project.host.me "apache2 restart"
```

Quelles sont la ou les sections que vous leur conseillez d'ajouter au fichier pour résoudre le problème ?

Attention, plusieurs réponses sont possibles.

- ✓ when: manual
- deploy: manual
- when: man_deploy
- ✗ environnement: suivi de l'URL du projet.

L'ajout du mot-clé "manual" permet de ne déclencher le pipeline que grâce à une action manuelle. L'ajout du mot-clé "environment" permet de fournir à GitLab un environnement sur lequel déployer l'application.

Question 5

Vous êtes administrateur d'un site CRM à destination des commerciaux de l'entreprise, appelé www.mes-clients.fr. Ce site permet aux commerciaux de prendre des rendez-vous, de faire de la prospection et résume leur chiffre d'affaires. Sans ce site, les commerciaux ne peuvent pas travailler !

À chaque fois que le site ne fonctionne pas, des centaines d'appels arrivent dans votre équipe afin de rétablir l'accès.

Vous voulez être averti de façon proactive en cas de non-fonctionnement de ce site. Comment procédez-vous?

- Vous mettez en place un job qui fait une requête sur le site toutes les 5 minutes, afin de voir s'il est interrompu ou pas.
- Vous déployez le site sur une machine plus puissante pour garantir son fonctionnement.
- Vous redémarrez tous les jours le site afin de prévenir d'un bug éventuel.
- Vous ajoutez une métrique dans GitLab qui fait des requêtes sur le site en temps réel pour voir sa disponibilité.

Les trois premières réponses ne sont qu'un moyen de contourner le problème et non de le résoudre. Le fait de mettre en place une métrique dans GitLab permet non seulement de voir la disponibilité réelle de l'application, mais aussi de savoir en temps réel que l'application ne fonctionne plus. Cette métrique sera aussi un indicateur tangible, afin de voir l'amélioration de la disponibilité de l'application à travers le temps.

Question 6

Afin de fluidifier la communication entre votre équipe et l'équipe en charge de l'application, vous voulez mettre en place un processus de diffusion des informations sur la vie du projet. Ces informations peuvent être la disponibilité de l'application, les corrections apportées, les commits effectués ou encore le déclenchement d'un pipeline.

- Vous envoyez des emails chaque mois avec les événements importants.
- Vous ajoutez une intégration de GitLab sur votre Slack interne.
- Vous invitez toute l'équipe le soir à partager un moment convivial.
- Vous intégrez l'équipe ops au sein de l'équipe de dev.

L'intégration dans Slack de Gitlab permet de voir tous les événements de la vie du projet et de les historiser, afin de garder une trace de qui a fait quoi. Ces notifications permettent un même niveau d'information pour toute l'équipe, que ce soit pour les devs comme pour les ops.

Question 7

Votre équipe travaille sur une application web de vente de produits de luxe et l'équipe commerciale se plaint des ventes en baisse. Après investigation, vous découvrez que le problème de vente provient du temps de réponse de la page principale du site. Celle-ci met plusieurs secondes à s'afficher, ce qui décourage les potentiels acheteurs.

Votre équipe corrige ce bug, mais vous voulez garder un œil sur les futures évolutions du site. Vous voulez à tout prix éviter de redéployer une version moins performante, c'est-à-dire une régression de l'application.

Comment assurez-vous la qualité des performances du site ?

- Vous déployez en double le site et séparez le trafic en deux sur ces deux sites, pour que la charge soit mieux répartie.
- ✗ ○ Vous ajoutez des métriques en production afin de mesurer les temps de réponse du site en continu.
- ✓ ● Vous ajoutez une étape de mesure de la performance dans votre pipeline qui teste les temps de réponse à chaque redéploiement de l'application.
- Vous enlevez l'étape des tests unitaires de votre pipeline car elle prend trop de temps, et cela permettra d'accélérer le fonctionnement du site.

Doubler le site permettrait d'augmenter le nombre d'utilisateurs présents sur votre site, mais ne résoudrait pas le problème de chargement de la page. Ajouter des métriques en production afin de mesurer les temps de réponse ne servirait qu'à constater que la page met plusieurs secondes à s'afficher, mais ne garantit pas qu'une régression puisse apparaître. Enfin, retirer l'étape de tests unitaires ne ferait qu'améliorer le temps de traitement du pipeline, et dégraderait la qualité de l'application.

La mesure des performances a tout à fait sa place dans l'étape de mesure de la qualité de l'application ! Vous pouvez même aller plus loin : si jamais vous mesurez un temps de réponse trop long, vous pouvez arrêter automatiquement la mise en production. Ainsi, vous redonnez du temps aux développeurs pour améliorer le code ou régler les éventuels problèmes sur cette nouvelle version.

Question 8

Vous devez expliquer à votre équipe pourquoi il est nécessaire de mettre en place de la livraison continue. L'équipe a l'habitude de travailler avec l'intégration continue pour sa compilation et ses tests, mais ne comprend pas bien l'avantage d'utiliser de la livraison continue.

Que leur dites-vous ?

Attention, plusieurs réponses sont possibles.

- ✓ La livraison continue permet de lancer des tests qui ne peuvent pas être lancés sans déploiement.
- La livraison permet d'assurer que l'application n'aura jamais aucun bug.
- La livraison continue est facile à mettre en place et ne nécessite pas de compétence particulière.
- ✓ La livraison continue permet d'avoir un feedback en continu de l'état de l'application en production.

La livraison continue a de nombreux avantages, mais elle est compliquée à mettre en place et demande de connaître de nombreux outils ! De plus, elle permet de tester facilement en production, certes, mais aucun code n'est parfaitement fiable ! Si vos tests ne couvrent pas certains bugs, alors ils peuvent passer en production, même avec un pipeline de livraison continue.

Question 9

Votre entreprise maîtrise déjà l'intégration continue, car un pipeline CI tourne pour l'intégration de toutes les nouvelles features de l'application principale. Vous vous intéressez donc à la mise en place d'un pipeline de livraison continue, et vous faites vos recherches sur les outils à utiliser.

Pour commencer, vous envisagez de mettre en place de l'Infrastructure-as-Code, un déploiement en continu et une supervision de l'application (étapes 1, 2 et 4 de la livraison continue).

Quels outils pourraient vous intéresser pour mettre en place une telle chaîne de CD ?

Attention, plusieurs réponses sont possibles.



- **Docker** pour l'Infrastructure-as-Code ;
- **Spinnaker** pour le déploiement continu ;
- **Prometheus** pour la supervision.



- **Bash** pour l'Infrastructure-as-Code ;
- **FTP** pour le déploiement continu ;
- **SNMP** pour la supervision.



- **Hyper-V** pour l'Infrastructure-as-Code ;
- **Git** pour le déploiement continu ;
- **Nagios** pour la supervision.



- **KVM** pour l'Infrastructure-as-Code ;
- **xcopy** pour le déploiement continu ;
- **Nagios** pour la supervision.