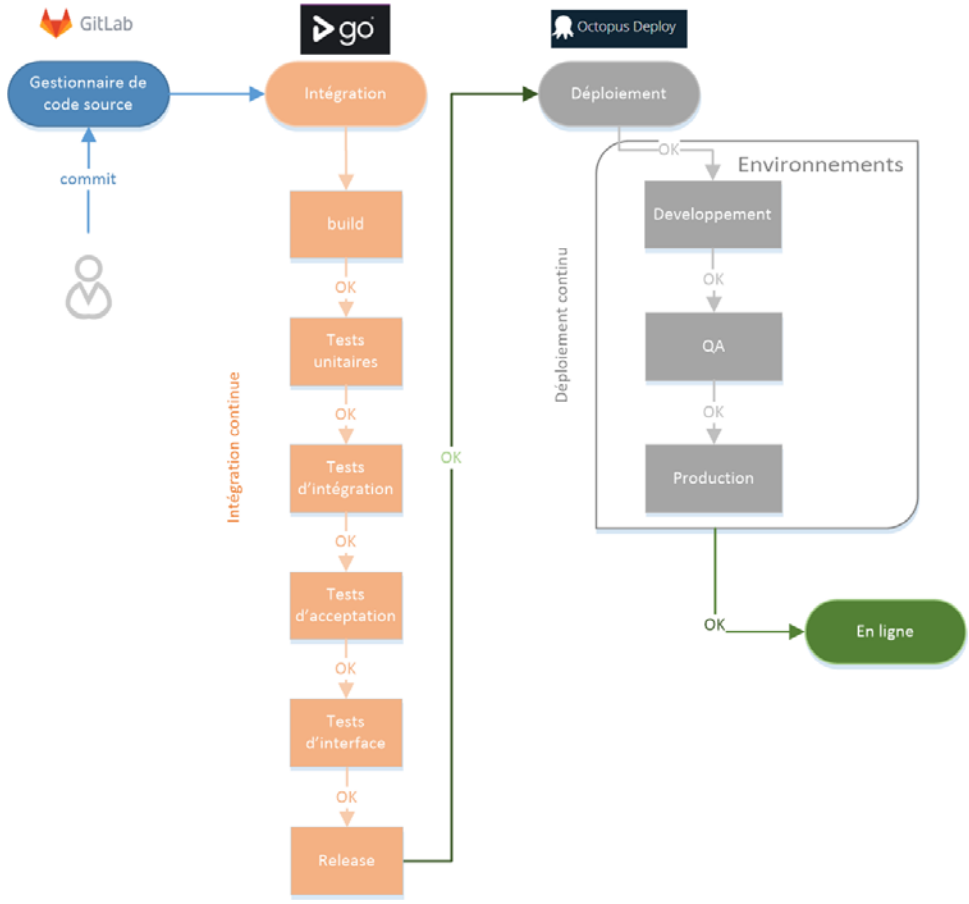
Chapitre 5
Intégrer en continu

1. Introduction

L'intégration en continu est une pièce centrale de tout système CI-CD. Elle a pour but de traiter chaque changement effectué sur le dépôt source et d'enclencher les mécaniques de vérification, de compilation et de tests. Lorsqu'une modification traverse avec succès l'ensemble de ces filtres, le système est alors en mesure de pousser la nouvelle version packagée vers un système de déploiement en continu.



Tout l'art consistera donc à intercepter au maximum les éventuels dysfonctionnements introduits par un changement dans le code. Il faut donc que chaque membre de l'équipe travaille pour atteindre cet objectif. Cela demandera d'une part que vous ayez défini de bonnes pratiques de développement, mais également octroyé **suffisamment** de temps projet au développement des tests. C'est primordial. Un CI-CD n'a de sens que si des tests **poussés** sont mis en œuvre ! Si vous décidez de faire l'impasse sur les tests, vous augmenterez le risque que le système de filtrage devienne trop permissif. Vous augmenterez donc la probabilité d'avoir un bug en production.

Prenons l'exemple d'une voiture moderne. Lorsque vous mettez le contact, une série de tests est lancée sans même que vous vous en rendiez compte. Ces tests ont pour but de détecter si des composants majeurs du véhicule sont toujours en bon état de fonctionnement. Ils peuvent par exemple vérifier la bonne pression des pneus, le niveau d'huile, le niveau d'essence, le bon fonctionnement des airbags, etc. Lorsqu'un problème est détecté, vous aurez probablement sur votre tableau de bord une alerte. La couleur orange indique souvent une alerte qui vous dit : "Attention vous pouvez démarrer le véhicule, mais un des éléments doit être corrigé rapidement". La décision vous incombera donc : démarrer le véhicule ou effectuer la réparation avant. Mais si vous décidez de le démarrer, ce voyant constamment allumé continuera de vous rappeler à l'ordre. Il vous faudra à un moment ou à un autre effectuer la réparation.

Un voyant rouge est plus embêtant. C'est l'avertissement qu'un des composants importants de la voiture est en panne. Cela est peut-être dû à un dysfonctionnement dans les airbags, le moteur, le détecteur d'arrêt d'urgence ou le circuit de refroidissement. Démarrer la voiture et la conduire augmenterait de façon très grande le risque de panne ou d'accident. Dans certains cas graves, la voiture ne démarrerait même pas !

Cet exemple introduit plusieurs principes que nous détaillerons dans les chapitres suivants dédiés aux tests.

Premièrement, plus nous avons de tests au démarrage (et pendant la conduite bien évidemment) et plus nous augmenterons notre dose de confiance vis à vis du véhicule que nous conduisons. Si notre voiture ne testait que le liquide de refroidissement, cela ne serait pas suffisant pour nous donner confiance sur **l'ensemble** du véhicule parce que le périmètre de vérification serait trop restreint.

Deuxièmement, il y a différents niveaux de criticité dans les tests. Certains ont une importance plus grande que d'autres. Le tout est de savoir **catégoriser** la levée d'alertes. C'est en effectuant ce type d'analyse que vous serez en mesure de dire quels aspects de la solution doivent être testés en priorité. Cela veut dire que vous devrez arbitrer le degré de profondeur de tests des domaines fonctionnels de votre application.

Par exemple, on comprend aisément que le système de paiement de votre application comportera probablement beaucoup plus de tests que la page de présentation d'un produit par exemple. Pourquoi ? Parce que l'impact d'un dysfonctionnement serait plus dommageable sur le premier que le second.

Nous avons mentionné dans les chapitres précédents qu'il n'est pas toujours possible d'allouer un temps important au développement des tests. Même s'il ne faut pas faire l'impasse dessus (nous l'avons bien compris) en fonction des contraintes de temps et de budgets qui vous sont imposés, vous devrez définir avec votre équipe ce qui doit en priorité être testé et ce qui relèverait plus d'une alerte orange pour vous.

Une des manières d'arbitrer ce qui doit être testé ou non est de savoir quel serait le coût de son dysfonctionnement en production. Reprenons l'exemple d'un système de paiement. Si vous possédez un site e-commerce, la durée pendant laquelle le système de paiement ne serait pas opérationnel peut vous donner son coût de dysfonctionnement à la minute/heure/jour en fonction de vos statistiques de ventes à la journée. Ce coût devra donc être mis en perspective avec le coût que vous devriez allouer au développement de ses tests et aux économies réalisées.

Si ce coût de dysfonctionnement est dérisoire, peut-être que certains tests pourraient suffire. Dans le cas contraire, si suite à une panne son coût monte en flèche (et de façon exponentielle), alors vous saurez que le temps passé au développement des tests serait vite amorti.

Il vous faudra aussi essayer d'approximer le niveau de panne qui pourrait vous être fatal. Par exemple, si vos contrats introduisent des pénalités lorsque votre système ne répond plus alors vous devriez être en mesure d'évaluer le montant qui pourrait mettre en péril votre activité parce que le degré de pénalités serait tel qui vous serait économiquement insurmontable. Ou alors ce serait peut-être en effectuant le calcul du manque à gagner que vous pourriez avoir une idée de la partie sur laquelle vous devriez concentrer votre attention.

Là encore, c'est en cas de dysfonctionnement que l'on comprend l'importance de pouvoir délivrer rapidement les corrections en un clic. Imaginez qu'il vous faille 2 heures pour corriger un bug, 2 heures pour déployer à nouveau votre solution et 1 heure pour mettre à jour vos bases de données. Cela équivaudrait à 5 heures d'arrêt. Un CI-CD opérationnel devrait vous faire économiser au moins 50 % de ce temps.

Ces différentes approches vous permettent de contre-balancer le ratio coût de développement des tests sur l'impact des dysfonctionnements en production. Un système d'intégration en continu doit assurer ce rôle de vérification que nous avons illustré avec le véhicule. Il doit vous donner suffisamment confiance que votre solution ne comporte pas de bugs majeurs.

Nous allons mettre en place un exemple d'un tel système dans les chapitres qui suivent. Vous devriez être en mesure de comprendre les briques qui composent un système d'intégration en continu et d'appliquer ces principes à votre propre organisation.

2. Mise en place d'un système CI : GoCD

Il existe de nombreux systèmes dans le marché qui répondent à vos besoins. On peut dire que vous avez dans ce domaine l'embarras du choix. Certains sont issus de la communauté du logiciel libre, d'autres sont semi-gratuits (ou semi-payants ça dépend comment on l'entend) et d'autres totalement payants, le curseur étant lié au degré de service qui leur sont attachés.

Ces systèmes sont souvent destinés à des environnements en particulier. Certains ne s'installent que sur une machine Linux, d'autres que sur du Windows. D'autres heureusement sont mixtes. Il vous faudra prendre un peu de temps pour savoir quel système répondra le plus à vos besoins.

Nous avons installé GitLab-CE. Ce système comporte son propre système d'intégration en continu. Mais dans le contexte de ce livre, nous opterons pour l'installation d'un autre système : GoCD. Bien entendu, nous pourrions faire l'ensemble de nos exemples sur GitLab-CE ou un autre système. Mais en optant pour GoCD, nous introduirons d'autres techniques qui étofferont un peu plus votre connaissance et diversifieront vos approches.

3. Installation et paramétrage

Passons maintenant à l'aspect pratique de ce chapitre. Nous avons plusieurs possibilités pour installer notre serveur d'intégration. GoCD peut s'installer indifféremment sur du Windows ou du Linux. Nous allons dans le cadre de ce livre l'installer sur une machine Azure Ubuntu pour que vous puissiez continuer à tester gratuitement.

Nous pourrions télécharger les binaires et faire l'installation en ligne de commande ou sélectionner un template sous Azure. Nous allons procéder différemment cette fois en vous présentant l'approche Docker.

3.1 Présentation de Docker

Docker est un système en vogue dans le monde DevOps. Historiquement, la montée en puissance des serveurs a fait émerger des systèmes de virtualisation comme VMWare ou encore HyperV. L'approche était donc de créer au sein d'un même serveur plusieurs machines virtuelles, puis de dédier ces machines virtuelles à un fonctionnement particulier : telle VM servirait pour le serveur SQL, telle autre comme serveur de messagerie, etc.

Le problème c'est le "gaspillage" des ressources que cela induit. D'une part, cela oblige à installer autant de systèmes d'exploitation qu'il y a de machines virtuelles. Nous ne parlons pas du coût des licences que cela suppose également (pour le monde Windows en premier, mais aussi Linux si nous optons pour des OS payants). Ces OS prennent de la place en disque et ont besoin d'un minimum de mémoire et de processeurs pour fonctionner. Autant de ressources qui auraient pu n'être allouées qu'une fois à la machine source et être libérées pour nos applications métier. Il faut aussi pouvoir gérer des parcs entiers de machines physiques, ce qui ajoute à la lourdeur du système.

Docker aborde les choses de façon différente en introduisant l'idée de conteneur. Cela ne remplace pas le besoin de virtualisation bien entendu, mais on pourrait qualifier cette approche de complémentaire (plutôt que concurrente) à celle-ci.