Introduction :

Dans un monde où les services informatiques se retrouvent à être le socle fonctionnel d’une grande partie de la société moderne, les attentes et les standards qui en découle deviennent de plus en plus élevé. Le temps ou la complexité d’un système était une excuse pour son mauvais fonctionnement sont révolus.

Mais dans une société qui évolue rapidement, maintenir et améliorer des systèmes à créer une myriade de profession et spécialisations. La tâche est d’autant plus ardue qu’il est attendu que tout se passe avec sans interruption de service pour les utilisateurs.

Aujourd’hui, les équipes n’ont plus que rarement à se soucier d’un jour me mise en production de leur nouvelle version, la culture de développement à évoluer, le reste de la chaine de mise en service s’y est plier et de nouveaux outils sont nés.

Dans ce travail, je tâcherai d’expliquer ce qui a amené le monde du logiciel à adopter une approche d’intégration et de développement continu. Les étapes nécessaires à la mise en place d’une architecture permettant de réaliser cette approche seront discutées. Un environnement de déploiement de logiciel sera mis en place en utilisant les services discutés. Le but étant d’arriver à une solution respectant des critères de réutilisabilités et respectant les bons principes de ce qu’on appelle aujourd’hui le DevOps.

Suivant les besoins liés à un projet, une chaine de déploiement peut devoir inclure un très grand nombre de services. Ce projet vise à se concentrer sur les besoins transversaux que pourrait avoir la grande majorité des projets de développement cherchant à mettre à disposition un service en ligne.

**Historique de développement et de mise en production d’applications**

**// ADD historic line and schema**

Les pratiques de développement ainsi que les architectures des logiciels ont énormément évolué ces dernières années. Il est important que comprendre que les deux sont extrêmes corrélées.

Lorsque le développement d’application moderne à commencer à exister, l’ensemble des technologies d’un projet était utilisé ensemble pour fournir le service souhaiter. Le terme ensemble est même quelque peu léger, les technologies devaient être configurée et utilisée comme un seul procédé unifié. Cette façon de fonctionner était très lourde et comportait de nombreux désavantages. Le processus de développement appeler « Waterfall » ou en cascade y est pour beaucoup. Le principe est simple, chaque étape du processus est dépendant de celui qui le précède. Tout problèmes apparaissant plus loin dans la cascade des étapes ne pouvait que difficilement être adressé plus loin qu’un retour d’une étape plus haute.

Les développeurs rendaient une version du logiciel complète supposée fonctionnelle souvent disponible sur un support dédié. //

Cette façon de fonctionner rencontra assez rapidement un nombre de problèmes conséquent dans une société ayant des besoins évoluant rapidement. Bien que les logiciels livrés puissent fonctionner correctement, il s’avérer compliqué de modifier ceux-ci en profondeur si le besoin venait à évoluer. L’utilisateur final réel du logiciel ne pouvait que très peu le tester avant d’en avoir une version complète. Une durée de plusieurs année pouvait être nécessaire avant d’obtenir une version fonctionnelle « application delivery lag » \* . //fi

Les technologies évoluant rapidement, et les interconnexions entre les systèmes devenant de plus en plus nécessaire, avoir une application isolée et difficilement modifiable n’était pas une solution cohérente pour permettre de répondre au besoins des utilisateurs.

Dans cette optique, les parties intégrantes d’un logiciel ont été petit à petite découplé et ainsi commencer à dialoguer entre elle sans entre couplée par l’intermédiaire d’interfaces. C’est cette même philosophie qui a amené les pratiques à évoluer dans les mêmes directions pour l’ensemble des composants d’un logiciel. Dans le début des année 2000’s \*, arriva le manifeste de l’agilité.

Dans cette optique d’acceptation du changement constant, le monde du développement devait s’adapter et avec lui les outils nécessaires apparurent.

L’agilité se focalisant sur les processus avec une création incrémentale et récursive, amenant avec elle le besoin de pouvoir délivrer ces changements au produit utiliser facilement et rapidement. Ce changement de paradigme força aussi la façon de construire les logiciels à changer et à s’adapter. C’est ainsi que l’architecture général des logiciels, leur cycle de déploiement et de vie on fait peau neuve.

MONOLITH vers MICROSERVICES

Comme énoncé précédemment, lorsque le développement logiciel à fait son apparition, une application était constituée d’un gros bloc et pouvait essentiellement être pensé comme un seul projet unique contenant l’ensemble du code nécessaire pour faire marcher toutes les par fonctionnalité du logiciel. Cette architecture porte le nom de « Monolithe »\* .mais avec un nombre de fonctionnalités croissant et une interconnexion avec d’autres logiciel, ce bloc unique pose de nombreuses limitations si le logiciel souhaite évolué.

Toutes les fonctionnalités et par extension le service qui se charge de la faire marcher sont fortement couplée, en effet vu qu’il s’agit d’un seul projet, une modification sur un service peut avoir un effet péjorant sur un autre. Ce qui en plus des bugs amène un niveau de complexité croissant au logiciel lorsque celui-ci grandi.

La mise en production d’un projet à l’architecture monolithique comportait également quelques obstacles qui avait la fâcheuse tendance de se montrer qu’une fois le logiciel mis en production. En effet, le processus n’était pas bien compliqué, il s’agissait de packager le projet sous la forme d’un gros artefact au format souvent propre au langage et de le déposer sur le serveur de production et le tour est joué. Le problème étant que l’environnement du serveur de déploiement était configuré manuellement et n’était pas du tout le même que celui de développement voir même différent de celui de test pour des questions de coût.

Il était donc monnaie courante de se retrouver pour les équipes à devoir tout lâcher pour régler un problème en production surtout si celui-ci n’était pas réplicable dans d’autres environnements.

Les logiciels étant de plus en plus porter sur une utilisation à travers le web, le problème de pouvoir supporter la demande croissante des utilisateurs se trouver être problématique pour une application en Monolithe. En effet, une fois la capacité maximum d’un serveur atteint, il serait possible d’en faire tourner un second puis un troisième et ainsi de suite, puis d’équilibrer la charge entre les serveurs. Cette option pose de nombreux problème suivant le type d’applications notamment si des données doivent être disponible pour chaque utilisateur, celui-ci devra toujours utiliser le même serveur. Cette architecture est également gourmande en ressource, car il est impossible d’augmenter les ressources pour un seul service du logiciel, il faut provisionner un serveur étant capable de tenir si tous les services sont utilisés au maximum de leur capacité en même temps.

Avec tous ces désavantages, il a été rapidement compris que cette architecture ne permettrait pas du tout de proposer des logiciels pouvant évoluer et s’interconnecter facilement et encore moi qu’il soit utilisable par un grand nombre d’utilisateur simultanément.

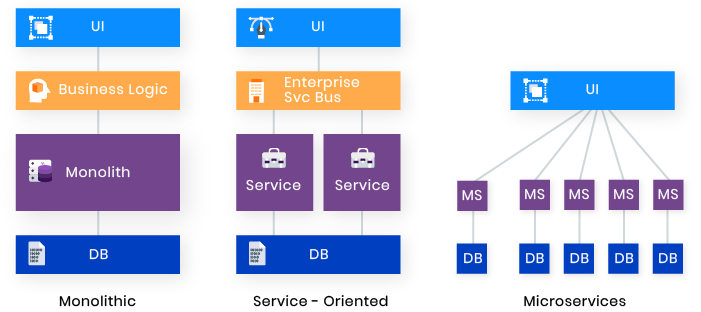
C’est pour cela que l’industrie à rapidement changer de cap est à commencer par découpler les services et les rendre suffisamment petit pour que leur compréhension, amélioration et maintenance soit facilité. Les microservices étaient nés.

Figure httpsrubygarage.orgblogmonolith-soa-microservices-serverless

Attention, cela ne veut pas dire qu’une architecture monolithique n’a pas sa place encore aujourd’hui. Suivant la complexité du projet et/ou la quantité d’utilisateur voués à utiliser le service, une architecture plus simple peut tout à fait faire sens.

Les microservices semblait résoudre beaucoup de problème sur le papier mais comme tout changement, il amenait avec lui de nouveaux problèmes. Avoir une miriade de petits services simplifiait grandement le développement de ceux-ci mais leur mise en production elle était devenue beaucoup plus complexe. En effet, chaque service demandait son lot de dépendances et configuration, maintenir des environnements de développement, de test et de production propre à chaque service était un casse-tête. Cela couplé aux releases fréquentes imposées par les nouvelles normes de développement, le goulet d’étranglement qu’était la mise en production imposa un besoin d’amélioration et d’automatisation.

CI/CD

Jusqu’à présent, le terme mise en production a été utilisé. Il est clair qu’avant qu’une version de logiciel puisse être mise à disposition pour son utilisation celle-ci doit passer à travers d’un certain nombre d’étapes. Chacune d’entre elle ayant pour but essayant de garantir au maximum que la version ne contient pas de bug, soit conforme à un niveau de qualité souhaité et surtout soit fonctionnelle.

C’est dans cet esprit qu’une chaine d’action en grande partie automatisé a vu le jour sous le nom de pipeline d’intégration et de distribution continue.

*« Un pipeline CI/CD est une série d'étapes à réaliser en vue de distribuer une nouvelle version d'un logiciel. Les pipelines*[*d'intégration et de distribution continues (CI/CD)*](https://www.redhat.com/fr/topics/devops/what-is-ci-cd)*désignent une pratique qui consiste à améliorer la distribution de logiciels »*

<https://www.redhat.com/fr/topics/devops/what-cicd-pipeline>

Le mot pipeline est utilisé car de manière image, nous allons avoir un tuyau qui transportera notre nouvelle version logicielle de la fin de sa création à sa mise en production. Comme nous le verrons ce projet sera triturer minutieusement à chaque étape d’avant d’être jugé acceptable pour continuer son chemin dans le tuyau.

Les termes intégrations et distribution continues, sont souvent couplés et acronymisés par leur version anglaise : CI/CD pour Continuous Integration /Continuous Delivery. L’acronyme englobe très souvent également Continuous Deployement(Déploiement continu).

La partie CI se focalisera sur récupérer le projet une fois le pipeline déclencher ainsi que la partie test pré-déploiement.

La partie CD se focalisera sur pousser le projet test sur son environnement de production et les tests post déploiement.

Le pipeline dans son ensemble récupérera le projet mis à jour par le développeur, il construira l’environnement attaché pour vérifier sa bonne construction, passera un certain nombre de test sur le code, déploiera ensuite sont le projet sur son environnement de staging. Il sera ensuite testé sous sa forme complète une fois validé, il sera mis en production. Le but est de fournir un feedback à chaque étape et ainsi permettre au personne responsable de régler le problème.

Les étapes peuvent varier mais nous reviendront sur les étapes de chacune des parties plus précisément plus loin.

DEVOPS//RAJOUTER PRINCIPES CALMS

Une notion importante à ajouter pour comprendre l’utilité et la réflexion derrière l’ensemble du processus.

Pendant de nombreuses années, les développeurs se chargeaient de coder le logiciel et les ingénieurs systèmes de mettre en place et maintenir les environnements de productions ou plus généralisé opérationnels.

Comme très souvent lorsque deux rôles doivent s’appuyer sur le travail de l’un et de l’autre, l’intersection pose un problème, qui a la responsabilité de quoi et jusqu’à où. En d’autres termes, les développeurs travaillant avec leur environnement local de développement ou un serveur dédié rempli avec mille et une dépendances souvent de le but de tests ou pour améliorer le projet, ils fournissent un projet pour eu fonctionnel sans trop se soucier de la suite. A l’inverse, les responsable des systèmes préparent des environnements bien pensés, et construit pour être sécure et maintenable. Lorsque le projet passe des mains des premiers au second, il arrivait très souvent que les dépendances ou configurations requises par le projet pour fonctionner ne soit pas les même que celle dont les environnements préparer disposent. Surtout lorsque l’on parle de développement continu, très souvent il peut même s’agir d’oubli de passage d’information.

Afin d’harmoniser le cycle de développement, il fallait joindre les deux mondes. C’est ainsi qu’est né le DEVOPS, jonction entre le monde du développement et des opérations. Le terme est souvent utilisé dans un sens très large englobant les pratiques de CI/CD, ainsi qu’un grand nombre de procédés et pratiques en lien avec la livraison de logicielles et les changements d’infrastructure résultant de leur utilisation.

*« Que signifie DevOps pour les équipes ? DevOps permet la coordination et la collaboration des rôles autrefois cloisonnés (développement, opérations informatiques, ingénierie qualité et sécurité) pour créer des produits plus performants et plus fiables. En adoptant une culture DevOps ainsi que des pratiques et outils DevOps, les équipes peuvent mieux répondre aux besoins des clients, accroître la confiance suscitée par les applications qu'elles développent, et atteindre plus rapidement les objectifs de leur entreprise. »*

*https://azure.microsoft.com/fr-fr/resources/cloud-computing-dictionary/what-is-devops*

Ces pratiques et façons de penser ont le mérite d’avoir grandement facilité le bon fonctionnement du monde informatique. En se basant sur des technologies comme la conteneurisation ainsi que des outils dédies, le pratique dites DEVOPS ont permis de soutenir l’évolution du développement et de la livraison de logiciel en permettant que l’ensemble des étapes se déroule sans accros et apporte une vraie plus-value au produit final.

C’est d’ailleurs pour ceci que le terme DEVOPS est souvent associé à une double boucle liée ou le symbole infini. Le croisement représentant l’intersection de milieu du développement et des opérations, chaque boucle comprenant les étapes de la vie d’une release logicielle lié à son milieu. Les professionnels liant leur activité à cette idéologie se chargeront de faire la jonction entre les deux mondes mais ne feront vraiment ni du développement ni de la maintenance de système.

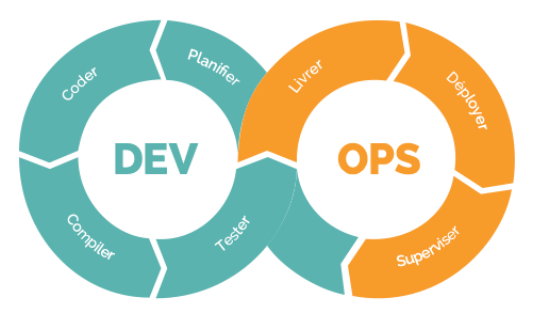


Figure https://blog.ippon.fr/2020/07/29/le-devops-explique-a-mes-parents/

CONTENEURISATION :

Comment il l’a été énoncé, un des problèmes dont souffrait le cycle de développement logiciel était la différence dont souffrait les différents environnements à travers lesquels le logiciel passait. Une première tentative aura été d’utiliser des machines virtuelles pour régler ce problème.

Une machine virtuelle (abrégé VM pour Virtual Machine) offre l’avantage d’être très portable en comparaison avec un serveur dont la configuration sera plus difficile copiable et réutilisable. C’est pourquoi, elle semblait être une bonne façon de fournir à tous les métiers, un artefact pouvant être modifier et réutiliser pour chacune des étapes.

Même si les VM(s) sont un outil excellent et qui permettrait de régler une bonne parti des problèmes liés aux différences d’environnement, elles souffrent encore de certains problèmes. Une machine virtuelle doit avoir des ressources précises lié à son hôte qui lui sont allouée à sa consommation, de pars son OS et ses dépendances elle a déjà beaucoup un poids et une consommation non négligeable. Sans compter qu’une VM dédiée à un environnement de développement sera configurée pour gérer les besoins des développeurs mais ceux-ci voudront tester de nouvelles choses, ajouter des librairies et autres dépendances, ce qui comme avec les serveurs de développement créera sur le terme un monstre contenant plus qu’il n’en faut. Il est tout à fait possible de faire attention et logger tout changements apporter afin de pouvoir les supprimer si besoin. Mais il n’en reste que cela n’est pas très automatisable et très prompt aux erreurs humaines. Sans compter que dans un monde de micro-service, il est très important de pouvoir conter sur des environnements légers au risque de voir ses coups exploser et ses performances amoindries.

Un concept est venu régler ces soucis, la conteneurisation. Le principe est assez simple, Un outil permettant d’automatiser la création d’un environnement possédant le minimum pour faire fonctionner chaque application. En parlant de minium, il s’agit bien sûr de tout ce dont le programme a besoin pour accomplir ses tâches même si comme nous allons le voir il faudra comme pour une machine virtuelle un système opérateur comme base pour y installer le reste. L’avantage de cette manière de faire un paquet avec ce qui est nécessaire, c’est que celui-ci est autonome, il pourra donc être installer et fonctionner indépendamment de l’infrastructure ou de l’OS de son hôte. Il pourra fonctionner comme à l’intérieur d’une bulle et ne laissera rentrer et sortir que ce qu’on lui demande.

Cette bulle portera le nom de conteneur, le nom se rapportant à un conteneur de transport convient à merveille car notre conteneur est fait pour être léger et transportable.

Comme nous le verrons ce concept convient parfaitement aux micros-services et à l’utilisation de ressources sur demande. Car notre conteneur ne contient en général rien de persistant. Il peut être utiliser et ainsi créer son environnement, y faire tourner le programme lié et ensuite être détruit. Il est donc parfait pour rapidement créer des nouvelles instances de supports si un service est en demande accru et à l’inverse, les détruire si ce n’est plus le cas.

Figure httpsmedium.comhackernoonwhat-is-containerization-83ae53a709a6

Dans notre cas, il permet de fournir un environnement évolutif et léger à l’ensemble de notre chaîne de production. Il permettra donc à nos développeurs de modifier leur environnement comme bon leur semble tout en pouvant le passer à la production sans que cela alourdisse ou complexifie le projet.

La conteneurisation à permis de grandement réduire les ressources demandée pour faire fonctionner des projets, c’est aussi cela qui a accéléré la transition en direction du cloud. La majorité des providers de IaaS profite de la flexibilité de qu’offre l’ensemble de chaine de déploiement pour mutualiser les ressources à des prix attractifs.

LE CLOUD

Une fois affranchie de la maintenance de serveur capricieux à maintenir et spécifiquement configurés pour le déploiement de leurs applications, la priorité des entreprises s’est fixée sur l’optimisation des coûts. Maintenir des serveurs dédiés qui plus est sur le site de l’entreprise est très coûteux sans compter qu’il faut provisionner l’infrastructure comme pouvant supportant la demande maximale prévu, ce qui n’est jamais le cas 24/24h. Fini également la maintenance sur site. Le Cloud prenait son envol.

Dans notre cas, la partie très intéressante qu’offre une solution d’hébergement dite cloud, est que le prix est calculé en fonction de l’utilisation ce qui nous permet en configurant bien les outils, il est possible de provisionner uniquement (ou presque) les ressources nécessaires. Ce qui soit dit en passant est une bonne chose pour notre planète.

IAAS VIA API -> Terraform + Ansible

RESUME

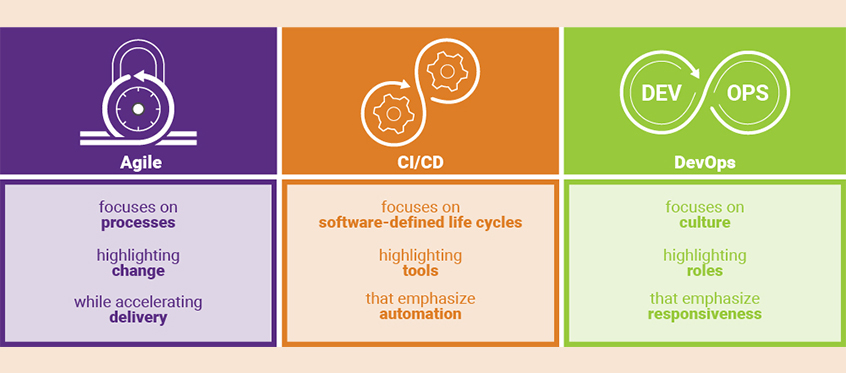


Figure 4https://www.synopsys.com/blogs/software-security/wp-content/uploads/2018/03/differences-wp.jpg

PROPOSITION DES ETAPES DU PIPELINE ET OUTILS :

Introduction :

Il est clair qu’un pipeline sera différent d’un projet à un autre. Ce qui peut vite devenir une grande quantité de travail pour un service devant en maintenir un certain nombre. C’est pourquoi il est préférable de reste sur une architecture aussi proche de l’agnostique que possible.

Il serait même intéressant d’aborder l’idée d’un pipeline qui puisse être utilisable peu importe le langage utilisé pour le projet. Cela n’est pas aussi complique qu’il puisse le paraitre mais demande de se reposer sur une grande quantité de scripts Bash ou autre suivant l’OS. Il n’en est pas moins que cette possibilité est attirante surtout dans une idée de micro-services qui pour une liberté de conception et dans le but d’être aussi performant que possible seront dans un grand nombre de cas développé dans des langages différents. Nous reviendrons un peu plus en détail sur cette possibilité au fur et à mesure que nous avançons dans la description du pipeline.

CRITERES DE CHOIX DES SOLUTIONS

Afin d’avoir une structure de pipeline qui soit portable sur n’importe quel architecture ou hébergeur, il est préférable d’éviter des solutions développer par les revendeurs de solution d’hébergement. N’étant pas une société disposant de moyens conséquents, nous nous concentrerons sur des solutions open sources. Elles ont l’avantage d’être gratuite et supporter par une communauté ce qui nous offrira également une base de support gratuite.

Le pipeline que nous allons essayer de mettre en place, ne sera surement pas le plus complet possible mais nous essaierons de sa complexité de mise en place et d’utilisation à un degré raisonnable pour éviter de se retrouver avec un monstre dont nous ne maitrisons pas la complexité.

Nous nous assurerons également d’utiliser des solutions qui ont prouvé quelle aller survivre dans le temps, qui sont reconnu dans par la communauté du monde informatique et qui sont utiliser communément.

Afin d’éviter de tomber dans la complexité de services distribué, nous allons rester sur un déploiement limité avec une possibilité de répondre à l’augmentation de la demande mais dans une limite. La mise en place du dit pipeline demande déjà de toucher un grand nombre de technologies y ajouter une capacité à pouvoir support une augmentation de charge conséquente requière encore une autre expertise et risque de générer plus de problèmes qu’il n’en règle dans notre cas.

Dans les chapitres à venir nous allons décortiquer chaque étape de notre pipeline et faire un choix technologique pour régler le problème si nécessaire.

L’idée finale est d’avoir un pipeline automatisable disposant de toutes les étapes nécessaires à amener un projet jusqu’à son déploiement. Afin que celui-ci soit vraiment utilisable dans le monde actuel, il faut qu’il soit portable sur le cloud.

OVERVIEW :

Diagram

Description automatically generated

La première étape est pour déclencher notre pipeline est une action qui signalera le besoin de déclencher les étapes, le déclencheur (en anglais Trigger) vient très souvent d’une action réalisée sur le dépôt contenant le code de notre projet. En parlant de dépôt, il s’agit bien sûr d’une base de données liée à un logiciel de contrôle de version.

Selon les avis, le trigger ne fait pas toujours parti du pipeline, je suis d’avis que le trigger fait parti de la solution complète est pas conséquent doit être abordé.

Le choix d’un logiciel de contrôle de version n’est pas un choix fondamentalement important pour ce que nous essayons de réaliser, c’est pourquoi nous ne nous attarderons pas trop longuement sur le choix de celui-ci.

Git représente la plus grande part du marché pour une bonne raison, historiquement le logiciel effectue le travail parfaitement. Il existe un grand nombre de concurrent qui chacun représente une petite part du marché, les avantages mis en avant par chaque concurrent est souvent lié aux autres produits qu’ils proposent et souvent dans une idée de garder l’ensemble des ressources utilisées par une entreprise au prêt d’un seul fournisseur.

Dans notre cas git, rempli l’ensemble de nos critères de sélections. Afin de comprendre ce qui va constituer le trigger, nous allons nous attarder sur quelques actions que le logiciel permet d’effectuer.

Comme nous l’avons discuté, le but d’un pipeline est de s’assurer que notre projet et donc sont code soit bien aux standards désirés avant de le rendre accessible à nos utilisateurs. Il ne fait donc sens de faire passer tous ces tests à notre projet uniquement lors que nous amenons des modifications à celui-ci.

Comme Git fonctionne-il : le principe est relativement simple. Vous disposez de votre dossier de travail localement mais vous souhaitez pouvoir travailler sur votre projet avec d’autres personnes. Vous allez donc mettre votre code à leur disposition via ce git appel un repository. Cela n’est rien d’autre qu’un dossier héberger soit sur l’un de vos serveurs soit par la solution cloud proposée par git, Github. Une fois votre dossier de projet disponible pour vos partenaires de travail, l’intérêt est que chacun puisse contribuer. Vu que chacun travaille sur le même code, git proposes quelques actions et outils permettant à tout le monde de travailler ensemble facilement.

Chacun va travailler sur sa version du code mais ne vas pas la partager avec les autres avant d’être sûr d’avoir correctement effectuer son travail. Il travaillera sur ce que l’on appelle une branche. Il s’agit du même principe que pour votre projet principal mais qui sera garder à part. La branche contenant votre projet principal mis en production s’appellera par défaut master ou main suivant les envies de Microsoft. Vous pouvez également disposez d’une branche à jour mais uniquement pour des tests. Une fois qu’un des contributeurs a fini sa tâche, il va vouloir rapatrier son travail sur le projet de test par exemple. Il va dans ce cas faire une action portant le nom de merge. Il va rapatrier son code sur celui de la branche souhaiter et ainsi fusionner les deux.

C’est ce qui suit qui va être important pour nous. Pour l’instant notre contributeur travail localement et veut maintenant partager son code avec le reste de l’équipe et le mettre en production ou le tester avec le travail des autres. Il va donc devoir envoyer son travail sur le répertoire distant partagé. C’est cette action qui sera notre trigger, en effet, avant de déployer notre projet avec ces nouveautés, nous allons nous assurer que tout ce qui fonctionnait le fait toujours. Il va donc passer à travers notre pipeline. Cette action appelée par Git Push qui envoie le code sur le dépôt distant va signaler que notre processus doit être lancer.

Il est important de préciser que git ne fait pas que les quelques actions mentionner mais bien plus et que les actions ont été simplifier dans les explications, git n’étant pas d’une grande importance dans notre projet, nous nous limiterons aux concepts importants pour comprendre le fonctionnement de bout en bout pour la mise en production de notre projet.

//Microservices

Une fois notre trigger déclencher, notre projet va être récupérer sur son répertoire. Avant de discuter qu’est ce qui va le récupérer et ce qu’il va en faire, nous allons devoir nous attarder sur ce que notre projet doit contenir pour pouvoir être mis en production. Comme discuté nous pourrions juste le packager et remplacer le projet existant sur notre serveur de production. Mais nous avons bien compris que ce n’est pas la meilleure façon de fonctionner et c’est dans cette optique que nous avions aborder la conteneurisation.

DOCKER :

Autant Git n’avait que peu d’importance pour notre projet autant docker lui est d’une importance capitale. Comme nous l’avons discuté en parlant de conteneurisation, il va nous permettre de s’assurer que notre environnement de développement sera le même que notre environnement de production ou du moins que notre environnement dans lequel nous testons le projet est le même que celui de production. Cela est crucial pour assurer que l’ensemble du projet soit fonctionnel. Pour cela les tests que nous ferons subir à notre projet dans notre pipeline seront déterminant.

Afin de faire passer ces tests à notre projet et de le déployer, nous aurons besoin d’un outil. Nous irons plus loin dans les détails de cet outil mais ce qui est important de comprendre à cette étape est que notre outil réalisant les tests devra utiliser un environnement dans lequel les effectués.

Encore une fois, nous nous trouvons dans une situation ou le moindre problème de configuration entre notre environnement de test est celui de développement pose rapidement des problèmes. La conteneurisation règle ce souci et va nous permettre de construire notre environnement et d’y faire tourner nos tests. Si nos développeurs travail dans un environnement également mis en place via un outil de conteneurisation, le moindre changement effectuer se fera via le fichier de configuration de l’environnement conteneuriser. Le fichier pourra ainsi être passé avec le reste du projet pour être utilisé lors des étapes du pipeline.

Il existe quelques outils de conteneurisation mais le plus connu est Docker. Ici le choix est simple il existe certes d’autres outils mais le plus complet et surtout le plus communément utilisé est docker. Le second argument est très important car tout nouvel outil ayant à travailler avec de la containerisation va chercher à être compatible avec le leader du marché. Docker domine le marché et a défini les standards de l’industrie.

Comment docker fonctionne-t-il : Le but étant de fournir un fichier de configuration qui contiendra l’ensemble des actions nécessaire à la création est la configuration d’un environnement souhaité. Le but n’étant pas de configurer l’environnement direct mais bien de construire par-dessus, comme pour une machine virtuelle, il faut un élément permettant de communiquer avec le système opérateur de l’hôte. Dans le cas de docker, il s’agit de Docker Engine. Docker engine à l’instar d’un hyperviseur pour une VM va permettre d’utiliser les ressources de l’hôtes et d’utiliser ses services pour par exemple communiquer avec l’extérieur ou rendre accessible les services installés dans le container. Pour que cela fonctionne dans l’environnement, il est important de préciser que Docker doit être installer sur son hôte ce qui lorsque l’on souhaite automatiser nos actions demandera l’utilisation d’un outil de configuration de ressources sur lequel nous revendrions plus tard.

Lorsque notre fichier de configuration sera prêt, celui-ci sera construit(build) est comme pour une VM donnera naissance à une image. Une image n’est rien d’autre que la représentation d’un environnement sous forme d’un fichier. Ce fichier pouvant ensuite être monter pour rendre l’environnement utilisable.

VM VS DOCKER

Le but de docker n’est pas seulement de pouvoir construire ces environnements mais également d’avoir la possibilité de se basé sur des images existantes et d’y rajouter sa surcouche. Un peu comme un jeu de poupées russes, il est possible de rajouter plusieurs surcouches les unes sur les autres. Cela permet à une communauté de mettre à disposition des images contenant des configurations simples sur lesquels il sera possible de construire des déclinaisons en fonctions des besoins.

DOCKERFILE

Graphical user interface, text, application

Description automatically generatedComment va se présenter notre fichier de configuration : Celui-ci s’appelle Dockerfile voici un exemple simple donné par Docker.

Figure https://docs.docker.com/get-started/02\_our\_app/

FROM va nous permettre d’aller récupérer une image existant sur laquelle nous souhaitons commencer à construire. Cette image se trouve dans un dépôt similaire au principe proposé par git. Le service en ligne contenant les images mises à dispositions par la communauté s’appelle Docker Hub. Si nous venions à construire notre ficher Docker téléchargerait l’image choisie et la monterait. Cela donnera une base sur laquelle la suite de la configuration sera effectuée.

WORKDIR va simplement spécifier l’emplacement dans lequel nous allons travailler. C’est donc dans ce dossier que les commandes qui suivent seront exécutées.

COPY va nous permettre de dupliquer le dossier du projet dans l’emplacement de notre environnement prévu pour les faire fonctionner.

RUN va permettre de lancer des commandes Shell par default ou autre si préciser et ainsi fini la configuration de l’environnement. Il est possible de lancer autant de commande que souhaité.

CMD lui est unique à chaque fichier de configuration. Il est le point d’entrer du service que l’on souhaite faire fonctionner dans le conteneur, Il permettra de lancer l’service pour que celle-ci soit fonctionnelle une fois le conteneur prêt.

EXPOSE Indiquera quel port ouvrir sur notre conteneur permettant ainsi l’accès au service. Comme nous l’avons discuté, le conteneur est de base une boite hermétiquement fermée, si nous souhaitant que notre service soit disponible pour être utilisé, il est essentiel d’ouvrir une porte d’accès.

Il est important de préciser que Docker va créer une image temporaire à chaque étape appliquant ainsi les avantages de la conteneurisation à son propre procédé. L’ensemble des étapes de travail sur l’image s’effectue bien sûr dans un container créer pour l’occasion.

Il existe d’autres mot clé d’instruction dans docker, certain sur lesquels nous reviendrons dans la partie projet.

DOCKER COMPOSE :

Les micro-services domine aujourd’hui le monde de l’architecture logicielle. Une application ne sera que très rarement composée d’un seul service. Nous souhaitons que ces services soit découplé mais suivant les besoins, ils ont souvent besoins d’être disponibles les uns pour les autres. L’idée est donc de combinée plusieurs DockerFile en un seul fichier afin de n’avoir à monter que celui-ci et que les services soient configurés pour fonctionner ensemble.

Un fichier dockercompose va permettre de faire tout cela. Une fois monter, il construira un conteneur qui contiendra les conteneurs de chaque service présent dans le fichier.

Graphical user interface, text

Description automatically generated with medium confidenceSa structure est du type :

Figure https://www.freecodecamp.org/news/what-is-docker-compose-how-to-use-it/

La version est annoncée en début de fichier est sert uniquement à garder un historique. C’est la partie suivante qui nous intéresse, celle nommée « services ». Nous nous retrouvons ici avec une application complète découpée en services, chacun découplé l’un de l’autre en termes d’environnement mais étant reliés et fonctionnant dans le même conteneur.

Ici, la syntaxe est les éléments sont un peu différent d’un dockerfile. Nous avons notre liste de services, chacun disposant d’un nom et lié à une image. Tout d’abord, le nom n’est pas présent que pour une question de clarté, suivant le contexte, il fera office de lien d’accès au service. Par exemple, pour la connexion à la base de données depuis l’un des autres conteneurs. Ensuite l’image, pour que notre docker compose puisse fonctionner, il doit pouvoir accéder aux images demandées, s’il s’agit d’image génériques de services par exemple Node.js pour le premier service, il ne s’agira que de node (docker hub à simplifier le nom des images les plus communément utilisées).

Dans le cas où le projet demanderait l’utilisation d’une image créer sur mesure, il est important que celle-ci soit hébergée sur un dépôt afin que les différents servers puissent venir la récupérer afin créer les environnements lors des différentes étapes de notre pipeline. Une image hébergée sur Docker hub aura un nom suivant la structure : votre\_profil/nom\_de\_votre\_image. Enfin un dernier élément important pour récupérer la bonne image, le tag. Il s’agit du texte suivant le nom de l’image et précédé de « : ». Par défaut, celui-ci sera « latest » il s’agira de l’image la plus récente mise à disposition sur le dépôt de ce nom. Mais si nous souhaitons faire évoluer notre environnement, la dernière pourra être créer pour des tests par un développeur et ne devra donc pas être utilisé à des fins de production, on utilisera un tag spécifique évitant ainsi tout problème.

Un des problèmes de la conteneurisation est la persistance des données. Comme nous l’avons discuté. Les environnements sont construits puis détruit au grès du besoin, mais que faire si nous avons des données qui sont persistante est que nous devons conserver même une fois l’environnement détruit. Lorsqu’il s’agit de données centralisées qui n’ont rien à faire dans des containers, le service est gardé à part sur des instances fixes et dupliquées pour assurer la stabilité du bien précieux qu’est la donnée. Dans une plus petite mesure, docker a créé un service annexe aux conteneurs, permettant de sauvegarder les données. Ce service porte le nom de volume. Il n’est pas dans le conteneur ce qui lui permet de continuer à exister même lorsque le conteneur est détruit, ce qui permets aux données d’être toujours disponibles sur la machine même si un nouvel environnement venant à remplacer l’ancien du moment que la référence sur le volume est la même.

Avec Docker, nous allons pouvoir créer un environnement portable disposant de tous nos services qui va accompagner notre projet tout au long de son cycle de vie et lui permettre d’être tester, de fonctionner ou encore d’être mis à l’échelle en fonction du besoin sans presque plus jamais avoir à se soucier qu’il dispose de toutes les dépendances et configurations dont il aurait besoin.

LIENS DEVOPS.

UNE PARANTHESE SECURITE

Dans le monde actuel, la sécurité d’une application prend une place élevée dans les priorités émises lors de la conception. Il est extrêmement important de s’assurer que lors que des modifications sont apportées à l’application celles-ci ne compromettent pas le travail effectué pour la sécurisée.

Dans le monde de l’informatique, les vulnérabilités sont appelées CVE pour « Common Vulnerability and Exposure » les faiblesses pouvant générer une vulnérabilité sont-elles nommées : CWE pour « Common Weakness Enumeration ».

Les CVE sont basé sur une organisation du même nom ayant mis en place un procédé de report et de suivi des vulnérabilités, celle-ci couvrent l’ensemble du monde informatique, des systèmes aux applications. Les CVE sont annoncé uniquement par des partenaires et entreprises reconnu afin d’assurer la véracité de l’information, le système est devenu la base du partage d’information lié à la sécurité. De nombreuses entreprises proposés même des récompenses pour l’annonce de vulnérabilité sur les produits. Permettant ainsi de profiter de la grande communauté présente autour du projet et assurer la sécurité de ses produits à moindre coût car même si certain « bug bounty » comme ils sont appelés peuvent atteindre plusieurs centaines de milliers de dollars, les sommes sont souvent beaucoup plus basses et ne représentent des sommes risibles pour de grandes entreprises. A titre d’exemple, Google aura payer un peu plus de 12 millions de dollars pour un équivalent de 2900 vulnérabilités reportées sur ses produits\*.

Diagram

Description automatically generated\*https://www.bleepingcomputer.com/news/security/google-paid-12-million-in-bug-bounties-to-security-researchers/

Figure httpswww.cve.orgAboutProcess#CVERecordLifecycle

L’idée est de centraliser le suivi et d’accorder à chaque vulnérabilité une note indiquant le degré de risque qu’encourt une application et son système si rien n’est entrepris pour remédier à la vulnérabilité. Toutes vulnérabilité est loin d’être catastrophique et ne demande pas forcément d’y remédier instantanément.

Afin d’assurer un suivi pas trop compliqué, une CVE doit respecter quelques critères.

* En plus d’être annoncé par une source fiable, elle doit être reconnu par l’autorité représentant le produit sur lequel elle a été découverte.
* Doit être solvable indépendamment d’autre bug, par exemple si une vulnérabilité est liée aux fonctionnements de plusieurs applications ou dépendances fonctionnant ensemble, une CVE sera créer pour chaque « codebase » autrement dit une collection de code source utilisée pour faire fonctionner un programme.

Une fois la solution trouver pour résoudre la vulnérabilité, celle-ci est ajouter aux informations de la CVE. C’est sur cette base de données que l’ensemble de l’industrie fonctionne permettant aux logiciels dédiés à la rechercher de vulnérabilité sur les systèmes et applications, de savoir quoi chercher et de proposer une résolution à ses utilisateurs.

CVSS

OWASP TOP 10

Comme nous l’avons discuté, les vulnérabilités sont répertoriées quand il s’agit de problème lié à un produit utilisé. La très grande majorité des applications aujourd’hui sont basées ou utilises des produits existants les CVE permettent donc d’assurer que les outils sur lequel se repose notre application ne souffre pas de vulnérabilités connues. Mais lors de la réalisation d’une application, ce qui en fait un produit est le travail réalisé par les équipes de développement. Ces équipes vont produire du code. Les standards et meilleurs pratiques évoluant constamment, notamment pour résoudre des soucis de sécurité, il est très fréquent que le code soit rempli de problèmes pouvant créant ainsi des vulnérabilités.

Le monde se dirigeant à pas de géant vers une digitalisation omniprésente, la sécurité des outils utilisés dans ce but est cruciale. Une des fondations les plus connue active dans ce milieu s’appelle The Open Worldwide Application Security Project (OWASP). Il s’agit de la plus grande communauté travaillant sur les aspects liés à la sécurité. Ils proposent des outils permettant la mitigation des risques et vulnérabilités ainsi que du matériel et des ressources permettant de former, de garder le monde de la sécurité à jour.

Chart

Description automatically generatedOWASP édite toutes les quelques années une liste contenant les 10 risques les plus répandus permettant une attaque sur une application. Cette liste est la référence en termes de points devant être vérifiés par une équipe avant de rendre son application disponible à l’utilisation. Vu qu’il s’agit des plus connus, il est garanti que ce seront les premières portes d’entrée testées par tout entité souhaitant du mal à votre application. Comme vous l’aurez compris, nous parlons ici des CWE mentionné plus tôt.

Figure https://owasp.org/www-project-top-ten/

Comme nous pouvons le voir les risques les plus présents ont quelques peu évolués ces dernières années. Nous allons nous attarder sur la liste actuelle, nous reviendrons à chaque étape sur les outils permettant de déceler ces risques et autres vulnérabilités.

Broken Acces Control : Il s’agit de cas ou l’application ne défini par bien ce qui peut être accessible à chaque type d’utilisateur. Très souvent cela est dû à des règles pas assez réestives que ce soit sur l’appel ou l’accessibilité de certaines ressources. Typiquement lors de l’utilisation d’API FrontEnd-Back end la possibilité de récupérer des informations ou pire de les modifier. Ces erreurs sont liées en grande partie au code et devront donc être adressé lors de tests sur le code.

Cryptographic Failures : Ici un grand nombre de scénario sont possible mais le problème vient de l’accessibilité à l’information lors de son transit entre services en utilisant des protocoles non sécurisés ou dépassés. Il peut également s’agir d’informations sensibles n’étant pas encrypté ou utilisé tel quel et donc lisible. Ici une partie peut être vérifier sur le code, une autre sur le système, le reste dépendra des bonnes pratiques.

Injection : Il s’agit d’un des risques les plus connu qui semble heureusement être moins présent depuis la dernière mise à jour de la liste. Ici tout champs permettant à l’utilisateur d’y entrée ses informations ou services pouvant recevoir de l’information qui sera transmises pour effectuer un appel à un autre service. Toutes informations fournies par l’utilisateur et utilisé par l’application doit être vérifiée proprement. Il est sinon possible d’accéder à ou d’altérer des données. Ici c’est le code qui devra être testé.

Insecure Design : Cette catégorie est beaucoup plus large et beaucoup plus difficile à tester, certaines choses peuvent tout de même être relevé comme par exemple, des informations de connexion, pour accéder à un autre service étant mal protégés ou encore des informations sensibles reporté dans la génération de message d’erreur.

Security Misconfiguration : Comme mentionnée, les applications développées aujourd’hui s’appuient sur de nombreux outils. L’avantage amené par l’utilisation d’outils peut créer certains risques liés à la configuration de ces outils, par exemple l’ouverture de ports par défaut sur l’outils, ces ports n’étant pas utilisé, ils peuvent être utilisés comme une porte d’entrée. Cette partie peut être test post déploiement sur l’infrastructure.

Vulnerable and Outdated components : Cette catégorie s’explique d’elle-même, l’utilisation de services n’étant pas à jour laisse des bugs et vulnérabilités ayant été résolu par l’éditeur du service. Ici un scan sur le code et post déploiement sur le système permettra de relevé convenablement les problèmes potentiels.

Identification and Authentication failures : Cette partie couvre tous les problèmes liés à une identification trop faible, l’utilisation de valeur par défaut, l’utilisation de valeur de session après une durée trop longue. Certains de ces soucis peuvent être relevé mais pour d’autres il s’agit de bonnes pratiques et standards.

Software and Data Integrity Failures: Il s’agit d’une catégorie important pour notre pipeline car lorsque nous faisons appel à des dépendances pour notre application, des téléchargement s’effectue sans vérifier la source et la validé des données téléchargées. Ici des logiciels dédiés pour la vérification de dépendances sont disponibles mais la mitigation demande de mettre en place des services n’étant pas nécessaire pour des petits projets. Dans notre cas, nous contenterons de nous basé sur des sources reconnues et des images de bases officielles.

Security Logging and Monitoring Failures : La difficulté de tester les problèmes liés à ce groupe de risques est élevée. Même si le Continuous Monitoring sur lequel nous reviendrons offre quelques possibilités.

Server-Side Request Forgery(SSRF): Il s’agit d’un type de risque très spécifique à l’utilisation d’url données par l’utilisateur pour la récupérations d’informations, pour par exemple récupérer des données afin de pouvoir les utilisé dans le logiciel. Ici également difficile de tester mais certains points peuvent être relevé sur le code.

Grâce à cette liste, nous allons pouvoir nous efforcer d’ajouter tous les outils nécessaires permettant de s’assurer que notre application soit sécure une fois validé par notre pipeline. Du moins les développeurs disposeront de toutes les informations nécessaires leur permettant de sécuriser l’application. Il peut être souhaitable que le pipeline bloque le déploiement en fonction des vulnérabilités détectées.

LES ETAPES DECORTIQUEES :

Nous allons nous arrêter ici sur les étapes de notre pipeline et les outils qui seront nécessaire à son bon fonctionnement. Comme nous l’avons discuté, si une des étapes venant à ne pas se déroulée de manière satisfaisante le processus n’ira pas plus loin et le développeur ayant envoyé son nouveau code vers le dépôt distant sera averti du souci et pourra travailler sur la résolution du problème sans que ses changement n’impact les différents environnements de tests ou de production.

BUILD :

La première étape peut sembler triviale mais n’en n’ait pas moins essentiel particulièrement suivant les langages utilisés pour construire le projet.

Nous avons ici 2 parties, vu que nous avons fait le choix de conteneuriser notre environnement, il est important de voir si celui-ci va pouvoir se construire correctement. Si ce n’est pas le cas, inutile d’essayer d’aller plus loin, rien ne marchera correctement.

La seconde partie de cette étape, dépend des langages utilisés. En effet, certain langage afin de pouvoir être utilisé par une machine doivent être compliés et ainsi transformer en fichier exécutables. Cette étape est cruciale pour tous les langages demandant à être compliés avant de pouvoir être exécutés, il s’agit du test le plus simple permettant de savoir si le code contient des erreurs pouvant empêcher son exécution.

LES TESTS :

Une fois que notre environnement et si besoin notre programme monté et prêt à fonctionner, il s’agit de s’assurer que tout fonctionne correctement et que les standards du produit proposé soit bon.

Première étape, les tests unitaires, ils sont la base de la vérification du bon fonctionnement d’un programme. Un test va s’occuper de valider le bon fonctionnement d’un morceau de code. D’une manière assez pragmatique, s’il l’on veut tester une fonction, il nous faudra écrire plusieurs tests s’assurant que la fonction remplie bien ce qu’on attend d’elle suivant les informations qu’on lui fournit où qui existent déjà dans le projet. Et à l’inverse qu’elle n’altère pas certaines données. Le but est simple, s’assurer que en cas de modification de la fonction ou d’un autre morceau de code auquel celle-ci ferait appel, les résultats attendus restent les mêmes.

Viennent ensuite les tests d’intégrations, en effet, il est rare que notre application fonctionne en autarcie sans autres services, qu’il s’agisse de l’utilisation d’une base de données, de l’appel à une API externe ou encore l’appel d’une fonction appartement à un autre service, notre application ne fonctionnera pas correctement si ces besoins ne sont pas également fonctionnels. Il est donc nécessaire de réaliser des tests utilisant ces services soit directement mais il est plus sage d’effectuer ces tests directement sur les parties du code de l’application y faisant appel.

Ces deux premiers jeux de tests sont souvent réalisés par les développeurs, ils sont adjacents au code et ne requière pas d’utilisation d’outils externes particulier. Chaque langage dispose de son propre outil de test, celui-ci est souvent en extension et installer dans l’environnement. Ils permettent de mettre en place une structure pouvant être run de la même manière que du code et d’obtenir un retour indiquant quels tests sont réussi ou non.

Un dernier test effectué directement sur le code, appeler la revue du code. Un logiciel externe va se charger de scanner le code à la recherche de code déprécier ou pouvant générer des problèmes de sécurité. Ici se trouve un grand nombre d’outils disponible permettant de scanner le code et de donner un retour. Ce type de test est particulièrement utile sur des systèmes complexes, il est difficile de tester l’entièreté du code sans compter qu’il est également difficile d’anticipé tous les scénarii possibles. Les outils vont s’appuyer sur les tests réalisés précédemment, ils ne sont en aucun une manière de remplacer les tests unitaires et d’intégrations. Il s’agit ici de tests dit statiques car le code n’est en aucun cas exécuté.

Les outils vont généralement tester et proposé des améliorations concernant :

La refactorisation du code qu’il s’agisse de code dupliqué voir même partiellement dupliqué, en proposant une ou plusieurs fonctions couvrant l’entier du besoin réduisant ainsi la complexité du projet. Sur le même principe, lorsqu’une partie du code peut être jugée trop compliqué et donc difficile à maintenir, l’outil peut proposer une alternative.

En cas de patterns utilisé pouvant générer des problèmes ou des fonctions pouvant être poussées à l’erreur en cas de passage d’informations problématique en paramètres.

Et enfin, dans un souci d’assurer la sécurité de l’application, l’outils testera le code et l’ensemble de l’application pour toutes CWE ou CVE. Qu’il s’agisse de s’assurer qu’il ne soit pas possible de réaliser d’injection SQL, d’attaques du type cross-site Scripting ou encore de Buffer Overflow. Cela permettra de garder l’application à jour contre un bon nombre de vulnérabilité connues pouvant être exploité à cause d’un mauvais code ou d’une mauvaise configuration. Il s’agit ici de la partie test de sécurité pré-déploiement, une autre partie se fait post-déploiement, l’application n’étant pas du tout dans le même état, tout ne peut pas être testé.

COMPARATIF DES OUTILS EXISTANTS (non exhaustif) :

Le choix d’un outil d’inspection de code va dépendre de chaque projet. Tous les outils ne gèrent pas forcément le langage que vous utiliserez dans votre projet. Il est également important de prendre en compte les outils avec lesquels vous allez devoir intégrer ce nouvel outil. De la même manière il peut être intéressant d’avoir un seul outil pour le test dynamique de votre code que pour les tests statiques. Le dernier paramètre à prendre en compte correspond à votre l’hébergement de l’outils, si vous travailler dans un environnement fermé, utilisé un outil uniquement disponible en version cloud peut demander de la configuration supplémentaire et des risques de problèmes dans l’intégration. A l’inverse, il peut être judicieux de se démettre du maintient d’un serveur dédié à un outils supplémentaire.

Voici une liste de 3 outils ayant chacun leurs particularités :

Les 3 outils vont offrir :

-Une analyse du code avec une vision de la couverture des tests effectués.

-Une recherche d’optimisation de code via la détection de code dupliquer ou de fonctions mal optimisées.

-La détection de vulnérabilité ou de problème pouvant amener à des vulnérabilités.

-Plus d’une 20aines de langages supportés.

-Sont intégrable avec la majorité des cloud provider, gestionnaire de dépôts de code et outils de gestion de pipeline CI/CD.

Codacy propose de personnaliser les règles de vérification de qualité du code. En effet comme tout outils de reporting, la quantité d’information peut être très élevée. Suivant le niveau de standards émis par la société, il se peut que toute l’information retournée ne soit pas utile. En effet certains points sont cruciaux, notamment au niveau des performance et de la sécurité mais d’autre peuvent être plus secondaire et l’entreprise les juger trop chronophage. En choisissant les règles vous amenez vos équipes à se focaliser sur les types de problèmes que vous jugez les plus importants. Disponible en version cloud ou on-premise.

Sonarqube fait partie de la suite Sonar, visant à couvrir tous les besoins en analyse de code, il s’agit du membre de la liste disposant de la couverture du plus grand nombre de langage ainsi que de la compatibilité avec le plus d’outils. Il s’agit également d’un des outils les plus utilisé dans sa catégorie, il dispose d’une version gratuite déjà très qualitative et est disponible en version cloud ou on-prem.

Veracode convient autant à l’analyse statique que dynamique du code avec un accent mis sur la sécurité. Il propose une option de scan lié à l’analyse de la composition du logiciel ce qui permet de refactorisé jugé complexe en plus petits morceaux moins complexes simplifiant ainsi la maintenance du logiciel. Il est disponible uniquement en version cloud.

Certains des avantages listés sont également présents dans les autres logiciels listés mais de manière moins poussée ou en tout cas moins mis en avant par le concepteur.

CHOIX SONARQUBE

RESUME CI

Nous avons maintenant fait le tour de l’ensemble des points à mettre en place dans la partie d’intégration continue de notre pipeline. Avec ces étapes et leur outils associés, les développeurs de l’application devrait pouvoir recevoir toutes les informations nécessaires à la réalisation du code pour un logiciel respectant les attentes et standards demandé. Cette partie du pipeline ne couvre certes pas tous les tests mais peut être utilisé très fréquemment par les développeurs. Il est important de noter que suivant le langage et le projet, une grande partie des tests mentionnés peuvent être réalisés par le développeur dans son environnement de développement voir même avec l’aide de son IDE directement. Il n’empêche que lorsqu’une nouvelle version est prête, il n’en est pas moins une obligation de s’assurer que rien n’a été oublié et d’avoir une trace du passage des testes et leur retour en cas de problème.

CI/CD TOOLS

Afin de faciliter la réalisation de ces tests, d’avoir un suivi sur leur retour et d’automatisé toute la chaine du pipeline, il faut un outil se chargeant d’orchestrer la chaine d’événement et d’arrêter le processus lorsque cela est nécessaire.

Les outils de CI/CD sont le cœur du pipeline, ils vont permettre d’automatiser toutes les étapes énumérées jusqu’à présent et de déployer le projet dans ces différents environnements que sont l’environnement de test/QA, de Staging et enfin celui de développement.

Ils sont suffisamment souples pour permettre d’effectuer différent type de tests et de déployer dans différents environnements en fonction de la source souhaité. Typiquement, un pipeline peut être prévu en fonction d’une branche d’un dépôt de code pour tester en profondeur certain aspect mais ne demande pas l’entièreté du pipeline lié à la production économisant ainsi du temps et des ressources.

Ces outils sont très souvent associés au terme DevOps car ils sont la jonction entre les équipes de développement et de production. Il s’agit des outils assurant que l’attente des deux parties est réalisé et permet ainsi de s’assurer que le logiciel passé dans la chaîne ne manquera pas à ses obligations.

Pour récapituler, nous attendons de notre outil de CI/CD, qu’il soit synchronisé avec notre dépôt de code et réagisse lorsqu’une nouvelle version est déposée sur celui-ci. Une fois qu’il reçoit le signal, l’outils va récupérer notre projet sur le dépôt. Il va se charger de construire l’environnement grâce à docker, si besoin il construira également notre programme et le mettra en fonction dans l’environnement qu’il a construit. Une fois le programme prêt, il exécutera les tests prévu unitaires, fonctionnel et de qualité de code afin de s’assurer que tout est aux normes attendues. Si c’est le cas, il se chargera de déployer notre projet dans les environnements souhaités.

OUTILS

Nous allons nous intéresser à 3 outils de CI/CD parmi les plus courant sur le marché. Le monde technologique tournant autour du DevOps étant devenu très au monde pour de bonne raisons, il existe un très grand nombre d’outils. La majorité des plus grands hébergeurs disposent de leurs propres outils de CI/CD voir même de l’ensemble des outils nécessaires à la réalisation de l’ensemble des étapes du pipeline. Ces outils sont souvent très bons notamment ceux proposé par Amazon Web Services et permettent la mise en place d’un pipeline rapidement et efficacement avec peu de configuration dites système. Je pense cependant, qu’il est important de rester agnostique et de se baser sur des outils open-source et ne liant pas la configuration à un hébergeur. Le vent tourne vite dans le milieu technologique on ne sait jamais trop de quoi est fait demain et ne pas pouvoir se permettre de migrer son infrastructure facilement peut s’avérer coûteux.

Travis CI :

GitLab :

Jenkins :

Circle CI :

CD

CRITERES

LE CLOUD ET L’INFRASTRUCTURE AS CODE

TERRAFORM ET LE PROVISIONNEMENT DE RESSOURCES

CONFIGURATION MANAGEMENT AVEC ANSIBLE

KUBERNETES

SCANNER DE VULNERABILITE

OWASP ZAP

CONTINUOUS MONITORING

NAGIOS

CONCLUSION

Général Bibliographie :

<https://www.redhat.com/en/topics/security/what-is-cve>

<https://www.cve.org/ResourcesSupport/AllResources/CNARules>

<https://owasp.org>

<https://www.sonarsource.com/>

<https://www.codacy.com/>

<https://www.veracode.com/>

<https://en.wikipedia.org/wiki/Codebase>

Securing Devops- Safe Services in the Cloud

Generic Pipelines Using Docker\_ The DevOps Guide to Building Reusable, Platform Agnostic CI\_CD Frameworks(Book)

The DevOps Handbook\_ How to Create World-Class Agility, Reliability, and Security in Technology Organizations(Book)

Cloud Native DevOps with Kubernetes(Book)

The DevOps Adoption Playbook. A Guide to Adopting DevOps in a Multi-Speed IT Enterprise(Book)

The DevOps 2.0 Toolkit\_ Automating the Continuous Deployment Pipeline with Containerized Microservice (Book)

Pipeline as Code - Continuous Delivery with Jenkins, Kubernetes, and Terraform (Vidéos)