
Du **monolithe**
vers les **micro-services**
chez **Macompta.fr**

Remerciements

Je tiens à remercier Annick LASSUS, dont l'une des casquettes est Co-Responsable des feues licences pro CDTL, pour son regard éclairé lors du suivi de ce mémoire.

Cyrille SUIRE, Enseignant chercheur et Co-Responsable des feues licences pro CDTL pour son approche épicurienne et passionnée de la tech.

Sylvain Heurtier, Président et Eric Pham, CTO chez macompta.fr pour me permettre de concrétiser cette reconversion. Ainsi que l'ensemble de leur équipe pour leur accueil immédiat.

Sommaire

0 - Résumé pour les citrons

(1) Macompta.fr propose un logiciel de comptabilité SAAS. C'est une entreprise d'une quinzaine d'années en évolution permanente autant sur le plan du produit que de l'équipe. Cette croissance est primordiale pour sa survie.

(2) D'un point de vue technique, l'application a été développée sous forme de monolithe pendant 10 ans. À la fin de cette période, cette architecture montrait ses limites autant pour le maintien du code que pour l'organisation des équipes. Le résultat étant un allongement des délais de commercialisation.

L'entreprise a alors entrepris un découpage de son application en se basant d'abord sur des réalités métiers (facturation, compta, paie). Ces transformations l'ont conduite progressivement sur la voie d'une architecture Micro-services, du DevOps et de la Conteneurisation.

(3) Cette transformation change le mode d'hébergement de macompta.fr qui passe de solutions PAAS à CAAS. Cette transition nécessite l'intégration dans l'entreprise de nouvelles connaissances (conteneurisation et orchestration). C'est dans cette dynamique que j'ai été recruté, pour intervenir au fil d'une montée en compétences, sur les aspects DevOps de l'entreprise. Cette acquisition de compétences se réalise par l'expérimentation et avec l'accompagnement de prestataires extérieurs pour de l'audit et de la formation.

Aujourd'hui, macompta.fr a migré ses environnements de Test dans une architecture micro-services conteneurisées mais sous une forme encore immature pour la Production. Je dois faire évoluer cette implémentation, avec l'accompagnement de prestataires, pour permettre une mise en production progressive.

(4) Suite à un premier audit, des préconisations ont été déterminées pour faire évoluer la conteneurisation, l'orchestration et la chaîne de déploiement, chez ma macompta.fr. Une première itération d'évolution a été réalisée par ce prestataire et je doit la finaliser et l'étendre à l'ensemble du produit macompta.fr.

Jusqu'à présent, mon impact dans cette transition a été principalement préparatoire.

J'ai acquis une certaine connaissance de l'existant, à travers diverses actions de maintenances ou d'extensions : de l'alimentation électrique des serveurs internes, à la configuration 'indirecte' du serveur de production, en passant par l'édition de pipeline, la gestion de VMs d'outils internes et bien sûr le maintient des clusters locaux.

Un premier constat serait que les problématiques sont nombreuses et diverses. L’enjeu de ce mémoire est de parvenir à les formaliser et de valider une vision commune. Ainsi, ce document veut non seulement répondre à un objectif scolaire, mais aussi à un besoin de l'entreprise ou du moins à un besoin de mon poste dans l'entreprise.

1 - Un contexte de croissance

1.1 - un peu d'histoire

Macompta.fr est une application de comptabilité en ligne développée depuis 2008.

Initialement, l'application était uniquement développée par Eric Pham (Actuel CTO), employé par Sylvain Heurtier dans son cabinet d'expertise comptable, à Paris. Le projet prenant de l'ampleur, le cabinet a été vendu et la société Macompta fondée.

Depuis l'entreprise et l'application n'ont cessé de croître. L'application, au début très basique, s'est vue ajouter régulièrement de nouvelles fonctionnalités. Pour réussir à maintenir, développer, tester, promouvoir cette application et assister ses utilisateurs, de nouvelles personnes ont été régulièrement embauchées.

Aujourd'hui, l'entreprise est basée à Lagord et récemment à Lyon. Elle est composée d'une trentaine de personnes, dont un peu plus de la moitié de développeurs.

Les fonctionnalités principales sont la gestion de la comptabilité, la facturation, la liasse fiscale, des notes de frais, des immobilisations et de la paie.

1.2 - une application SAAS

L'application Macompta.fr est un produit SAAS (Software As ÀService) : en contrepartie d'un abonnement, le client accède au service via un navigateur web ; il n'a pas à se préoccuper du maintien de l'application. Tous les utilisateurs utilisent la même "installation" de l'application et donc la même version.

L'application est codée en PHP puis hébergée sur des serveur tiers, gérés et maintenus par des prestataires.

L'application demande un travail constant de développement. D'une part pour la maintenir et l'adapter au fil des évolutions comptables, fiscales et technologiques. D'autre part pour répondre à toujours plus de cas d'utilisations, par l'ajout de fonctionnalités.

Cette capacité à évoluer est critique, car elle permet de conserver et capter de nouveaux clients.

1.3 - Des problématiques de croissance

Ce développement constant apporte ses propres problématiques. Un exemple trivial lié à l'augmentation de l'équipe est la nécessité régulière d'augmenter la taille des locaux.

D'autres problématiques plus techniques vont avoir des implications à la fois sur le code, sur l'organisation des équipes et sur l'hébergement des produits.

Résoudre ces problématiques est essentiel pour ne pas freiner le développement de l'entreprise.

1.4 - un découp(l)age

Une de ces problématiques de croissance est la suivante. Le produit et l'entreprise Macompta.fr ont d'abord été envisagés comme un tout, c'est une vision monolithique.

Au fil du temps et son développement, ce bloc est devenu trop complexes et un travail de découpage a été initié.

Ce découpage traverse plusieurs couches techniques, du métier à hébergement, en passant par l'équipe est le flux de travail.

C'est un processus qui s'inscrit dans un temps long, car non seulement car il nécessite un travail important et mais également car il faut le piloter à chaque étapes.

Aujourd'hui, cette transformation touche une partie liée à l'hébergement et la façon de mettre le code en production : le déploiement. C'est cette partie qui me concerne et qui sera développée.

2 - vers les micro-services hybrides

Avant de détailler ses implications dans le déploiement, il faut d'abord bien comprendre l'historique et la genèse de ce découpage. Découper quoi ? pour quelles raisons ?

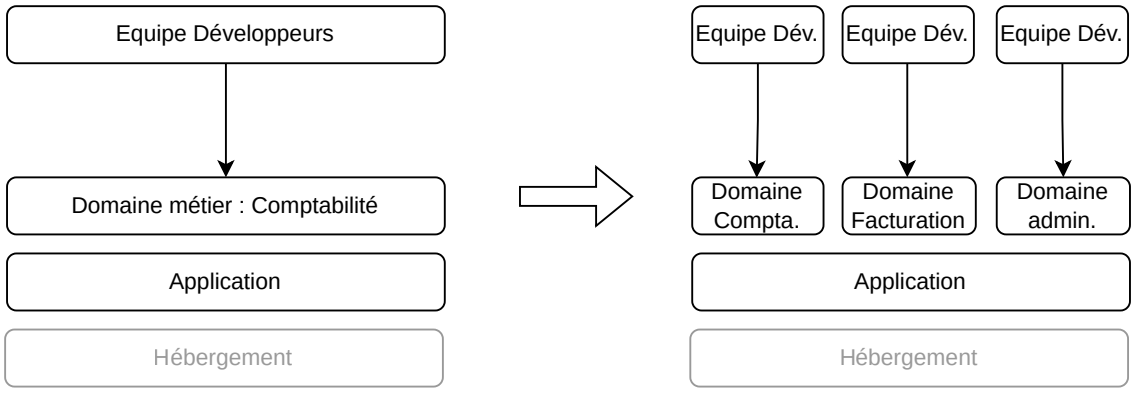
2.1 - plusieurs domaines métiers, un monolithe

Pendant plusieurs années, macompta.fr proposait un seul produit métier – la comptabilité – avec une seule application portée par une seule équipe.

Rapidement, macompta.fr a proposé d'autres domaines métier comme *la facturation*, puis *les immobilisations*, ou même d'administration du site. À ces domaines se sont spécialisés des développeurs. Toutefois, tous étaient adressés par la même application, la même base de code commune.

Lorsqu'un seul bloc répond à toutes les problématiques, on parle de Monolithe.

> **Représentation en couche**
>
> La représentation ci-dessous représente le produit macompta.fr découpé en couches d'abstractions. De la couche la plus abstraite, au-dessus, représentant le domaine métier à la plus concrète, en bas représentant le matériel physique de l'hébergement.
>
> L'hébergement est grisé, car géré par un prestataire extérieur.
>
> Le schéma ci-dessous montre l'évolution du produit macompta.fr, avec l'ajout de nouveaux domaines métiers et la spécialisation des développeurs. On voit que tous les domaines reposent sur la même application.



limites du monolithe

L'application unique avait un couplage fort. C'est-à-dire que n'importe quelle partie du code pouvait utiliser ou être utilisée par une autre partie.

Dans un premier temps, ceci a simplifié les liens d'un domaine à l'autre et a été source de simplicité de rapidité.

Ce couplage a montré ses limites avec le grossissement de l'application.

Chaque modification du code devenait plus complexe, car elle pouvait avoir des implications toujours plus nombreuses sur d'autres parties du code.

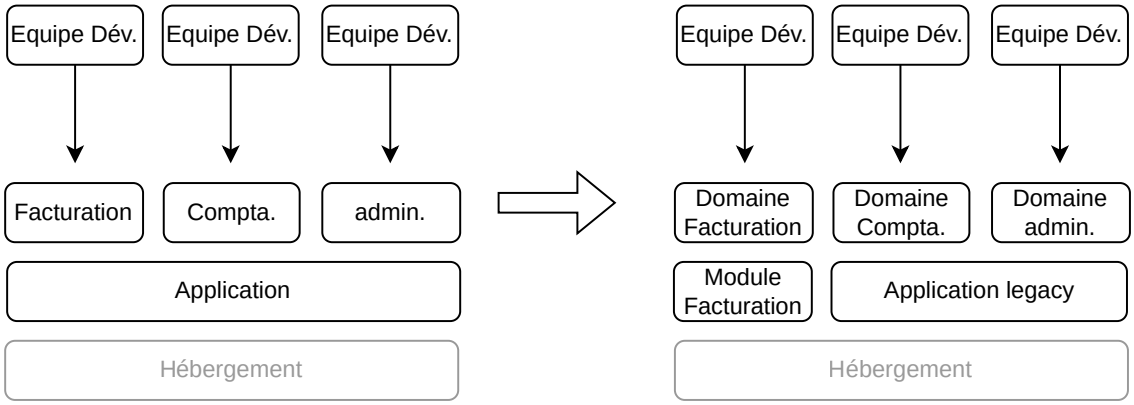
De plus, la création d'équipes spécialisées par domaine métier, engendre des chantiers simultanés. Ceci ajoute une couche de complexité car chaque équipe intervient en même temps sur la même base de code.

Ces inconvénients, liés à l'architecture monolithique, ralentissaient la commercialisation de nouvelles fonctionnalités, augmentaient le risque de panne en production et compliquaient l'intégration de nouveaux développeurs.

2.2 - un découplage applicatif

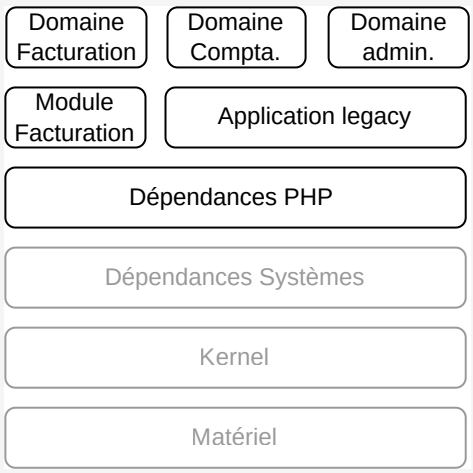
Pour pallier ces problèmes, de nouvelles architectures ont été envisagées : MVC, structuration autour de Frameworks. Puis d'autres modèles plus centrés sur le métier comme le DDD, l'architecture hexagonale et l'*event sourcing*...

Ces différentes recherches ont permis de structurer et d'organiser les nouvelles fonctionnalités incluses au monolithe. Toutefois, l'existant étant parfois trop important pour être converti, ces nouveautés ont continué de cohabiter avec, pour créer un patchwork de technologies, un monolithe multiforme.



limites du découplage applicatif

Avec ces premières recherches, sont apparus de nouveaux couplages et limites liés aux dépendances.

> Chez macompta.fr l'essentiel du code est écrit en PHP. Ce code utilise du code de librairies PHP externes (appelé *vendors* en php). Ce PHP est interprété par un interpréteur sur lequel il est possible d'activer des *extentions*. Cet interpréteur PHP est intégré au serveur HTTPD d'Apache sur lequel il est également possible d'activer des modules.
>
> Enfin, toutes ces dépendances peuvent avoir leurs propres dépendances, parfois même dans d'autres langage.
>
> On remarque que l'on a donc plusieurs niveaux de dépendances. En reprenant notre schéma précédant, on peut détailler les couches de dépendances d'application et Hébergement.
>
> Les *vendors* sont les dépendances PHP. Les extensions PHP, et les modules Apache et les librairies binaires ou autres sont inclus dans les dépendances systèmes ; elles sont fournies par le fournisseur de la plateforme (l'hébergeur); macompta.fr n'a pas la main dessus.
>
> 
>

On voit que toute l'application utilisait le même ensemble de dépendances PHP.

Lorsqu'un changement de version d'une dépendance comportait des *breaking changes*, il fallait chercher et mettre à jour toutes ces utilisations, au risque d'en oublier.

Le fait de partager les mêmes dépendances système est également bloquant, en particulier concernant l'interpréteur PHP :

PHP a beaucoup évolué ces 10 dernières années (2 versions majeures depuis 2015). Changer la version du langage a nécessité de mettre à jour l'ensemble du code et de ses dépendances externes. C'est donc un chantier important, global, en partie bloquant qui a nécessité un effort notable. Et qui est à renouveler aujourd'hui avec le passage de la version 7 à 8 de PHP.

2.2 - un découplage des dépendances

2.2.1 - la paie une application à part

Vers 2020, l'application Paie est lancée. D'un point de vue métier, cette application est totalement détachée de l'application macompta.fr. Il est donc décidé de la développer à part et de s'affranchir ainsi des dépendances de sa grande sœur.

Ceci lui permettra, par exemple, de rapidement passer à la version 8 de PHP.

2.2.2 - les micro-services, la promesse

Au même moment, la notion de micro-services est introduite chez macompta.fr.

La promesse de l'architecture micro-service est de parvenir à un découplage total de chaque fonctionnalité du code. Pour cela, chacune est isolée dans un service, qui sera tel "un serveur" à part entière, exposant sa propre API.

Chaque service disposant de la même liberté technologique que la *paie*. Leur code est plus simple, les risques de régression plus réduit, les déploiements séparés, moins risqués. Au final, le délai de commercialisation réduit.

Pour être implémenté à moindre coût, cette architecture s'accompagne souvent d'autres éléments.

2.2.3 - Pas sans les conteneurs

Cette architecture imposerait un coût d'infrastructure trop important sans l'arrivée à maturité de la technologie des conteneurs applicatifs avec Docker. Cette technologie permet de virtualiser avec beaucoup de légèreté, plusieurs systèmes sur une même machine, afin d'isoler des applications.

Dans le cas des micro-services, la conteneurisation permet de créer un réseau de services sur une ou plusieurs machines.

2.2.4 - Pas sans le DevOps

De même, cette architecture est généralement mise en place en même temps qu'une philosophie DevOps. Cette démarche vise à permettre aux développeurs d'accompagner leurs applications en Production : en les y déployant et aussi et les y observant. Le délai de déploiement est ainsi raccourci et la sur-charge de déploiement/observation, lié à la multiplication des services, est répartie.

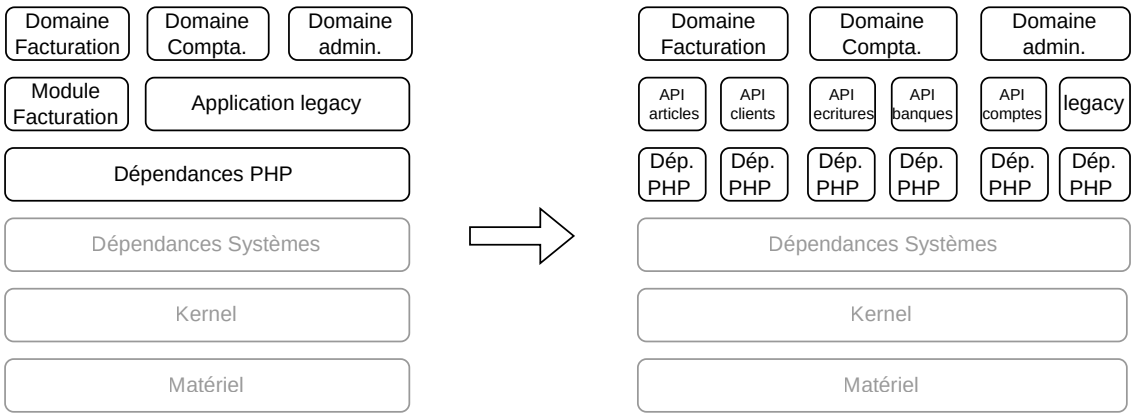
Pour arriver à cela, le DevOps, décloisonne les métiers de développeur (Dev) et ceux d'admin système/opérateur (Ops), en les faisant collaborer. Les Ops créent des automatismes et des outils pour permettre aux Dev de déployer leurs applications et d'en surveiller le bon fonctionnement "applicatif".

2.2.5 - Du micro-service hybride

À l'heure où macompta.fr s'intéresse au micro-service, elle ne maîtrise pas la conteneurisation et ne dispose pas de chaîne de déploiement automatisée.

Plutôt que d'aborder toutes ces transformations de front, l'entreprise décide de les entreprendre par étapes. Un Ops intègre l'entreprise ; cette nouvelle ressource va permettre d'implémenter, dans un premier temps, une forme de micro-services, sans conteneur.

En effet, Httpd d'Apache, le système de serveur historique de macompta.fr permet de séparer virtuellement des hôtes différents, les *VHosts*. Cette fonctionnalité, très accessible a permis le découplage des librairies PHP externes.



On voit toutefois que l'ensemble partage encore les mêmes dépendances systèmes qui comprennent l'interpréteur PHP, les extensions PHP, les modules APACHE... Ceci signifie par exemple qu'une montée en version du PHP, se ferait simultanément sur l'ensemble de l'application.

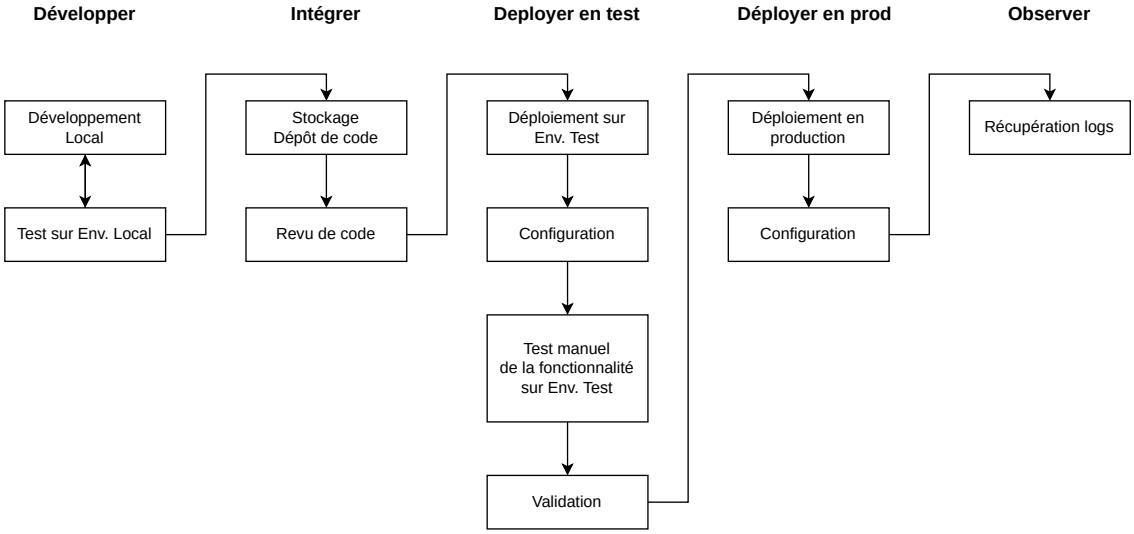
Cette multiplication des services (qui vont rapidement dépasser le nombre de développeurs), impose également une transformation de la chaîne de déploiement.

2.3 - une automatisation de la chaîne de production

2.3.1 - au temps du monolithe, l'huile de coude

Le code suivait une chaîne de production essentiellement manuelle qui n'a pas changé concernant le *legacy* (ancien monolithe).

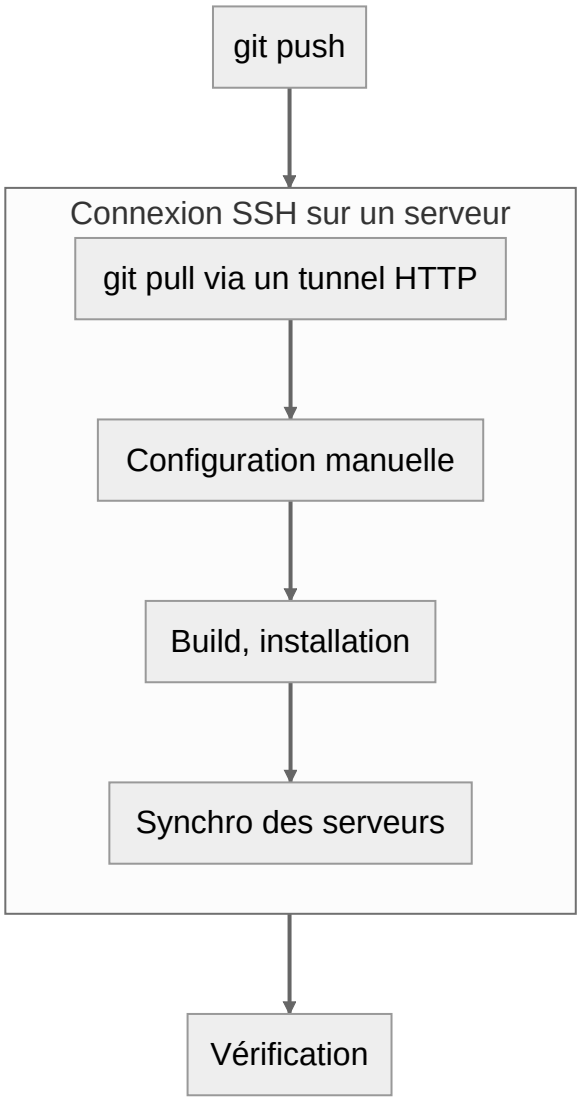
Vérifier la qualité du code, déployer, configurer, tester, lire les logs, toutes ces étapes étaient réalisées directement par les développeurs. Cette charge de travail étant facilement soutenable puisqu'il n'y avait que le monolithe.



Le flux de déploiement

Une des étapes les plus complexes et cruciales est le déploiement.

Le flux de déploiement du monolithe, devenu aujourd'hui le *legacy* :



Le code du *legacy* est stocké et versionné dans un dépôt Git accessible uniquement en intranet. Lors du déploiement, un tunnel HTTP temporaire et sécurité expose le dépôt sur Internet.

Le *déployeur*, humain, se connecte alors sur le serveur, en SSH, et *pull* le code via le tunnel HTTP.

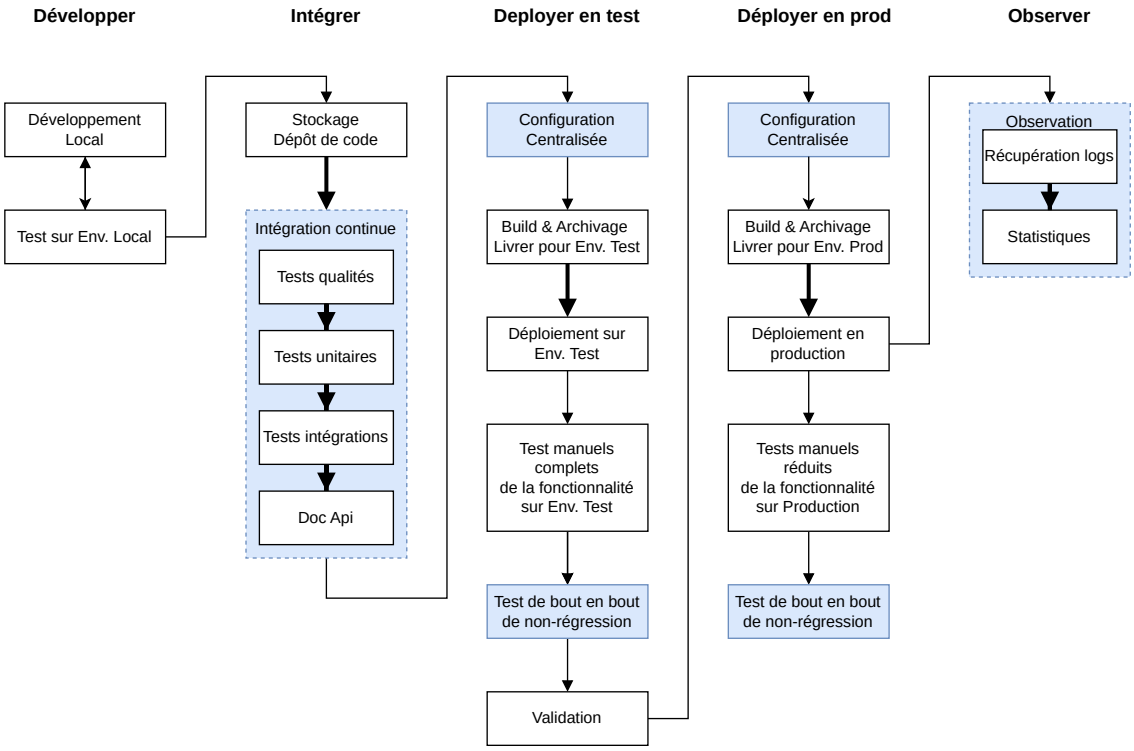
Si nécessaire, les fichiers de configuration sont édités (des fichiers de variables d'environnement, par exemple).

Si nécessaire, une phrase d'installation et/ou de build est lancée (installation de dépendances, build de script js...).

Enfin, une fois le serveur prêt, celui-ci est synchronisé avec son répliqua.

2.3.2 - Vers la multitude

Le passage en micro-services multiplie le nombre de flux de production, à tel point qu'une personne ne peut les avoir tous en tête. L'automatisation de plusieurs des tâches de ces flux, l'implémentation de méthodes DevOps, devient inévitable.



Intégration

Une intégration continue (CI) est mise en place afin de vérifier automatiquement la qualité du code (conventions d'écritures, tests unitaires) à chaque modification.

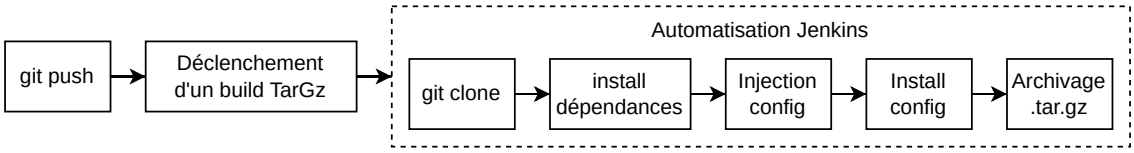
Configuration

Toutes les configurations & secrets sont centralisés et stockés dans un outil "Hesperides" – un outil Open source produit par *Sncf connect* – . Ce service permet de stocker des variables, par environnement d'exécution, qu'il utilise pour fournir des fichiers de configurations.

Déploiement

Le déploiement est désormais en deux étapes : livraison et déploiement.

La livraison ou étape de *release*, produit (*build*) une archive de l'application prête à utiliser.
Il est a noté que cette archive est déjà configurée pour un environnement d'exécution spécifique.



Le déploiement installe cette archive dans l'environnement d'exécution.

Ces deux étapes peuvent être déclenchées séparément ou séquentiellement.

Les automatisations de ces deux étapes sont exécutées de façon centralisée par l'outil Jenkins, les pipelines sont écrites en *_Groovy et en _Bash*.

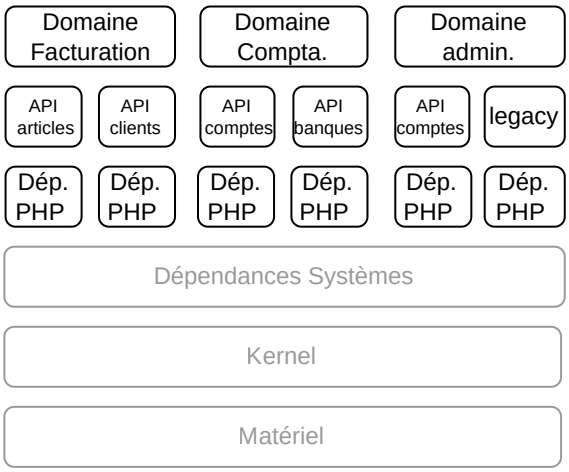
La plupart des micro-services sont construits et déployés via les mêmes *pipelines*.

Observation

D'autres outils – Logstash, Elastic Search, Kibana – permettent de traiter, centraliser et agréger les remontées d'informations (logs essentiellement).

Limitations

Le contexte système unique (serveur) et les automatismes de déploiement communs, apportent une vision claire, mais limitent le choix des technologies système dans la conception de micro-services.



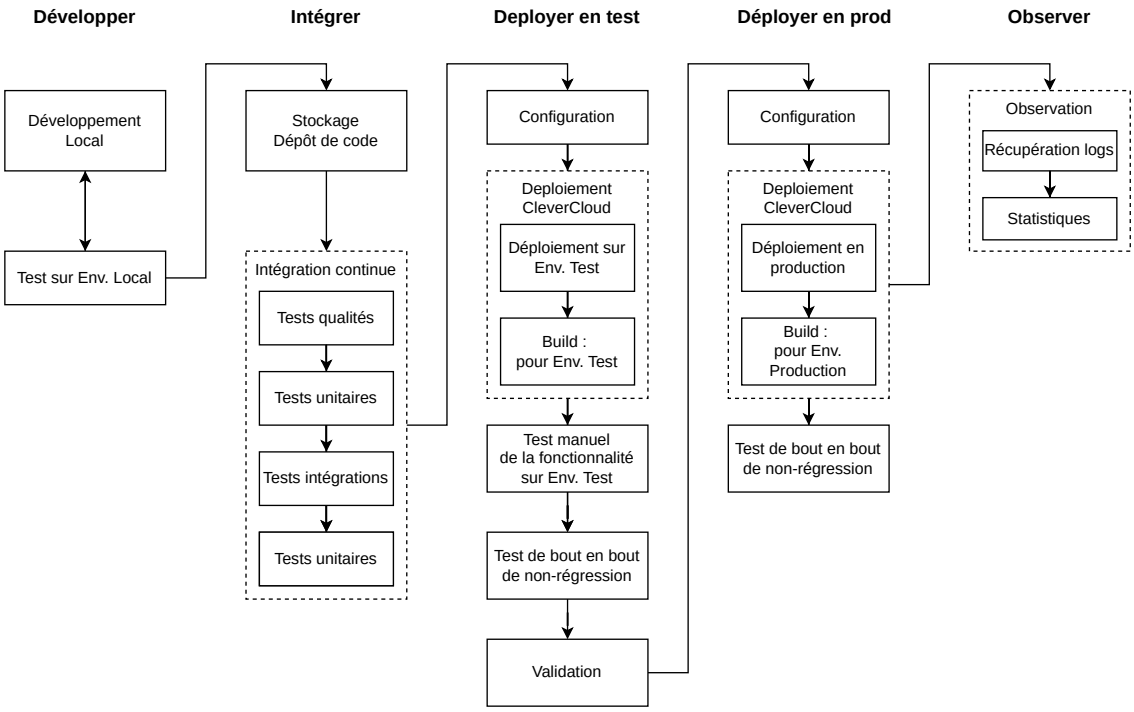
La configuration à l'étape de livraison nécessite de multiplier cette étape par environnement d'exécution (test, pré-production, production...). Comme il y a plusieurs livraisons, il n'y a pas de garantie d'avoir un code identique.

Les modifications de la configuration système, doivent être demandées à l'hébergeur et nécessite un délai non-négligeable pour être mises en place.

2.3.3 - Et à côté, la paie

Le logiciel de gestion de la paie, pour s'affranchir des contraintes systèmes de l'application *legacy*, est déployé sur des serveurs différents. Par ailleurs, ce projet, plus léger, sert de pilotes dans la recherche de nouveaux modes d'hébergement.

Aujourd'hui, la paie est hébergée chez CleverCloud, dans une solution également de type PAAS dont le mode de déploiement est imposé.



Ici, le mode de déploiement est imposé mais clés en main : on configure via l'interface de l'hébergeur et on pousse le code sur un dépôt *git* distant (chez l'hébergeur). Les étapes de build et de mise en production sont alors totalement automatisées par le fournisseur.

Limitations

L'environnement d'exécution est opaque, à la discrétion de l'hébergeur ainsi que les étapes de *build*.

Les marges de liberté sont limitées aux configurations proposées par CleverCloud.

Créer un environnement de test similaire à la production nécessite de commander une plateforme similaire chez Clevercloud. Il se crée une dépendance coûteuse avec le fournisseur cloud.

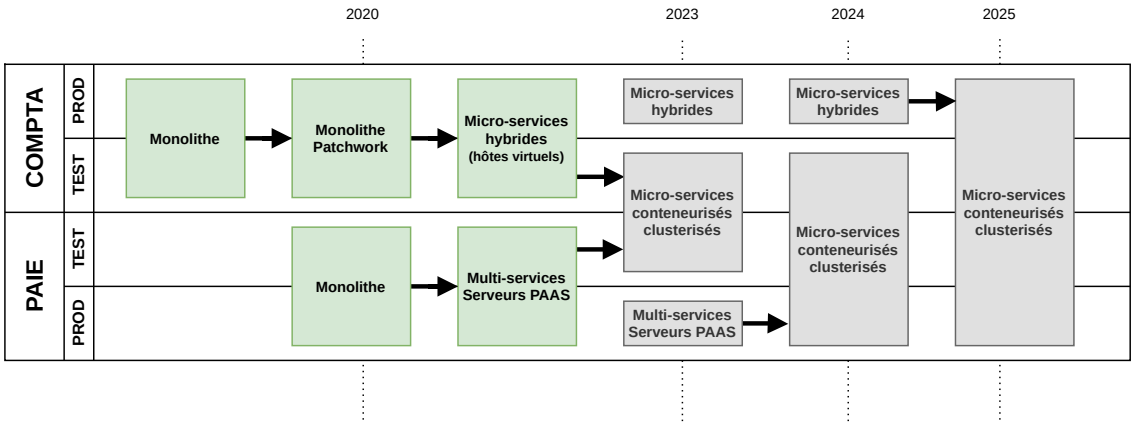
Depuis son lancement, l'application Paie compte maintenant 4 services. Le nombre est limité par le coût. En effet avec ce type d'hébergement, les solutions hybrides à base d'hôtes virtuels ne sont pas possible ; il faut commander un nouveau serveur.

2.4 - premier bilan

Le produit macompta.fr, emprunte la voie des micro-services, une transition sur le long terme. Toutefois certains bénéfices se sont déjà fait sentir comme l'accélération du rythme de déploiement et un premier niveau de découplage.

La nécessité de continuer cette transition est également ressentie ; en particulier pour pouvoir découpler les services au niveau de leurs dépendances systèmes.

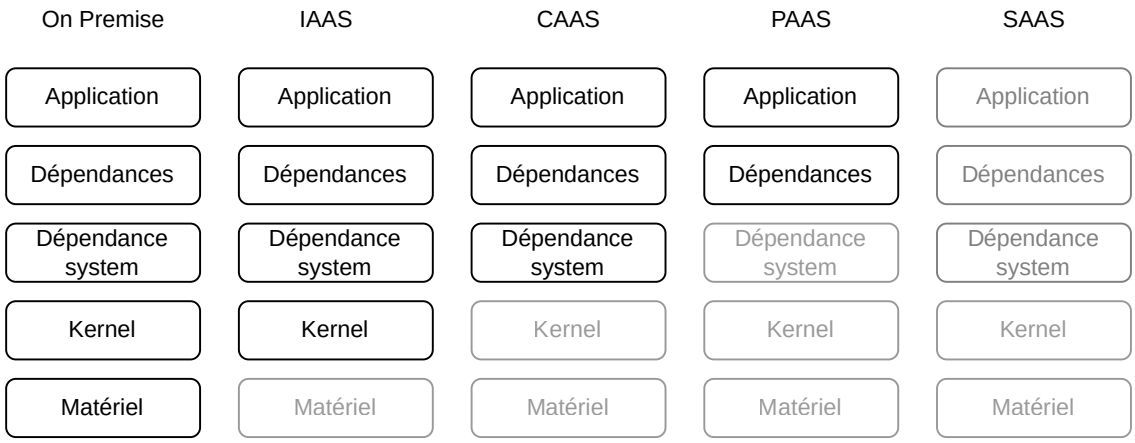
Ce changement va nécessiter de pouvoir intervenir sur des éléments qui étaient jusqu'alors maintenus par l'hébergeur. C'est donc un changement de modèle d'hébergement qui est en cours.



3 - Changement de modèle d'hébergement

On va vu que macompta.fr souhaite changer son mode d'hébergement. Quels sont les différents modèles d'hébergement ? Lesquels concerne macompta.fr dans sa transition et quels sont les enjeux.

3.1 - Les modèles d'hébergement



OnPremise

Correspond au choix de tout traiter en interne, des locaux et de l'alimentation électrique aux données elles-mêmes. Cette solution demande un fort investissement au départ et une maîtrise, en interne, de chaque couche de connaissance.

IAAS, Infrastructure As A Service

C'est un service qui met à disposition des ressources à travers une machine virtuelle. On fait ainsi abstraction de la gestion du matériel. Le client est responsable du bon fonctionnement de son environnement d'exécution.

PAAS, Platform As A Service

Le client paie un service pour disposer d'un environnement d'exécution. Cet environnement peut être plus ou moins spécifique ou standard. Comme le fournisseur est responsable de l'exécution de l'ensemble (runtime), il a tendance à encadrer et donc à limiter la configuration de l'exécution du langage.

C'est la formule actuelle chez macompta.fr : les deux fournisseurs Clevercloud et Claranet proposent des environnements exécutant du PHP. Clevercloud propose un produit très standardisé, en partie configurable. Claranet louent des plateformes sur-mesures, mais très peu configurables directement.

Ce choix a été fait pour des raisons de simplicité et de sécurité. Mais au fur et à mesure de l'internalisant de connaissances opérationnelles – dans un esprit DevOps – ce verrouillage de la configuration système est de plus en plus ressentie comme un frein.

Le CAAS, un entre-deux

Le CAAS, Conteneur As A Service, est né du besoin de faire abstraction du système bas niveau (OS), tout en ayant la main sur la configuration de l'exécution du langage.

Cet intermédiaire entre le IAAS et le PAAS est rendu possible par la conteneurisation. Brièvement, les conteneurs encapsulent les couches supérieures à l'OS (interpréteur PHP par exemple), et partagent un même noyaux d'OS, le kernel. Ce qui est dans le conteneur est à la responsabilité du client ; à l'extérieur, celle de l'hébergeur.

Dit autrement, les conteneurs sont une forme légère de virtualisation qui permet de séparer des applications tout en les gardant sur un même système.

Macompta.fr s'intéresse à ce type d'hébergement afin d'être maître de ses environnements d'exécution et de pouvoir les multiplier. Ceci en se protégeant d'un surcoût lié à la multiplication des *platform* en PAAS ou des contraintes de compétence de l'IAAS.

SAAS, Software As A service

C'est ce que vend macompta.fr. : pour le client, c'est un logiciel directement disponible sans avoir à se pré-occuper de sa maintenance.

3.2 - de nouvelles compétences à acquérir

Ce changement (du PAAS au CAAS) implique de nouvelles compétences qui seront détaillées plus tard.

Conteneurisation

Savoir conteneuriser une application (ou un service). Comment intégrer cette étape dans le flux de production et comment l'automatiser ?

C'est lors de cette étape que sont définies les dépendances systèmes, l'environnement d'exécution pour chaque service. Cette étape demande donc évidemment des compétences systèmes.

Orchestration

Savoir orchestrer l'ensemble de ses conteneurs afin de définir leurs cycles de vies, leurs connexions réseaux, leur accès aux données... Comment intégrer cette compétence à l'entreprise, l'automatiser ?

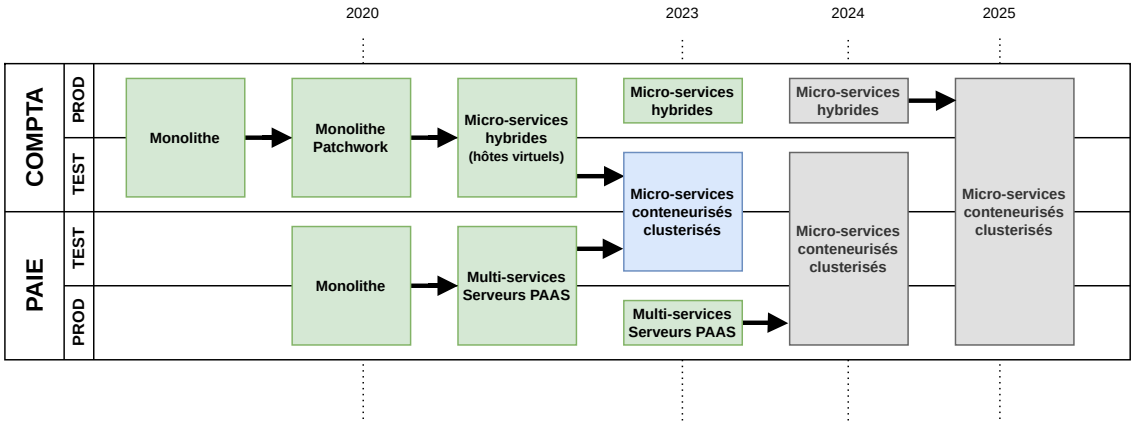
L'orchestration est réalisée par des outils que l'on configure. Dans le contexte du CAAS, ces outils sont *managé*s, ils sont maintenus par l'hébergeur. La configuration, elle est fournie par le client ; dans notre cas macompta.fr.

3.3 - Une transition entamée

3.3.1 - Des environnements de Test différents de la production

L'Ops ayant mis en place les pipelines de déploiement a également initié cette transition.

Pour cela, il a monté *OnPremise*, un environnement qui propose un service CAAS aux développeurs. Ce montage, initialement un simple *POC* a évolué de façon incrémentale. Aujourd'hui, ce CAAS interne héberge les environnements de test.



Cette étape intermédiaire permet d'expérimenter une solution qui groupe l'ensemble des produits macompta.fr, y compris la Paie. Elle a permis de simplifier et d'accélérer les déploiements en Test. Elle crée un premier niveau d'expérience sur ces technologies, pour l'ensemble de l'équipe.

Il faut également noter qu'à cette étape, les environnements de Test de Production ne sont plus *iso*. Il n'est pas souhaitable de rester dans cet état et l'objectif est toujours de tout migrer en CAAS.

Mais avant d'être utilisé en production, et pour garantir, la sécurité, la performance et la fiabilité des produits macompta.fr, ce système a encore besoin d'évoluer et l'entreprise à besoin de plus d'expérience et de recul sur ces technologies.

Pour cela, macompta.fr utilise plusieurs stratégies.

3.3.2 - progressivité dans les applications

Une partie des services macompta.fr est totalement détachée du reste : la gestion de la paie. Étant également plus légère, plus récente et moins utilisée, il a été décidé de la migrer en première.

Dans un second temps, à la lumière de cette nouvelle expérience acquise, les autres applications macompta.fr migreront également.

3.3.3 - progressivité des dossiers

Pour chacune des migrations applicatives précédemment évoquées, une progressivité de migration des dossiers est envisagée. Pour commencer, le compte propre de Macompta qui est un relativement gros dossier, puis migrer par lots les clients.

Cette stratégie soulève des problématiques d'administration des comptes qui n'ont pas encore été abordées.

3.3.4 - un accompagnement

— Il faut éviter que ça casse quand on passe du PAAS au CAAS —

Pour partager les risques, l'entreprise a décidé de se faire épauler dans cette transition. Un prestataire a été sélectionné pour deux missions :

- accompagner la transformation des méthodes de conteneurisation, d'orchestration et de déploiement, actuellement en Test pour être utilisables en Production.
- infogérer, la future production.

Dans un premier temps, il avait été prévu que ce prestataire opère directement les modifications, en commençant par la Paie. Une première itération a été effectuée, mais avec une explosion du budget. Le rôle du prestataire a été ré-arbitré à du conseil/audit, l'objectif étant que j'implémente ces solutions et que le prestataire les valide. Il est également prévu que je reçoive une formation complémentaire sur ce domaine.

Le feu vert du prestataire, pour la mise en production, sera toujours nécessaire afin qu'il prenne l'infogérance.

3.3.5 - Transferts de compétences

L'internalisation de cette compétence repose sur un double transfert de compétences. Premièrement une formation d'un référent interne (moi dans le cas présent, mon poste étant désigné à cette tâche). Deuxièmement, un transfert du référent au reste de l'équipe de développeurs afin de sécuriser cette connaissance dans l'entreprise et afin de donner toujours plus d'autonomie aux équipes sur leurs déploiements.

4 - Le CAAS d'aujourd'hui et de demain, chez macompta.fr

On a vu que macompta.fr adopte une architecture micro-service. Ce changement lui fait changer de modèle d'hébergement pour du CAAS. Ce nouveau mode d'hébergement demande d'internaliser de nouvelles compétences. Cette transition est effectuée au fil de l'acquisition de cette connaissance et est suffisamment avancée pour être utilisée en environnement de Test.

Nous allons maintenant aborder plus spécifiquement ces savoirs. D'une part en détaillant leurs implémentations actuelle chez macompta.fr et d'autre part en décrivant les évolutions préconisées par la future infogérance.

4.1 - la conteneurisation

4.1.1 - Qu'est-ce qu'un conteneur

Il est important de bien comprendre ce qu'est un conteneur et pourquoi cette technologie accompagne souvent l'architecture micro-services.

L'idée du micro-service est de séparer totalement chaque unité afin de les découpler. Pour parvenir à découpler jusqu'au système, intuitivement, on pourrait imaginer monter une machine par service. Cela serait trop coûteux en matériel, en entretien, certainement disproportionné par rapport au besoin du micro-service et n'offrirait pas la souplesse demandée par les équipes de développement.

La virtualisation répond à cette problématique, en permettant de simuler sur la même machine plusieurs systèmes. Il en existe de plusieurs formes.

La VM (virtual machine)

La VM propose une séparation au niveau matériel. Une partie des ressources du système hôte sont allouées à la VM pour simuler les ressources d'une nouvelle "machine ". Cette machine virtuelle dispose de son propre OS. Les VMs sont pensées pour une longue durée de vie.

Cette technologie, très répandue, permet d'exécuter des instances de puissances et d'OS différents, sur une même machine physique. C'est aussi la solution de virtualisation la plus lourde et consommatrice de ressource.

Une VM est généralement ce qui est livré par en IAAS.

Le conteneur système

Ce type de conteneur cherche à répondre aux mêmes usages que la VM, mais avec plus de légèreté. Quand la VM simule un nouveau matériel et nécessite un OS Complet pour le gérer, le conteneur utilise directement le matériel et le kernel de la machine hôte.

Ce gain de légèreté ce fait en contrepartie du choix du noyau de l'OS qui sera forcément identique à celui de l'hôte.

Le conteneur applicatif

Le conteneur applicatif est assez similaire au conteneur système, tous les conteneurs partagent le même kernel. La différence principale est que le conteneur applicatif est pensé pour isoler une application plutôt qu'un système complet.

Typiquement, dans une architecture micros-services, les conteneurs applicatifs ont vocation à n'exécuter qu'un service et ses dépendances.

Très légers, ils doivent être temporaires et sans état. C'est-à-dire qu'ils doivent pouvoir être remplacés, de façon transparente, à n'importe quel moment, par un nouveau conteneur identique.

C'est une technologie qui est arrivée à maturité avec la marque Docker.

C'est ce type de conteneur, dont il est généralement question dans l'offre CAAS et qui intéresse macompta.fr. Cette solution permet d'agir sur l'environnement d'exécution de l'application, sans la responsabilité du reste du système.

4.1.2 - Des images toutes prêtes

Pour exécuter un conteneur, on doit utiliser une image de conteneur. On peut voir ça comme une classe (l'image) et son instance (le conteneur) en programmation objet.

Chaque image est indépendante du contexte d'exécution ; la configuration liée à ce contexte est réalisée au lancement du conteneur.

Il existe une variété d'images prêtes à l'usage et disponibles dans des dépôts (*registry*) dont le plus connu est DockerHub. Les images les plus simples proposent les librairies et outils disponibles dans des distributions Linux ; d'autres proposent des applications complètes (Gitlab ou une base de données par exemple).

Le conteneur Docker est devenu l'un des modes de livraison d'application. Et certainement l'un des plus simples car le service est directement prêt à exécuter.

Ce type d'image est utilisé chez ma compta.fr pour faire tourner des services internes. Le choix de leur utilisation repose sur la simplicité d'installation/réinstallation même pour des applications tournant avec des technologies inconnus à l'entreprise (java par exemple).

Dans le produit macompta.fr, dans les environnements de test, les bases de données (MariaDB, Postgresql, Redis) tournent à partir d'images prêtes à l'usage. Ce sont des images Bitnami, une collection d'images réalisée, optimisée et sécurisée par VMware.

4.1.3 - Fabriquer des images

Dans le contexte de macompta.fr, le but est de produire ses propres images, embarquant chaque micro-service. Bien sûr, Docker permet également cela.

Dockerfile

Il est effectivement possible de construire des images. Ce processus est basé sur un document le *Dockerfile* qui décrit les étapes de création d'une image en se basant sur une autre.

Par exemple :

Ce *Dockerfile*, très basique, utilise comme base une image de serveur PHP Apache et décrit l'installation d'un outils de traitement de texte et la copie de notre code.

On peut voir les *Dockerfile* comme une recette pour cuisiner notre image. Les étapes seront souvent de l'ordre de l'installation de dépendances, de configuration, de copie de code et de compilation.

```
1  FROM php:7.4-apache
2  RUN apt-get update && apt-get install -y vim
3  COPY src/ /var/www/html/
```

Build

Ensuite, ce Dockerfile est utilisé dans l'étape de *build* au bout de laquelle une image docker est construite. Cette étape peut demander de la puissance de calcul, car les étapes décrites dans le *Dockerfile* vont réellement être exécutées.

Il faut bien noter que le *build* créer un modèle et non un système en cours d'exécution. Une image est un objet *froid* qui sert à démarrer des conteneurs (objet *chaud*).

Deux images produites à partir du même *Dockerfile* peuvent être différentes. En effet, si la recette ne change pas, le contexte peut évoluer. Dans notre exemple :

- l'image de base peut changer, car nous n'avons pas précisé la version de correctif (cf SEMVER). Si aujourd'hui la version 7.4.33 serait utilisée, demain ce pourrait être la 7.4.34 ...
- la version de *vim* installée peut changer, car nous n'avons pas précisé de version, ce sera toujours la dernière.
- Notre code peut avoir évolué.

Docker utilise un système de cache, qui permet de ne pas recalculer une étape du *Dockerfile*, si le contexte n'a pas changé.

Stockage

Une image peut être stockée sur la machine qui l'a produite et/ou dans des dépôts privés ou publiques. Plusieurs versions d'une image peuvent être stockées dans le même dépôt. On les distingue par leur *tag*. Par exemple, pour la même image PHP :

- php:7.4-apache est une image embarquant PHP 7.4 et un serveur Apache.
- php:8.2-fpm est une image embarquant PHP 8.2 et un serveur FPM.

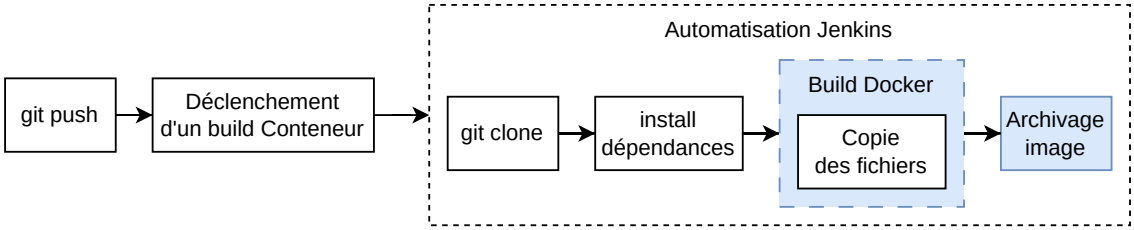
1 image = 1 livraison = X déploiement

Une bonne pratique Docker est de faire des images agnostiques ; c'est-à-dire qu'une image ignore les détails de son/ses futur(s) environnement(s) d'exécution. Les configurations spécifiques à un environnement sont apportées au dernier moment, c'est-à-dire au déploiement du conteneur.

Ainsi, le *build* est clairement séparé du *run* . Le premier dépend uniquement de la version du code, c'est une étape de livraison. Le second dépend aussi de l'environnement, c'est une étape d'utilisation, de déploiement.

Chez macompta.fr

Des images sont produites pour chaque micro-service via des scripts Jenkins.



Ces scripts sont déclenchés au besoin par les développeurs. La plupart des scripts n'utilisent pas réellement de Dockerfile : Jenkins construit le service puis génèrent un Dockerfile, de deux lignes, pour copier les fichiers dans l'image.

Ces builds se basent sur des images maisons communes contenant par exemple un serveur PHP avec l'ensemble les modules utilisés par les tous les micros-services macompta.fr. Ceci permet d'éviter des *Dockerfile* répétitifs, mais contraint tous les services à utiliser la même configuration apache/php.

Ces images sont ensuite stockées sur un *registry* interne : Nexus.

Préconisations pour la Production

Notre prestataire, nous a préconisé de réaliser le maximum des étapes de build pendant le build docker, donc de détailler ces étapes dans un *Dockerfile*. Ceci permet de tirer parti du cache Docker lors du build.

Une autre optimisation soulevée et concernant le *build* est d'utiliser un cache centralisé (via le registry Nexus par exemple), plutôt que les caches individuels des machines de build.

Ces Dockerfiles peuvent être basés sur des images qui ne contiendraient que les dépendances utilisées par tous les micro-services. On optimise ainsi la taille des images.

Si les conteneurs de test nécessitent certains outils de débogage, il est conseillé d'utiliser un Dockerfile en plusieurs partis, différenciant un build de Production d'un Build de Test.

Il nous a également été conseillé de stocker ce fichier avec le code du service. Ainsi, il peut varier au fil des versions. Et ce Dockerfile vient en quelques sortes "documenter" l'installation du service.

Préparation à l'application

Pour implémenter ces préconisations, certaines opérations "système" étaient nécessaires. Pour pouvoir utiliser un cache centralisé, il était préférable de faire mettre à jour, réorganiser certains outils internes. Par exemple, le serveur Gitlab et Nexus (registre multi usage (archive et cache d'image docker, package JS ou PHP...) partageaient une même machine physique tournant avec un OS déprécier (Cent OS). Je les ait relocalisés dans deux VM. Pour cela, il a fallu augmenter la capacité de stockage des serveurs internes et réinstaller un hyperviseur (Proxmox). Ces transferts ont également été l'occasion de monter en version de chaque élément.

Application

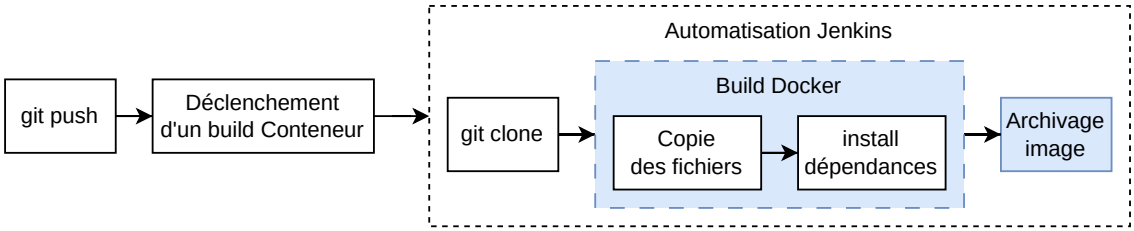
Trois fronts (des applications JS en Vue.JS) ont vu le jour pour faire évoluer l'administration du site. La création des pipelines de déploiements de ses fronts a été l'occasion pour moi d'expérimenter la mise en œuvre de ses recommandations.

J'ai créé une image de base servant des fronts JS. La particularité de celle-ci est qu'il devait être possible d'activer une connexion de l'utilisateur via un LDAP. Ainsi au démarrage des conteneurs, si la configuration l'exige, un script d'initialisation active le module nécessaire et configure des fichiers.

Puis un Dockerfile se basant sur cette image a été créé dans chaque projet. Ainsi, il est clairement indiqué l'utilisation de l'image de base ou d'une future autre au sein du projet.

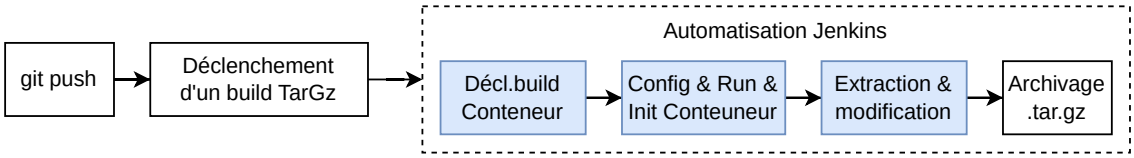
Grâce à ce fichier, chaque projet peut avoir ses propres spécificités ; et pourtant être ensuite "builder" de la même manière.

La *pipeline* Jenkins, ne fait plus que lancer un build docker et l'archiver dans nexus.



Cette occasion a également permis d'expérimenter un rapprochement entre la *pipeline* de Test (conteneurisé) et la production (non-conteneurisée). J'ai créé une *pipeline* de production qui commence par utiliser celle de test et donc créer un conteneur. Ce conteneur est alors configuré avec les variables de Production et initialisé. Son contenu est ensuite extrait puis modifié pour être compatible avec la production.

Ainsi les différences entre test et production, s'il y en a, son clairement explicitées dans la *pipeline*. Et ce qui est identique a été testé avec le conteneur.



4.1.4 - configurer des conteneurs

On l'a évoqué précédemment, l'image d'un conteneur doit être "générique", elle doit être configurée pour s'exécuter comme voulu, dans un contexte donné.

Plusieurs types de configurations

Toutes les configurations qui ne dépendent pas de l'environnement d'exécution ont normalement été injectées lors du build.

Ainsi, on n'a pas besoin de connaître toute la technicité de l'application pour la lancer. Le conteneur apporte une couche d'abstraction de notre service.

Il faut donc apporter la configuration qui dépend de l'environnement d'exécution. On en distinguera deux types.

La configuration applicative, influe sur le fonctionnement de l'application. En définissant des identifiants d'accès par exemple.

La configuration d'orchestration qui va définir comment le conteneur prend place dans l'environnement d'exécution. Cette configuration sera traitée dans la partie Orchestration.

Configuration applicative

Pour modifier la modification de l'application, on peut lui injecter des informations. L'un des moyens, le plus simple à utiliser et à documenter et l'injection de variables d'environnements.

Pour des jeux de données plus structurés et conséquents, on peut également lui injecter des fichiers (de configurations par exemple).

Une difficulté est que certaines de ces informations sont critiques et confidentielles : les secrets. Ils doivent être stockés de façon spécifique en respectant une gestion des droits.

```
# Dans cet exemple, de configuration "manuelle", on lance notre service "admin".

# On passe a notre application la variable LDAP_ENABLE pour activer la connection via un LDAP

# On lui passe également un fichier 'ldap.conf' pour configurer le fonctionnement de la connexion au LDAP.
docker run \
    --env LDAP_ENABLE=true \
    --volume ldap.conf:/config/ldap.conf:ro \
    serveurAdmin
```

Macompta.fr , configuration par .env

La plupart des applications et services chez macompta.fr sont configurés en utilisant un fichier de propriété : .env

Les services, lisent les variables dans le fichier .env, mais peuvent également les lire dans les variables d'environnement.

```
# Exemple de .env
# ces fichiers peuvent facilement servir à définir des variables d'environnement
LDAP_ENABLE=true
LDAP_URL=https://example-ldap.fr
LDAP_PASSWORD=PADL-PA2Q8CE
```

Des propriétés centralisées

Les variables (ou propriétés) de configuration et secrets sont centralisés dans une application open-source développée par Sncf Connect : Hesperides.

Cet outil, accessible en intranet, permet d'une part de renseigner via une UI des variables et d'autre part de construire des *templates*. L'application génère ensuite, à la demande, les fichiers de configuration, en substituant les variables par leurs valeurs dans les *templates*.

Par ailleurs, cet outil permet une gestion des droits et une protection des secrets.

L'essentiel des sorties sont des fichiers de propriétés, des .env .

Initialisation

Il est possible d'exécuter des scripts d'initialisations, depuis le conteneur, qui se lancent à son démarrage et avant l'appel du service (serveur par exemple).

Les conteneurs de macompta.fr exécutent pour la plupart un script qui leur est injecté au démarrage. Ce dernier récupère le fichier de configuration sur un volume réseau et réalise des opérations installation, voir de build sur certaines applications comme le *legacy*.

Préconisation pour la production

Notre prestataire a apporté plusieurs conseils qu'il a commencé à mettre en œuvre sur l'application Paie.

Premièrement, les étapes de build réalisées lors du démarrage du conteneur et qui ne dépendent pas du .env, doivent être réalisées lors du build du conteneur. Si ces *build* dépendent des variables d'environnement, il nous est conseillé de les en découpler. Ceci afin d'accélérer le démarrage du conteneur et d'éviter des piques de charge lors des déploiements.

Une autre recommandation est d'injecter directement la configuration via des moyens standards ; plutôt qu'injecter un script... Ainsi, la configuration nécessaire à l'exécution du conteneur est plus explicite.

4.2 - Orchestration

L'architecture micro-service implique une multitude d'entités. Il faut définir comment chacune d'entre-elles va prendre place dans l'ensemble que forme l'application macompta.fr. Cette gestion des services s'appelle l'orchestration et il existe plusieurs outils permettant de l'aborder.

Chez macompta.fr, le choix s'est porté sur Kubernetes. C'est un orchestrateur open-source, très complet et évolutif, initialement développé par Google.

Pour les environnements de test, deux clusters sont maintenus de bout en bout, OnPremise, sur les serveurs internes de l'entreprise.

Pour la production, il est projeté d'utiliser un Kubernetes *managé* par l'hébergeur. C'est donc bien un environnement prêt à exécuter des conteneurs qui est loué, du CAAS. Toutefois, même sous une forme managée, elle nécessite une somme de connaissance pour la configurer.

Cette solution a été choisie pour sa couverture quasi-exhaustive des cas d'utilisations. C'est également une solution leader et donc proposée par de multiples hébergeurs ; il y a donc moins de risque de dépendances à ce prestataire comparé à des solutions propriétaires proposées par les majors du domaine (AWS, AZURE, GCP, ...).

4.2.1 - Des clusters internes

Kubernetes est une solution hautement paramétrable et nécessitant l'installation d'une série de composants. Pour simplifier son installation, de grandes marques du secteur proposent des "distributions" Kubernetes. Macompta.fr utilise MicroK8s, pour ses deux clusters d'environnements de test. C'est une distribution légère développée par Canonical.

J'ai été amené à intervenir de différente manières sur ces clusters. Premièrement pour les rétablir quand ils ont saturé (de différentes marnières). J'ai également eu à "remonter" l'un d'eux lorsqu'il est tombé. Finalement, j'ai également monté un autre cluster, avec une autre distribution (RKE2 de Rancher), tournant sur plusieurs nœuds (machines) afin de reproduire des cas plus proche de la future production : haute disponibilité, base de données sur des machines

différentes... Ces opérations, même si elles ne m'incomberont pas en production me permettent une meilleure connaissance de Kubernetes et de ses contraintes de fonctionnement.

4.2.2 - Prérogatives de l'orchestration

Il est nécessaire, pour bien comprendre les enjeux, de détailler les différentes responsabilités des systèmes d'orchestration de conteneurs.

Toutes ces prérogatives sont des opérations que l'on pourrait imaginer assurer manuellement. Elles peuvent nécessiter une surveillance ou/et une planification. Par exemple, lorsque l'on veut garder une continuité de service. Dans un contexte de multiplication de micro-services, réaliser ces tâches manuellement deviendrait trop chronophage, pour trop peu de valeur ajoutée.

C'est pourquoi les orchestrateurs se proposent de les d'automatiser.

Cycle de vie

Le cycle de vie comprend les phrases de démarrage, de redémarrage en cas de panne, de remplacement lors d'une nouvelle version, de réplication pour plus de performance ou de disponibilité, de lancement périodique de tâche...

Par exemple, sur les environnements de test, Kubernetes permet de redéployer des services sans rupture de disponibilité. Pour cela, il démarre les nouveaux conteneurs ; lorsqu'ils sont tous prêts, il y redirige le trafic, coupe et supprime les anciens.

Une dizaine de tâches periodiques sont réalisés via l'exécution de CronJob (mécanisme Kubernetes comparable aux *cron* Linux). Certaines tâches ne sont exécutées qu'en production car non-implémentées dans la configuration Kubernetes de Test.

Les applications PHP, servies par Apache, ont la particularité de ne pas réellement planter. En effet, généralement, le code PHP n'étant pas utilisé en dehors d'une requête : une erreur dans le code cassera uniquement la requête en cours. Par contre dans le cas de mauvaise configuration, Apache peut ne pas démarrer.

Ainsi, chez macompta.fr le principal contrôle demandé à l'orchestrateur est de vérifier que le conteneur a démarré correctement.

Réseau

L'orchestrateur va également être chargé de rendre accessibles les services, en interne et ou en externe. Éventuellement de faire de la répartition de charge lorsque le conteneur du service est répliqué. On va retrouver tous les rôles des composants d'un réseau : allocation d'IP, résolution de nom, proxy...

Actuellement, tous les micros-services macompta.fr sont accessibles de l'extérieur. Ils s'appellent d'ailleurs entre eux en utilisant leurs adresses publiques.

Ils sont accessibles via un proxy inverse, qui assure le routage et l'encryptions SSL extérieures : le proxy-inverse reçoit toutes les requêtes via HTTPS et redirige vers le bon service en HTTP.

Dans les clusters de test, ceci est réalisé en utilisant un Ingress-controller (reverse-proxy) et des Ingress (configurations) qui sont des concepts natif de Kubernetes.

Stockage

La source principale de donnée des applications macompta.fr est les bases de données. En cluster, des mécanismes permettent de changer le container de bases de données tout en concernant une continuité dans le contenu.

Pour cela, les bases doivent stocker leurs fichiers dans des volumes performants et persistants. Ceux-ci sont souvent des stockages Blocs directement connectés au Nœud. Ceci impose de spécifier le nœud de démarrage des bases de données.

D'autres stockages plus lents, mais pouvant être partagés entre nœuds sont expérimentés chez macompta.fr ; en particulier des stockages objets (très économiques) pour remplacer des NFS sur du stockage bloc (plus coûteux).

Configuration / Secret

Les orchestrateurs permettent de transmettre des configurations à des conteneurs via leurs variables d'environnement ou via l'injection de fichier. Kubernetes propose également de stocker configurations et secrets dans le cluster.

Actuellement, la configuration des conteneurs se fait via une troisième voix, moins standard, mais faisant la transition avec les méthodes utilisées pour le monolithe.

Pour rappel, les configurations et secrets "applicatifs" sont stockés dans un service interne (Hesperides). Un script d'initialisation (Bash) est injecté dans le conteneur à son démarrage. Ce dernier, récupère et installe un fichier de configurations depuis un lecteur distant (stockage objet).

4.2.3 - orchestrateur déclaratifs

La configuration d'orchestration

Les technologies comme docker ou Kubernetes permettent de démarrer directement des conteneurs, manuellement, via leur CLI (Command Line Interface).

Pour cela, on configure le conteneur en ajoutant des paramètres à la commande. On apporte à la fois de la configuration applicative (qui aura un effet à l'intérieur du conteneur) et de la configuration d'orchestration qui va déterminer la place du conteneur dans son environnement d'exécution.

C'est une méthode de configuration manuelle d'un cluster qui est plutôt utilisée pour des tests.

```
# Dans cet exemple, de configuration "manuelle", on lance notre serveur admin.

# On lui défini des limites de ressources CPU et RAM
# On le connecte à un réseau
# On demande à ce qu'il soit redémarré en cas de crash
docker run \
    --cpus 1.5 \
    --memory 200m \
    --network admin_network \
    --restart always \
    serveurAdmin
```

Le problème de cette méthode et qu'elle peut être laborieuse, répétitive et source d'erreur par exemple si on oublie un paramètre dans la commande.

Des Configuration *as code*

Kubernetes propose de consolider ces configurations dans des fichiers YAML, les *manifests*. On y décrit l'état du système désiré. Chez macompta.fr, ces fichiers sont conservés et versionnés dans des dépôts Git.

Certains orchestrateurs comme Docker Compose, permettent de décrire un système en utilisant très peu de concepts et donc un document unique et de taille réduite.
Kubernetes permet une configuration plus fine en utilisant une multitude de concepts. Il faut souvent plusieurs documents YAML pour déployer un service.

Par exemple, les conteneurs sont démarrés dans des *pods*; des *replicaset* vérifient la réplication de ces *pods* ; des *deployment* gèrent le cycle de vie de ces *replicasets* ; des *services* gèrent les accès aux *pods*... Chaque concept a son manifeste.

L'utilisation de Kubernetes génère ainsi la création d'une multitude de documents de configurations.

Helm et les Charts

Concernant Kubernetes, les manifestes embarquent des paramètres essentiellement d'orchestration, mais ils contiennent également des données de configurations des conteneurs. Certains paramètres vont varier d'un environnement d'exécution à un autre, d'autres non.

Séparer ces différents types d'information permettrait de simplifier la configuration pour le déploiement.

Pour cela, macompta.fr a choisi l'application Helm. Cet outil permets de regrouper dans une *chart* tous les manifestes Kubernetes nécessaires à l'orchestration d'un ou plusieurs services, applications, etc... La granularité est a définir par le concepteur de la *chart*. Aujourd'hui, chez macompta.fr, on compte une *Chart* par domaine ou groupe métier (admin, compta, facturation...).

Cette outil permet également de séparer d'un côté des *values* : les valeurs destinées à être modifiées par l'utilisateur (Développeur ou Ops), et de l'autre des *templates* de manifeste Kubernetes. Les fichiers *values* sont écrits en YAML et leur structure est à la discrétion du concepteur.

Au déploiement, Helm utilise les *values* qu'il injecte dans les *templates* pour générer les manifestes finaux. Ces fichiers finaux sont directement transmis à l'API Kubernetes.

Ce système, apporte une couche d'abstraction et permet de configurer très simplement un service.

Actuellement, chez macompta.fr, la configuration qui dépend de l'environnement d'exécution est donc séparée à deux endroits : Hespérides pour les paramètres applicatifs, les *Charts* Helm pour les paramètres d'orchestration.

Dans les faits, certains paramètres rentrent dans les deux catégories, nécessitant une répétition et exposant à un risque d'erreur.

4.2.4 - préconisation pour la production

Séparation des processus

Certains conteneurs exécutent plusieurs processus principaux, par exemple un serveur HTTP et un processus de fond. Or, il est préférable de les séparer dans des conteneurs différents, pour en permettre une gestion plus spécifique et un comportement du conteneur plus prévisible.

Exposer les configurations via les concepts natifs (configmap, secret)

Les avantages de cette méthode seraient la centralisation de toutes les configurations en un lieu unique et la suppression de scripts liés à l'extraction des configurations depuis Hespérides.

Actuellement, les conteneurs récupèrent directement leur configuration dans un volume partagé avant de s'initialiser. Pourtant, Kubernetes permet de stocker, directement dans le cluster des configurations et des secrets et de les injecter aux conteneurs cible, sous forme de fichiers ou de variables d'environnement. Utiliser ces outils permettrait de simplifier et de normaliser l'initialisation des conteneurs.

Les *configmaps* et les *secrets* seraient générés par les *charts* Helm en question. Ce qui centraliserait les configurations en un seul endroit. Les secrets devront être cryptés avant d'être stockés avec les autres configurations.

Ceci permettrait également de partager certaines configurations communes entre services et limiter ainsi la répétition d'informations présente dans Hespérides.

Observabilité

Vérifier la qualité d'un micro-services passe aussi par l'observation de son fonctionnement : charge, temps de requête, erreurs, pic de fonctionnement...

Un des défis du micros-service est également de tracer une requête à travers différents micro-services.

Chez macompta.fr, l'observabilité est implémentée en production avec la suite ELK (Elastic Search, Logstash, KIbana). Cette implémentation n'as pas été adaptée sur les clusters de test et reste à faire.

Par ailleurs, en environnement conteneurisé, le standard est d'envoyer les logs sur les sorties standards des containers (consoles). Le gestionnaire de conteneur, centralisant les logs.

Actuellement, chez macompta.fr, la plupart des logs sont dirigés vers des fichiers.

Possibilité multi-node

Kubernetes peut gérer plusieurs nœuds. Un nœud peut être vu comme une machine d'exécution. Kubernetes va, en fonction de nos instructions et de la disponibilité de chaque nœud, choisir où il démarre les *Pods*. Ce système permet une mise à l'échelle globale plus simple : on ajoute un nœud et les *Pods* se répartissent en fonction du besoin. Il offre également au cluster une bonne résilience : si un nœud tombe, les *Pods* sont redémarrés sur un autre.

Chez macompta.fr, dans l'état actuel, les services ne peuvent pas changer, ni être réparti sur plusieurs nœuds. Ceci, car ils stockent et partagent des fichiers sur le disque de leur nœud , entre autres pour les sessions.

Des alternatives aux fichiers sont envisageables comme l'utilisation de base type Redis ; mais ne sont pas encore implémentées.

Séparation des services

Kubernetes apporte le concept de *Pod*, qui est la plus petite unité gérée par le système. Une particularité est qu'un *Pod* peut contenir plusieurs conteneurs.

Des conteneurs partageant le même *Pod* partagent la même connexion réseau, peuvent communiquer directement et simplement les uns avec les autres. On peut comparer des conteneurs partageant le même *Pod* à des processus d'exécutant sur le même système.

Actuellement, chez macompta.fr, les services sont regroupés dans des *Pods* par pôles métiers. Cette solution avait été choisie de limiter le nombre de *Pods*. En effet, comme vu précédemment, chez macompta.fr les *Pods* doivent être liés à un nœud ; or les nœuds ne peuvent héberger que 110 *Pods* (limitation de Kubernetes).

Une fois, que les conteneurs macompta.fr pourront être répartis sur plusieurs nœuds, cette limite du nombre de *Pods* pourra être levée en ajoutant de nouveaux nœuds. Ils sera alors possible de respecter la bonne pratique d'un service dans un *Pod*, . Il faudra toutefois trouver des solutions pour se retrouver dans cette multitude de *Pods* (*namespace*, convention de nommages pas exemple).

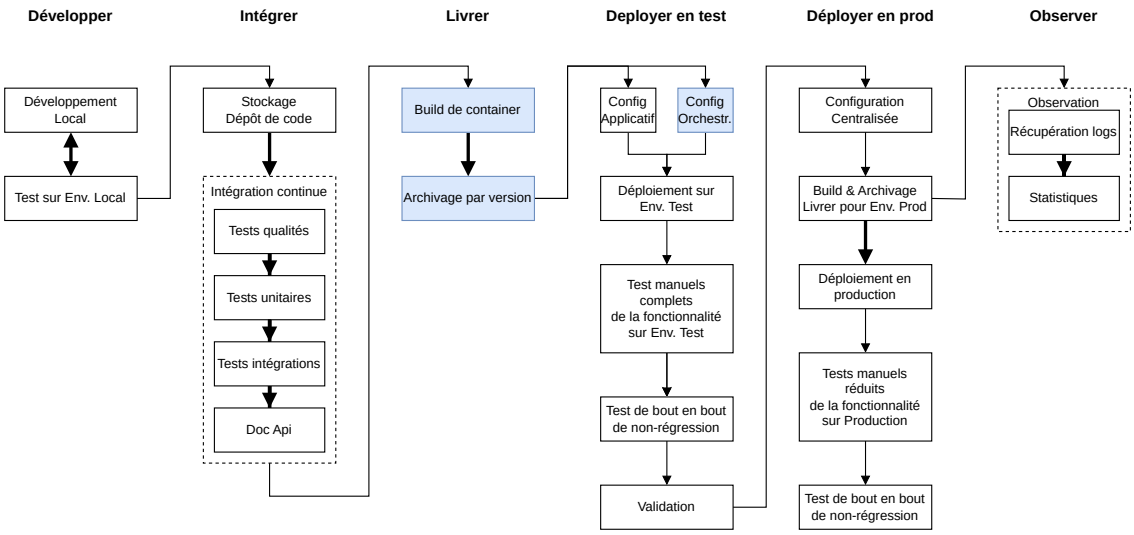
Gestion des droits

Actuellement, chez macompta.fr, tous les développeurs ont accès à la totalité des clusters, en lecture, comme en écriture. Ceci n'est pas gênant dans un environnement de test, mais devrait être plus contraint dans un contexte de production.

Il sera donc nécessaire d'introduire une gestion des droits pour le déploiement, la configuration et la gestion des clusters.

4.3 - chaîne de déploiement

4.3.1 - actuellement



Un conteneur, des déploiements

La différence la plus notable avec le mode de déploiement précédant (encore utilisé pour la production) est la séparation de la livraison (build conteneur) et du déploiement. En effet, l'image produite lors du build est indépendante de tout environnement d'exécution. Ainsi, chaque image peut être déployée dans plusieurs environnements. L'objectif étant ultérieurement d'utiliser la même image pour les environnements de tests et la production.

Détails du déploiement

Comme évoqué précédemment, le build de conteneur est réalisé avec Docker, mais par une *pipeline* Jenkins. L'image produite est stockée dans le *registry* interne sur le serveur Nexus.

Le déploiement est lancé via une interface web qui permet de sélectionner les versions des services.

Cette interface déclenche une *pipeline* Jenkins qui :

- réalise un pré-rendu de la *charts* Helm pour récupérer l'identifiant d'une configuration Hespérides.
- charge la configuration en question depuis Hespérides et l'envoie dans un stockage distant (un stockage objet type S3).
- déploie la *Chart* Helm vers le cluster cible, en remplaçant les numéros de version par ceux envoyés par l'interface web.

Le cluster télécharge alors l'image du/des conteneur(s) concernés et les démarre.

Lors de leur démarrage, ces derniers téléchargent leurs configurations depuis le stockage distant.

4.3.2 - Préconisation pour la production

Plusieurs évolutions sont prévues avant de passer en production.

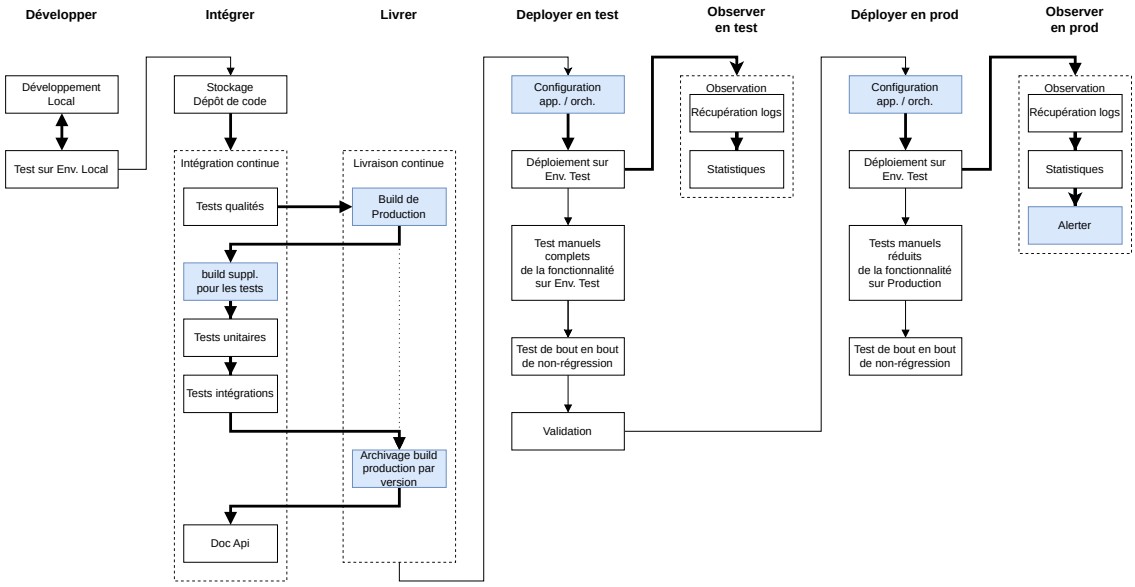
Il est prévu de migrer les *pipelines* de Jenkins vers Gitlab CI/CD. Les fichiers de Pipeline de ce dernier se stockent en général directement avec le code. Ainsi, comme pour le Dockerfile, toutes les informations de CI/CD sont stockées avec le code et peuvent suivrent les mêmes évolutions (branches, version...). Le choix de cette solution permet de lier très naturellement le flux *git* au flux de CI/CD.

Ainsi, il pourra être envisagé d'automatiser certaines tâches comme par exemple les build qui pourraient être déclenchés à chaque changement de code. L'image du build pourrait alors être utilisée comme base pour les tests unitaires et/ou d'intégrations.

Il est également projeté d'unifier la configuration applicative (dans Hespérides) et d'orchestration (Chart Helm). Cette fusion s'opérerait directement au sein des *Charts*. En effet, la structure des fichiers *values* d'Helm étant totalement libre, il est possible de mettre au point une organisation adaptée à macompta.fr.

Afin de permettre aux *Charts* Helm, de stocker des secrets sans les exposer sur le dépôt *git*, il est envisagé d'utiliser des solutions de cryptage à la volée de certaines valeurs. Notre futur hébergeur utilise et préconise Sops.

Enfin, il nous faut absolument récupérer et traiter l'ensemble des logs et métriques produites par nos clusters. Un autre besoin important est également la capacité à pouvoir suivre une même requête à travers divers micro-services.



5 - Ceci n'est pas une conclusion

Ce mémoire traite d'une transition sur le long terme, beaucoup plus longue que ma période d'alternance : elle a démarré avant et finira après.

C'est une évolution importante pour Macompta.fr. En effet, cette entreprise évolue en permanence – c'est un de ses traits de caractère – et cette transition vers le micro-service a pour objectif de permettre de meilleures évolutions futures.

Toutefois, jusqu'à présent, mon impact dans cette transition a été principalement préparatoire : mes actions ont été de l'ordre de la maintenance ou de l'extension de l'existant : de l'alimentation électrique des serveurs internes, à la configuration 'indirecte' du serveur de production, en passant par l'édition de pipeline, la gestion de VMs d'outils internes et bien sûr le maintient des clusters locaux.

Par ces actions, j'ai acquis une certaine connaissance de l'existant et une compréhension des raisons et de l'historique de cette transition. Ainsi, ce mémoire traite assez peu des tâches concrètes que j'ai pues effectuer au quotidien, mais il témoigne d'un travail de réflexion, de compréhension et de formalisation de ce changement.

Comme, Macompta.fr m'invite à continuer avec eux, cette aventure, ce mémoire me permet d'aborder ce poste avec vision plus claire et plus construire.