Architecture Serverless

De la théorie à la pratique

Steve Houël

LIVRE BLANC

NOVEMBRE *2017*

IPPON
Discovery to Delivery

SOMMAIRE

1	L'auteur	**4**
2	A propos de Ippon Technologies	**5**
2.1.	Nos solutions	**6**
3	Licence	**7**
4	Présentation du livre blanc	**8**
5	Qu'est ce que le Serverless ?	**9**
5.1.	«Serverless» ≠ sans serveurs ?	**10**
5.2.	Quelques cas d'utilisations	**10**
5.3.	Ce qui n'est pas «Serverless»	**14**
6	Le Serverless : Licorne ou désillusion	**16**
6.1.	Avantages	**16**
6.2.	Les inconvénients	**20**
4	Le FaaS dans le détail	**25**
6	Faire du Serverless	**30**
6.1.	BaaS et nos amis les Cloud Providers	**30**
6.2.	Et le FaaS dans tout ça !	**32**
7	Pour conclure	**36**

1 — L'AUTEUR

Steve Houël

Solution Architect chez Ippon Technologies. J'interviens sur de nombreux domaines techniques (conseil, conception, formation,..). Mes domaines de prédilection sont les architectures microservices et tout ce qui tourne autour du DevOps & Cloud.
Speaker lors de Meetups et membre de la Team Jhipster à mes heures perdues. Pour me résumer, je pense donc que je suis «Architect as code ».

Découvrez tous ses articles sur **blog.ippon.fr**

Comment faire du Serverless ?

Les architectures Serverless

2 — A PROPOS DE IPPON TECHNOLOGIES

Ippon est un cabinet de conseil en technologies, créé en 2002 par Stéphane Nomis, sportif de Haut-Niveau et un polytechnicien, avec pour ambition de devenir leader sur les solutions Digitales, Cloud et Big Data.

Ippon accompagne les entreprises dans le développement et la transformation de leur système d'information avec des applications performantes et des solutions robustes.

Ippon propose une offre de services à 360° pour répondre à l'ensemble des besoins en innovation technologique : Conseil, Design, Développement, Hébergement et Formation.

Nos équipes tirent le meilleur de la technologie pour transformer rapidement les idées créatives de nos clients en services à haute valeur ajoutée. Elles accompagnent à la fois les grands groupes (Axa, EDF, Société Générale, LVMH,...) et les champions français de l'industrie numérique (CDiscount, LesFurets.com, Aldebaran....) dans leurs innovations.

Nous avons réalisé, en 2016, un chiffre d'affaires de 24 M€ en croissance organique de 20%. Nous sommes aujourd'hui un groupe international riche de plus de 300 consultants répartis en France, aux USA, en Australie et au Maroc.

Ippon votre partenaire
End to End

Conseils
Formations

UX
design

Réalisation
Agileté/DevOps

Hébergement

2.1. Nos solutions

Fort d'un savoir faire technologique riche et de compétences certifiées dans le domaine du Cloud Computing, Ippon investit constamment dans la veille technologique pour identifier dès aujourd'hui les solutions les plus performantes, sécurisées, rentables de demain. Ippon a développé dans ce cadre une expertise autour du Hosting et du Cloud et dispose aujourd'hui de références solides dans ces domaines. Celles-ci sont renforcées par ses partenariats avec de grands fournisseurs de services Cloud.

UNE IDÉE, UN POC
UN PRODUIT

Idée

Lean Startup

Conseil POC

Product Backlog

projet par itérations

équipe agile

projet run

projet run

infogérance cloud

Vos besoins, nos solutions

3 — LICENCE

Ce document vous est fourni sous licence Creative Commons Attribution Share Alike. Le texte ci-dessous est un résumé (et non pas un substitut) de la licence.
Plus d'informations sur www.creativecommons.org/licenses/

Vous êtes autorisé à :

- Partager — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- L'offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.
- Selon les conditions suivantes :

Attribution :

- Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.
- Pas d'utilisation commerciale — vous n'êtes pas autorisé à faire un usage commercial de cette œuvre, tout ou partie du matériel la composant.
- Pas de modifications — dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'œuvre originale, vous n'êtes pas autorisé à distribuer ou mettre à disposition l'œuvre modifiée.
- Pas de restrictions complémentaires — vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser l'œuvre dans les conditions décrites par la licence.

Notes :

- Vous n'êtes pas dans l'obligation de respecter la licence pour les éléments ou matériel appartenant au domaine public ou dans le cas où l'utilisation que vous souhaitez faire est couverte par une exception.
- Aucune garantie n'est donnée. Il se peut que la licence ne vous donne pas toutes les permissions nécessaires pour votre utilisation. Par exemple, certains droits comme les droits moraux, le droit des données personnelles et le droit à l'image sont susceptibles de limiter votre utilisation.

4 — PRÉSENTATION DU LIVRE BLANC

Nous le savons, le monde informatique est en constant changement. Que ce soit les évolutions matérielles, l'avènement de l'IoT dans les objets de tous les jours ou encore les services proposés par les Cloud Providers. Le monde du développement logiciel n'échappe pas à cette tendance. Outre les nouveaux frameworks Web qui sortent plus vite que notre courbe d'apprentissage, les architectures applicatives elles aussi se voient repensées, remaniées. Il y a encore peu de temps, nous pensions tous qu'un bon vieux monolithe était "LA" solution simple, efficace et pas chère.

Hors avec l'émergence de la conteneurisation et du DevOps, un nouveau panel d'architectures a vu le jour. Nous avons ainsi hérité des architectures Microservices. Simples, scalables et rapides à développer lorsque l'on se base sur des générateurs tels que JHipster (***http://www.jhipster.tech***), elles ont ouvert de nouvelles voies dans le développement d'applications Web. Mais comme toute nouvelle architecture, celle-ci vient avec son lot de contraintes. L'une d'entre elles est la gestion de l'infrastructure. Même si le DevOps et la conteneurisation ont apporté beaucoup dans cette problématique, ils ne l'ont pas résolue pour autant.

Aujourd'hui, une nouvelle architecture fait parler d'elle dans le monde de l'IT, c'est le «Serverless»

5 — QU'EST CE QUE LE SERVERLESS ?

Les architectures sans serveur se réfèrent à des applications qui dépendent de manière significative de services tiers (connue sous le nom de Backend en tant que service ou «BaaS») ou sur un code personnalisé exécuté dans des conteneurs éphémères (Function as a Service ou «FaaS»), le fournisseur de ce service le plus connu est actuellement AWS Lambda.

De nos jours la migration de nombreuses fonctionnalités côté FrontEnd nous a permis de supprimer nos besoins de serveurs "Always On". Selon les circonstances, de tels systèmes peuvent réduire considérablement le coût et la complexité opérationnels et ainsi se résumer à payer uniquement les frais d'utilisation (bande passante, volume de stockage). Ainsi on ne paie que ce que l'on consomme (connu aussi comme le pay-as-you-go).
Comme de nombreuses tendances dans le logiciel, il n'y a aucune vision claire de ce qu'est «Serverless», et cela n'a pas été aidé par le fait qu'il s'agit vraiment de deux domaines différents, mais qui se chevauchent :

- «Serverless» a d'abord été utilisé pour décrire des applications qui dépendent de manière significative ou totale à des applications / services tiers ('dans le cloud') pour gérer la logique et l'état du serveur. Ce sont généralement des applications «client complète» (pensez à des applications Web en une seule page ou à des applications mobiles) qui utilisent le vaste écosystème de bases de données accessibles sur le cloud (comme Parse, Firebase, AWS DynamoDB, ...), les services d'authentification (Auth0, AWS Cognito), etc. Ces types de services ont été précédemment décrits comme Backend as a service (j'utiliserai le terme BaaS comme abréviation dans le reste de ce livre blanc).

- «Serverless» peut également symboliser des applications où une certaine quantité de logique serveur est toujours écrite par le développeur, mais contrairement aux architectures traditionnelles (exemple : Monolithique), elle est exécutée dans des conteneurs stateless qui sont

déclenchés par le biais d'événements, éphémères (uniquement une invocation) et entièrement gérés par une tierce party. Ceci coreespond au concept de terme Function as a service ou Faas. AWS Lambda est l'une des implémentations les plus populaires de FaaS à l'heure actuelle, mais il y en a d'autres (Google Functions).

Nous parlerons principalement de la deuxième définitioncar elle est plus récente et possède le plus de différence avec la vision que l'on a d'une architecture technique traditionnelle (elle est tout simplement plus hype !!)

5.1. «Serverless» ≠ sans serveurs ?

Le terme «Serverless» est source de confusion car, dans de telles applications, il y a à la fois des notions de matériel et de systèmes. La différence avec les approches basiques est que l'entreprise créant et prenant en charge une application dite «Serverless» ne s'occupe pas de ces deux aspects, elles externalisent cela à un fournisseur (AWS, Google, …) et ainsi se concentrent uniquement sur la partie fonctionnelle de l'application.

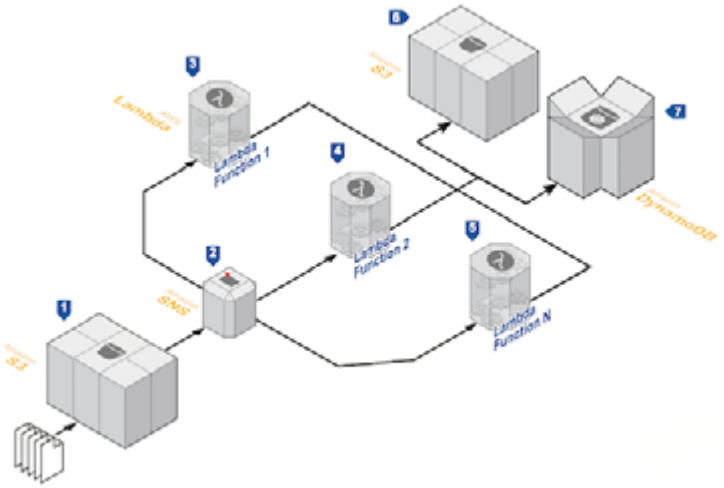
5.2. Quelques cas d'utilisations

Etudions maintenant des cas concrets d'utilisation du «Serverless». Vous pourrez trouver plus de détails sur ces architectures et même des exemples réels en vous rendant sur le Github awslabs (***https://github.com/awslabs***).

- **Traitement de fichiers temps réel**

La première architecture que nous verrons concerne le traitement des fichiers. Celle-ci s'applique très bien lorsque que l'on a besoin d'avoir différents traitements de la donnée source et de générer plusieurs formats en sortie.

Dès lors, nous pouvons utiliser une concaténation de plusieurs services et effectuer l'intégralité de notre processus de traitement des données.



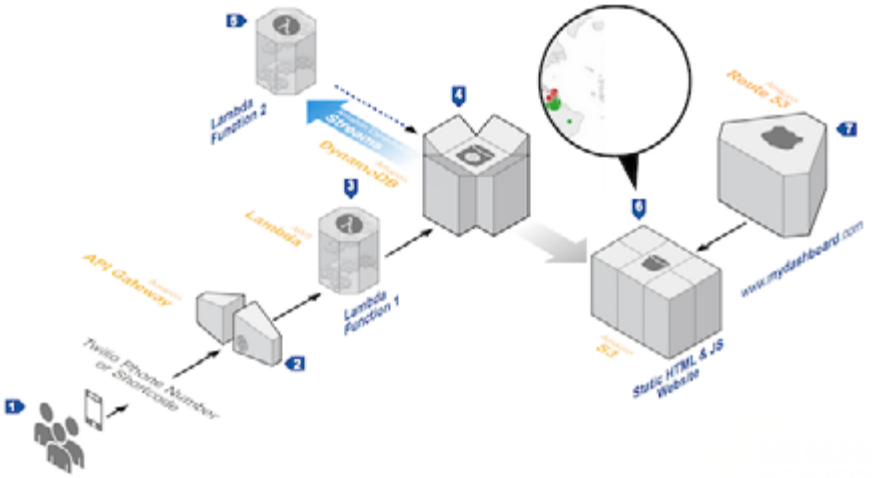
TRAITEMENT EN TEMPS RÉEL

- **Application Web**

En combinant du FaaS avec d'autres services managés, les développeurs peuvent créer de puissantes applications Web évoluant automatiquement dans une configuration hautement disponible sur plusieurs centres de données, sans aucun effort administratif requis pour l'évolutivité, la mise à l'échelle, les sauvegardes ou la redondance des données.

Cette architecture est un exemple concret d'une application Web de vote dynamique, qui reçoit les informations via SMS (au travers d'un service BaaS comme Twilio), agrège les totaux dans une base de données et utilise un gestionnaire de fichiers pour mettre à disposition les résultats en temps réel.

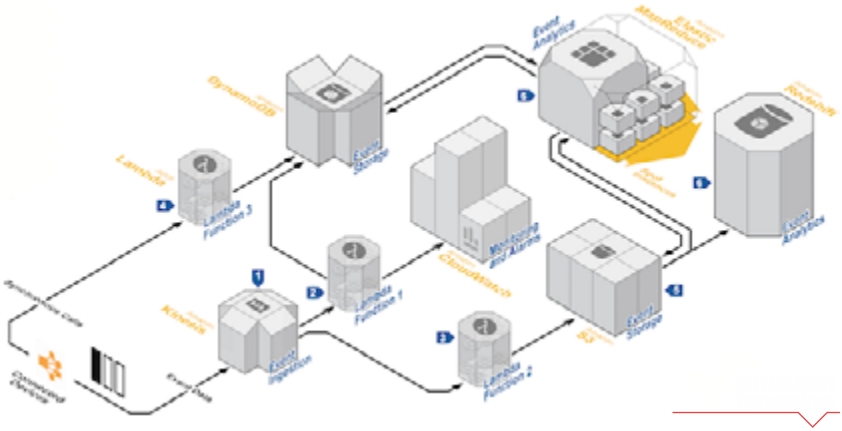
Cette architecture peut être générée rapidement au travers de technologies InfraAsCode (CloudFormation, Terraform).



TRAITEMENT EN TEMPS RÉEL

- **Backend IoT**

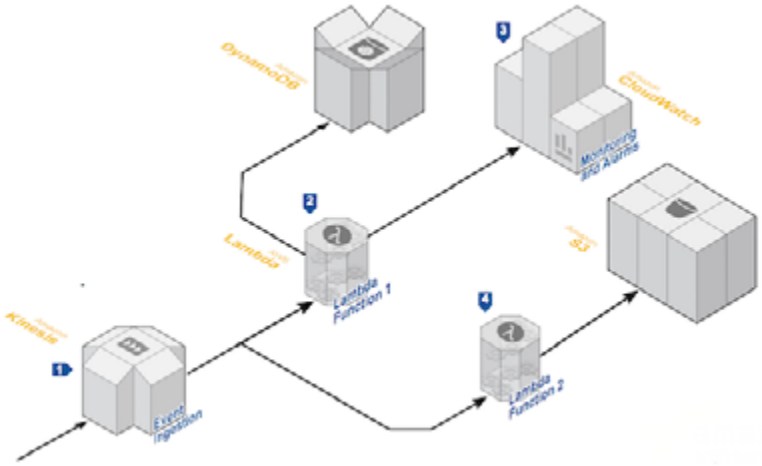
Cet exemple d'architecture de référence pour le domaine de l'IoT illustre la possibilité de récupérer et traiter les données issues de capteurs IoT.
En tirant parti de ces services, vous pouvez créer des applications rentables, capables de répondre et de s'adapter aux demandes massives générées par par les objets connectés.



BACKEND IOT

- **Traitement d'un flux de données temps réel**

Vous pouvez utiliser des Fonctions et des services de gestion de flux de données pour traiter de nombreux use cases comme le suivi des activités, l'analyse des clics sur une application Web ou même l'analyse des réseaux sociaux.



TRAITEMENT D'UN FLUX DE DONNÉES EN TEMPS RÉEL

5.3. Ce qui n'est pas «Serverless»

Jusqu'à présent, j'ai défini le terme «Serverless» pour signifier l'union de 2 principes - «Backend as a Service» et «Functions as a Service».

Avant de commencer à examiner la promesse que peut offrir ce nouveau type d'architecture, j'aimerais d'abord expliquer ce que n'est pas «Serverless». Actuellement lorsque l'on recherche ce terme, nous pouvons trouver de nombreuses définitions ou assimilations erronées.

- **PaaS (Platform as a Service)**

On peut en effet trouver des similitudes entre certains services PaaS comme Heroku qui propose des déploiements à la volée de notre application sur simple exécution d'une commande. Cependant ceux-ci ne procurent pas encore le niveau d'abstraction nécessaire pour être considérés comme «Serverless». Nous avons toujours besoin d'allouer une quantité de ressources pour garantir l'exécution de notre application ou notre système.

- **Conteneur**

La conteneurisation connait une popularité croissante de nos jours, surtout depuis l'arrivée de Docker. Nous pouvons en effet trouver certaines similarités entre FaaS et la conteneurisation. Mais rappelons le, FaaS offre une couche d'abstraction telle que nous n'avons plus la notion de processus système au contraire de Docker qui est basé sur la notion de processus unique.

Parmi ces similitudes, nous retrouvons l'argument de la mise à l'échelle. Fonctionnalité disponible niveau conteneur grâce aux systèmes tels que Kubernetes, Rancher ou Mesos. Dans ce cas nous pouvons nous poser la question du pourquoi faire du FaaS alors que nous pouvons faire du conteneur ?
Il faut savoir que malgré le buzz autour de cette technologie, elle reste toujours immature et de nombreuses entreprises ont encore du mal à basculer leur infrastructure de conteneurs en production. De plus les systèmes de mise

à l'échelle niveau conteneur est encore loin d'arriver au niveau de celle des FaaS même si cet écart tend à se réduire avec l'arrivée de nouvelles fonctions telles que Horizontal Pod Autoscaling pour Kubernetes.

Finalement, le choix de la technologie se fera selon les cas d'utilisation.

- **#NoOps**

Il ne faut pas confondre «Serverless» et NoOps. Si on prend le mot Ops (Opérations) cela ne signifie pas uniquement des opérations d'admin systèmes.
Cela signifie également au moins le suivi, le déploiement, la sécurité, la mise en réseau et aussi souvent une certaine quantité de débogage de production et de mise à l'échelle du système. Ces problèmes existent toujours avec des applications «Serverless», je dirais même qu'ils sont plus compliqués étant donné la jeunesse de la technologie et les nouvelles fonctions et paramètres à prendre en compte.

6 — LE SERVERLESS : LICORNE OU DÉSILLUSION

Les technologies «Serverless» sont souvent comparées au monde des licornes tant les promesses qu'elles offrent font rêver. Étudions cela de plus près et voyons ensemble les avantages et inconvénients de ces technologies.

6.1. Avantages

- **Coût opérationnel réduit**

«Serverless» est par nature une solution simple d'externalisation. Elle vous permet de payer quelqu'un pour gérer les serveurs, les bases de données et même la logique des applications. Etant donné que votre service fait partie d'un ensemble de services similaires, la notion d'économie d'échelle va alors s'appliquer - vous payez moins cher vos coûts de gestion vu que le même service est utilisé par de nombreux autres ce qui permet de réduire les coûts.

Les coûts réduits apparaissent comme le total de deux notions:
- le coût d'infrastructure
- le coût des employés (opérations / développement).

Alors que certains des gains de coûts peuvent venir uniquement de l'infrastructure de partage (matériel, réseau) avec d'autres utilisateurs, l'attente derrière peut aussi se traduire dans une réduction des coûts liés à l'utilisation du personnel d'exploitation du fait de l'utilisation de technologies managées.
Cet avantage, cependant, n'est pas trop différent de ce que vous obtiendrez en utilisant des technologies de type Infrastructure as a Service (IaaS) ou Platform as a Service (PaaS).

- **Coût de développement réduit**

Afin d'illustrer ce point, prenons en exemple le cas de l'authentification. De nombreuses applications codent leur propre service d'authentification et de gestion des utilisateurs, implémentant par la même occasion leur propre niveau de sécurité. Parmi les fonctionnalités implémentées, nous pouvons retrouver :

- L'enregistrement et la validation d'un utilisateur (Enregistrement Suppression)
- La récupération d'un mot de passe
- La connexion et l'accès aux services

Dans l'ensemble nous retrouvons à peu de choses près ces fonctionnalités dans la plupart des applications actuelles. Même si des solutions de type générateur d'application comme JHipster existent et permettent de générer rapidement plusieurs types d'authentification, il n'en reste pas moins à la charge de l'entreprise de maintenir ce code et de les faire évoluer. A ce jour, nous voyons l'émergence de services tels que Auth0 qui fournissent des fonctionnalités d'authentification "prête à l'emploi". L'application peut ainsi se décharger de ces fonctionnalités et laisser le fournisseur du service être responsable de leur maintien.

Un autre exemple qui se prête bien au jeu est l'utilisation de services de base de données tels que Firebase (**https://firebase.google.com/docs/database/**). On retrouve principalement ces cas d'utilisations au sein des architectures mobiles qui préfèrent créer une communication directe entre le client (mobile) et la base de données et ainsi supprimer tous les tiers, et par ce fait l'administration de la base de données et son optimisation. Ce système apporte aussi une nouvelle couche de sécurité et permet ainsi une gestion plus fine des données accessibles en fonction des différents profils utilisateur.

- **Mise à l'échelle automatique**

De mon point de vue l'un des avantages les plus importants du «Serverless» est la mise à l'échelle horizontale automatique, élastique et surtout gérée par le fournisseur. Cela peut se traduire par plusieurs avantages, principalement au niveau infrastructure, mais surtout cela permet d'avoir une facturation très fine et de ne payer que la charge dont vous avez besoin, que ce soit en temps de calcul utilisé (à partir de 100 ms pour AWS Lambda) ou en quantité de données récupérées ou analysées. Selon votre architecture et vos Use Cases cela peut engendrer une énorme économie pour vous.

Un des cas d'exemple d'économie est l'utilisation occasionnelle d'une fonction. Par exemple, disons que vous exécutez une application serveur qui ne traite que 1 demande chaque minute, qu'il faut 50 ms pour traiter chaque requête et que votre utilisation moyenne de CPU pendant une heure est de 0,1%. D'un point de vue charge de travail serveur, cela est extrêmement inefficace.

La technologie FaaS capte cette inefficacité et vous permet ainsi de ne payer que ce que vous consommez, c'est-à-dire 100ms (valeur minimum) de calcul par minute, soit moins de 0,5% du temps global.

- **L'optimisation, la clé de votre facture**

Même si cette nouvelle architecture propose de nouvelles fonctionnalités telles que la mise à l'échelle, elle n'en subit pas moins les contraintes inhérentes au développement d'applications. Ainsi la phase d'optimisation des fonctions prend encore plus de valeur vu qu'elle permettra, en plus d'améliorer le temps de réponse aux utilisateurs, d'économiser de l'argent sur la facturation. Par exemple pour une opération qui initialement prend 1 seconde et qui après optimisation prend 200ms, nous aurons une réduction immédiate de notre facture de 80% du coût de calculs.

- **Une informatique plus “verte”**

Etant donné que l’écologie est un sujet phare du moment, je lui dédie un paragraphe.

De nos jours, nous avons vu le nombre de datacenters augmenter de plus en plus au fil des années. La consommation en énergie de ceux-ci est énorme et de plus en plus de Cloud providers se sensibilisent à l’écologie et aux énergies renouvelables. Ainsi Google, Apple, … parlent de construire et d’héberger certains de leurs datacenters dans des zones à fort potentiel en énergies renouvelables afin de réduire l’impact sur l’environnement de ces sites.
L’une des causes de cette hausse du nombre de serveurs est due au maintien et à la consommation des serveurs dits “inactifs” face à une demande toujours croissante des entreprises.

Cela représente un fonctionnement extrêmement inefficace et surtout un impact environnemental non négligeable.

Ces charges non utilisées viennent principalement de décisions faites par les entreprises sur les capacités nécessaires au fonctionnement d’une application et des “marges de sécurité” faites afin de pallier les fluctuations de charges. Avec une approche «Serverless», nous ne prenons plus de décision sur la capacité nécessaire à l’exécution d’une fonctionnalité, cette charge revient désormais aux fournisseurs du service. Il se doit de fournir une capacité de calcul suffisante pour nos besoins en temps réel. Il pourra ainsi avoir une vision globale des capacités nécessaires pour l’ensemble de ses clients. Ils pourront par ce biais optimiser la gestion des ressources et permettre une réduction du nombre de serveurs et en conséquence l’impact environnemental des datacenters.

6.2. Les inconvénients

Et oui nous ne sommes pas là que pour vanter les mérites de cette nouvelle technologie et dire qu'elle peut résoudre tous nos problèmes (on compare d'ailleurs souvent celle-ci par un arc-en-ciel et une licorne). Comme toute technologie, elle vient aussi avec son lot d'inconvénients qui ne sont pas à prendre à la légère car ceux-ci pourraient devenir votre pire cauchemar selon vos Use Cases.

Il faudra toutefois séparer ces inconvénients en 2 types, ceux inhérents à ces nouvelles architectures et cette technologie et ceux qui ont pour origine sa jeunesse et son manque d'outillages et de solutions.

- **Verrouillage des Cloud Providers**

Le premier et pas des moindres pour moi est la dépendance forte que l'on crée avec le fournisseur de service. A ce jour aucune spécification n'est sortie afin d'adopter un langage commun entre les fournisseurs. Même si certains frameworks (Serverless.io) essaient de briser ces limitations, lors de la conception de votre solution et du choix des fonctionnalités, vous devrez faire le choix d'un fournisseur unique afin de garantir une certaine homogénéité de communication entre les différentes couches et pour pallier au vérouillage que les fournisseurs font de leurs services. Par exemple pour la liaison API Gateway > AWS Lambda, vous êtes contraint d'utiliser les 2 technologies AWS pour garantir son fonctionnement. Vous pourrez toujours trouver des passerelles afin d'utiliser plusieurs fournisseurs de services différents (Authentification via Auth0, DB via Firebase et API + Lambda via AWS) mais dans ce cas, cela compliquera fortement l'administration et la facturation de votre solution.

Il en va de même pour le code utilisé au sein des fonctions, il est propre à chaque fournisseur et il vous faudra donc prévoir un coût de réécriture lors de son changement.

- **Optimisations des serveurs**

A partir du moment où vous décidez d'utiliser des technologies «Serverless», vous abandonnez par ce fait le contrôle de certains systèmes tiers et de leur configuration. Même si cette gestion par le fournisseur vous conviendra dans 99% des cas, il reste 1% des cas où votre solution nécessitera d'avoir une configuration spécifique du service afin d'améliorer ses performances ou une meilleure qualité de services.

- **Sécurité**

Je ne pouvais pas écrire un livre blanc sur le «Serverless» sans parler de la sécurité de cette technologie. De nombreuses entreprises se sensibilisent de plus en plus aujourd'hui à la sécurité de leurs applications et à l'accès à leurs données. Les services «Serverless» ne vont pas échapper à la règle et vont apporter leur lot de questions. Je vais tenter d'en expliquer 3 mais de nombreuses autres sont à considérer.

1. En sécurité, on parle souvent de périmètre ou de surface d'action d'une solution. Cela correspond à l'empreinte que celle-ci a sur Internet. Plus l'empreinte est grande, plus la surface d'attaque est importante. Or l'utilisation de plusieurs services «Serverless» va augmenter votre empreinte et créer une certaine hétérogénéité dans vos politiques de sécurité. Par ce fait vous augmenterez votre probabilité d'intention malveillante à l'encontre de votre solution et la probabilité qu'une de ces attaques soit réussie.

2. Si vous utilisez le service de base de données de type BaaS et permettez l'accès direct à vos données via une API cliente, vous perdrez la barrière de protection qu'une application serveur traditionnelle peut fournir de par sa configuration réseau ou ses restrictions d'accès au serveur. Cependant, même si ce problème n'est pas une fin en soi, il est à prendre en compte lors de la conception de votre application.

3. Du fait de la mutualisation des services que vous utilisez, vous héritez des problématiques de sécurité inhérentes au services clients multiples. Par exemple l'accès aux données d'autres clients du fait du partage des processus.

- **Problème à l'utilisation**

Passons maintenant aux inconvénients inhérents aux solutions actuellement disponibles. Ceux-ci, pourront, en effet être corrigés avec l'évolution de la technologie et de l'écosystème qui l'entourent.

- Durée d'exécution

Un problème actuel concerne la limitation faite sur la durée d'exécution des fonctions. Actuellement nous avons une durée limite de 5min pour AWS et 9min pour Google Cloud. Cette contrainte restreint le périmètre d'actions des fonctions et ainsi empêche leur utilisation pour un grand nombre de Use Cases comme le traitement vidéo par exemple.

- Latence de démarrage

Un autre inconvénient que certaines implémentations de FaaS provoquent est la latence au démarrage. En plus du temps d'exécution de la fonction, vient s'ajouter une latence pouvant aller jusqu'à 10 secondes selon les cas de figure (exemple lors de l'utilisation d'une JVM ou lors d'un premier lancement). C'est pourquoi certains fournisseurs comme Google contraignent le langage de développement de leurs fonctions et ainsi permettent uniquement l'utilisation du JavaScript (ce qui se comprend vu les performances de leur moteur).

De même nous pouvons observer des latences réseaux lorsque l'on met en série plusieurs fonctions lambda.

- Tests

Vu que l'on parle beaucoup de développement il en est de même pour les tests. Même si certains pensent que "Tester c'est douter", nous nous devons de traiter ce point, d'autant plus qu'il s'agit d'un lourd inconvénient lors de l'utilisation de ce type de services.

Certains peuvent penser qu'en raison de l'isolation de chacune des fonctions, il peut être relativement facile de les tester. Ceux-là ont raison pour le périmètre des tests unitaires vu qu'il s'agit simplement d'un bout de code, cependant lorsque l'on aborde le sujet des tests d'intégration c'est une autre paire de manches. De nouvelles notions et questions vont alors apparaître venant principalement du fait que vous dépendez de services externes (base de données, authentification). Nous pouvons nous interroger sur leur périmètre et sur la pertinence d'effectuer ces tests bout en bout. Si tel est le cas, est-ce que ces services sont compatibles avec vos scénarios de tests comme la gestion des états avant et après ? De plus, est-ce qu'une grille de coûts spécifique est prévue par les fournisseurs lors de tests de charge par exemple ?

Si votre volonté est, au contraire, de vous soustraire à ces services temporairement, il vous faudra alors des systèmes de stub local qui ne sont pas forcément fournis par le fournisseur. Sur ce point, Google se différencie des autres par le fait que ses solutions sont généralement basées sur des systèmes Open Source, de ce fait l'écosystème qui gravite autour fournit assez rapidement des moyens de stub ces services. D'autres questions apparaissent alors sur le niveau de confiance que l'on peut avoir dans ces stubs et s'ils ne sont pas fournis, comment faire pour les implémenter.

Il en va de même pour l'intégration des services FaaS. Il est encore difficile de trouver une implémentation locale de la structure qui embarque les fonctions. Il va donc falloir utiliser directement l'environnement final. Même si des notions de staging permettent de séparer l'utilisation en test de l'utilisation en production, celles-ci ne s'appliquent pas à tous les services.

- Déploiement et versionning

Actuellement aucun pattern probant n'est sorti sur la phase de packaging et déploiement. C'est pourquoi nous avons rapidement des contraintes sur le déploiement atomique des fonctions. Prenons le cas d'une série de fonctions qui s'exécutent, afin de garantir un déploiement uniforme, il va falloir arrêter votre service à l'origine des événements de déclenchement, puis livrer l'ensemble de vos fonctions pour ensuite activer de nouveau le service. Cela peut représenter un problème important pour les applications nécessitant de la haute disponibilité. Il en va de même pour le versionning "applicatif" et la phase de rollback.

- Supervision

A ce jour les seuls solutions possibles pour effectuer la supervision et le débogage de vos fonctions sont celles fournies par votre fournisseur et il faut se le dire, elles ne sont pas encore au niveau attendu. Même si des efforts énormes sont faits comme avec la sortie en preview de AWS X-Ray (***https://aws.amazon.com/fr/xray/?tag=viglink122648-20***), il reste encore du chemin à parcourir afin d'avoir une solution complète et spécifique FaaS.

4 — LE FAAS DANS LE DÉTAIL

Précédemment nous avons vu ce qu'était le FaaS dans les grandes lignes. Mais creusons un peu plus cette notion. Pour ce faire, je me suis inspiré du découpage de Mike Roberts (**https://martinfowler.com/articles/serverless.html**) qui me semble très clair et qui traite correctement chacunes des couches du FaaS. Mais d'abord reprenons la définition AWS du produit "Lambda" puis étudions chaque concept un par un.

AWS Lambda vous permet d'exécuter du code sans avoir à mettre en service ou gérer des serveurs **(1)**. Vous payez uniquement pour le temps de calcul consommé, il n'y a aucun frais lorsque votre code n'est pas exécuté. Avec Lambda, vous pouvez exécuter du code pour pratiquement n'importe quel type d'application ou service dorsal (back-end) **(2)**, sans aucune tâche administrative. Il vous suffit de charger votre code : Lambda fait le nécessaire pour l'exécuter **(3)** et le dimensionner **(4)** en assurant une haute disponibilité. Vous pouvez configurer votre code de sorte qu'il se déclenche automatiquement depuis d'autres services AWS **(5)**, ou l'appeler directement à partir de n'importe quelle application Web ou mobile **(6)**.

1. **Fondamentalement, FaaS concerne l'exécution de code back-end sans gérer de systèmes de serveurs ou de serveurs applicatifs.** Cette seconde clause "serveurs applicatifs" est la différence clé lorsque l'on compare cette technologie avec d'autres tendances architecturales modernes comme le PaaS ou la conteneurisation.

2. **L'offre FaaS ne nécessite pas de coder au travers d'un framework ou d'une librairie spécifique.** L'unique dépendance d'environnement que l'on peut avoir est liée aux possibilités offertes par le service selon le fournisseur. Par exemple, à ce jour les fonctions AWS Lambda peuvent être implémentées en JavaScript, Python ou tout autre langage issu de

la JVM (Java, Clojure, Scala, …) alors que Google Cloud Platform propose uniquement le JavaScript (dû principalement à la très haute performance de son moteur JavaScript). Cet environnement d'exécution n'est cependant pas fixe en raison de notre capacité à exécuter d'autres processus. De ce fait, il nous est possible d'utiliser tous les langages sous condition qu'ils puissent être compilés par un processus Unix. Nous pouvons toutefois retrouver des restrictions d'architectures principalement lorsque l'on parle d'état ou de durée d'exécution, mais nous reviendrons plus tard sur ces points.

3. **À partir du moment où nous n'avons pas de serveurs applicatifs pour héberger le code, l'exécution est de ce fait très différente des systèmes traditionnels.** Le processus de mise à jour commence par l'upload du code au fournisseur de service puis l'appel d'une API de fournisseur pour signifier la mise à jour du code.

4. **Le scaling horizontal (augmentation du nombre d'instances) est complètement automatique, élastique et géré par le fournisseur de services.** Si votre système a besoin de répondre à 100 requêtes en parallèle, le fournisseur du service s'en chargera sans aucune intervention de votre part sur la configuration.

5. **Rappelons que le FaaS est un service basé sur l'event-driven.** De ce fait, les fonctions sont déclenchées par des événements. Nous pouvons avoir différents types d'événement comme ceux issus des services du fournisseur. Parmi ceux-ci nous avons des événements lors de la mise à jour d'un fichier sur un bucket, des événements planifiés (cron) ou même la réception de messages via un système de queuing. Votre fonction devra par la suite fournir des paramètres spécifiques à la source de l'événement à laquelle elle est liée.

6. **Il vous sera aussi possible de déclencher des fonctions en réponse à des requêtes HTTP entrantes,** généralement au travers d'une passerelle API (API Gateway, WebTask, Cloud endpoints).

- **Cycle de vie**

Les fonctions FaaS ont des restrictions importantes en ce qui concerne leurs cycles de vie. La principale est son mode “Stateless”, aucune donnée ne sera disponible lors d’une invocation ultérieure, cela inclut les données en mémoire ou celles que vous auriez pu écrire en local sur le disque. Cela peut avoir un énorme impact sur votre architecture applicative mais rappelons-le, cette spécification est aussi présente dans “The twelve-factor App” (***https://12factor.net/fr/***).

- **Durée d’exécution**

Les fonctions FaaS sont limitées en durée d’exécution. Par exemple les AWS Lambda ne sont pas autorisées à s’exécuter plus de 5 minutes et si elles le font, elles se termineront automatiquement à la fin de cette période.

Cela remet l’accent sur le fait que les fonctions FaaS ne sont pas appropriées à tous les use-cases ou du moins devront être adaptées pour pallier ces contraintes. Par exemple, dans une application traditionnelle vous pouvez avoir un seul service qui exécute une longue tâche alors qu’en FaaS, il vous faudra sûrement le séparer en différentes fonctions indépendantes les unes des autres mais possiblement séquencées.

- **Latence de démarrage**

Le temps de réponse de votre fonction FaaS à une demande dépend d’un grand nombre de facteurs et peut aller de 10 ms à 2 minutes. Soyons un peu plus précis, en utilisant AWS Lambda comme exemple.

Si votre fonction est implémentée en JavaScript ou Python qui sont des langages interprétés avec un contenu simple (moins d’un millier de lignes de code), la durée de chargement (warmup) devrait se situer entre 10 et 100 ms. Des fonctions plus importantes peuvent occasionnellement voir ce temps augmenter.

Si votre fonction Lambda est exécutée dans une JVM (Java, Scala, Kotlin, …), vous pourrez avoir un temps de démarrage drastiquement plus long (> 10 secondes) rien que pour le chargement de la JVM. Cependant, ceci ne se produit que pour l'un ou l'autre des scénarios suivants :

- Votre fonction traite rarement des événements (plus de 10 minutes entre les invocations)
- Vous avez des pics très soudains dans le trafic, par exemple : vous traitez 10 requêtes par seconde, mais cela accélère jusqu'à 100 requêtes par seconde en moins de 10 secondes.

Il est possible d'éviter la première situation via le maintien opérationnel de votre fonction au travers d'un "ping" toutes les N secondes.

Ces problèmes sont-ils préoccupants ?

Cela dépend du style et de la forme du trafic de votre application et du maintien en activité de vos fonctions. Cela dit, si vous étiez en train d'écrire une application à faible latence, vous ne voudriez probablement pas utiliser les systèmes FaaS à ce moment-là, quel que soit le langage que vous utilisez pour la mise en œuvre.

Si vous pensez que votre application peut avoir des problèmes comme celui-ci, vous devriez tester avec une charge représentative de la production pour effectuer un bench des performances. Si votre cas d'utilisation ne fonctionne pas maintenant, gardez à l'esprit qu'il s'agit d'un domaine majeur de développement par les fournisseurs de FaaS.

- **La Gateway API**

Jusqu'à maintenant nous avons abordé de nombreuses notions et services liés au FaaS. Mais je tiens particulièrement à mettre l'accent sur l'un d'eux qui est la Gateway API. Une Gateway API est un serveur HTTP où les contrats

/ endpoints sont définis au travers d'une configuration. Ceux-ci vont alors générer un événement qui pourra être utilisé au travers d'une fonction FaaS. En règle générale, la Gateway API permet le mapping des paramètres d'une requête HTTP aux arguments d'entrée d'une fonction FaaS puis elle transformera le résultat de l'appel à la fonction en une réponse HTTP et la renverra à l'appelant d'origine. Au-delà des demandes de routage simple, ce service permet aussi d'avoir des notions d'authentification ou même de validation des paramètres.

L'utilisation combinée des API Gateway + FaaS peut donner lieu à la création de microservices http-frontend de type «Serverless» avec tous ses avantages comme par exemple le scaling automatique.

À l'heure actuelle, l'outillage pour les passerelles API est immature et comporte de nombreuses lacunes lorsque l'on veut y appliquer des processus de développement (versioning, ...).

- **L'outillage**

Ce manque de maturité des outils liés aux services d'API Gateway s'applique malheureusement aussi aux services FaaS. Cependant il existe des exceptions : un exemple est Auth0 Webtask (***https://webtask.io***) qui accorde une priorité significative au développement d'interfaces orientées utilisateurs.

Parmi ces lacunes, nous retrouvons le manque de fonctionnalités de débogage, de versioning ou de logging, même si celles-ci commencent à être comblées petit à petit (exemple X-Ray (***https://aws.amazon.com/fr/xray/***) chez AWS).

7 — FAIRE DU SERVERLESS

Après avoir expliqué le **Pourquoi** et le **Quand** passons maintenant au Comment.

Ne nous voilons pas la face, aujourd'hui le FaaS a du mal à être adopté au sein des sociétés, seuls certains use cases sont à ce jour applicables. La principale raison vient du fait qu'il est encore difficile de simuler en interne toutes les briques BaaS et FaaS permettant de développer et tester sa solution avant une mise en production.

Cependant de nombreuses solutions sont en cours de développement à la fois par la communauté Open source et les Cloud Providers afin d'offrir le niveau de services suffisant pour pallier ce problème.

Je ne pourrai pas vous détailler tous les services disponibles de type «Serverless» dans cet article mais je vous invite à vous rendre sur la page suivante pour avoir un listing exhaustif :
https://github.com/anaibol/awesome-»Serverless»

7.1. BaaS et nos amis les Cloud Providers

Il n'y a pas de FaaS sans BaaS. Les principaux fournisseurs de ces services managés "clé en main" sont bien évidemment nos amis les Cloud Providers. Nous pouvons lister 3 acteurs majeurs aujourd'hui. Nous retrouvons bien sûr les géants : Amazon Web Services, Google Cloud Platform et Microsoft Azure mais des solutions comme Auth0 et Iron commencent à faire parler d'elles.

Vous trouverez ci-contre un tableau récapitulatif des principales solutions 100% managées disponibles dans le Cloud.

Amazon Web Service	Google Cloud Platform	Microsoft Azure Services
Compute		
AWS Lambda	Google Cloud Functions	Azure Functions
Storage		
Amazon Glacier and Amazon S3 Standard - Infrequent Access	Google Cloud Storage Nearline	Azure Cool Block Storage
Amazon S3	Google Cloud Storage Standard	Azure Block Storage
Amazon EC2 Container Registry	Google Container Registry	Azure Container Registry
Database		
Amazon DynamoDB	Google Cloud Datastore or Google Cloud Bigtable	Azure DocumentDB
Big Data		
AWS Data Pipeline	Google Cloud Dataflow and Google Cloud Dataproc	Azure HD Insight
Amazon Kinesis and Amazon Simple Queue Service (SQS)	Google Cloud Pub/Sub	Azure Event Hubs and Azure Service Bus
Amazon Redshift	Google BigQuery	Azure SQL Data Warehouse and Azure Data Lake Analytics
Monitoring		
Amazon CloudWatch	Google Cloud Monitoring and Google Cloud Logging	Azure Application Insights and Azure Operational Insights

7.2. Et le FaaS dans tout ça !

L'intégration d'une fonction est relativement simple. Il s'agit généralement d'un bout de code dans un langage compatible avec le fournisseur du service (généralement NodeJS, Java, C#, Go et Python) qui est compilé, zippé et déployé. La partie complexe se trouve dans la configuration et l'interaction des fonctions avec les autres services (API Gateway, DB, Stockage, …).

C'est pourquoi nous avons aujourd'hui deux approches qui sont en train d'émerger au sein des frameworks Open Source.

La première plus orientée infrastructure vous permettra de gérer relativement facilement l'intégration de vos fonctions et de toutes les ressources liées à elles. Ces frameworks se basent généralement sur les API fournies par les Cloud Providers ou les services de type DevOps (exemple de CloudFormation chez AWS).
La seconde, plus orientée API et ressources Web correspond le plus à la vision actuelle des développeurs de solution avec une abstraction simple de la communication entre une API Gateway et une fonction.

- **Vision infrastructure**

A ce jour la tendance de développement de la communauté Open Source se dirige plus vers une vision InfraAsCode. Ils permettent d'une façon relativement simple, via des CLI et des fichiers de configuration, d'interagir avec les différents services Cloud et de gérer l'intégralité de la stack technique liée à une fonction.

L'un des acteurs les plus connus à ce jour est le framework Serverless (***https://serverless.com***). Il est compatible avec les solutions Cloud suivantes : AWS, IBM OpenWhisk, Microsoft Azure, GCP, Kubeless, Spotinst, Webtask.

Afin de vous donner une première vision de ce que vous pouvez faire, voici un petit Getting Started que vous pouvez faire assez rapidement et voir la

puissance que ces fonctions «Serverless» peuvent avoir.

Pré-requis : AWS Cli (***http://docs.aws.amazon.com/fr_fr/cli/latest/userguide/installing.html***) avec un compte AWS fonctionnel

```
$ mkdir my-first-function && cd my-first-function
$ «Serverless» create --template aws-nodejs
```

Modifiez ensuite le fichier Serverless.yml comme suit :

```
service: aws-nodejs
provider:
  name: aws
  runtime: nodejs6.10
  region: eu-central-1

functions:
  hello:
    handler: handler.hello
    events:
      - http:
          path: users
          method: get
```

- **Vision API**

L'un des exemples d'implémentation de cette vision est le framework Chalice (***https://github.com/aws/chalice***). Pour le petit historique, ce framework a été initialement développé par AWS afin de permettre de concilier et simplifier l'implémentation pour les développeurs. Il est à ce jour à la version 1.0.3 et évolue très rapidement.

Il vous permettra d'avoir le niveau d'abstraction nécessaire pour qu'un développeur sans réelle connaissance du monde du cloud ou des services associés puisse en 5min mettre en place une fonction accessible via un contrat REST.

Au même titre que l'article sur le framework «Serverless», voici un aperçu rapide de celui-ci :

Requirements : AWS Cli (***http://docs.aws.amazon.com/fr_fr/cli/latest/userguide/installing.html***) avec un compte AWS fonctionnel et bien sûr le package Chalice

```
$ chalice new-project s3test && cd s3test
```

Remplacer le contenu du fichier app.py par celui-ci. Il vous permettra de créer un contrat REST '/objects/{key}' accessible via GET et PUT afin de pouvoir lire ou déposer un contenu sur un bucket S3.

```
import json
import boto3
from botocore.exceptions import ClientError

from chalice import Chalice
from chalice import NotFoundError

app = Chalice(app_name='s3test')

S3 = boto3.client('s3', region_name='eu-west-1')
BUCKET = 'ippevent-public'

@app.route('/objects/{key}', methods=['GET', 'PUT'])
def s3objects(key):
    request = app.current_request
    if request.method == 'PUT':
        S3.put_object(Bucket=BUCKET, Key=key,
                      Body=json.dumps(request.json_body))
    elif request.method == 'GET':
        try:
            response = S3.get_object(Bucket=BUCKET, Key=key)
            return json.loads(response['Body'].read())
        except ClientError as e:
            raise NotFoundError(key)
```

Puis exécuter les commandes suivantes :

```
$ chalice local
$ http put  http://localhost:8000/objects/document.json value1=123
$ http get http://localhost:8000/objects/document.json
$ chalice deploy
$ chalice url
```

En l'espace de quelques lignes il vous est possible d'accomplir une tâche qui en prendrait beaucoup plus si vous deviez gérer le démarrage ou la gestion d'un serveur.

8 — POUR CONCLURE

La mise à l'échelle automatique, le pay-as-you-go, la gestion automatique de vos services sont autant de concepts clés qui caractérisent le «Serverless». L'utilisation de ce type de services doit être soumis à réflexion afin de comprendre et connaître au mieux vos cas aux limites et de valider son applicabilité. Cette architecture peut ne pas convenir à certaines typologies pour plusieurs raisons comme le besoin de synchronisme dans vos communications. Cependant elle peut, par essence, vous fournir des garanties de haute disponibilité, résilience de vos données et une mise à l'échelle automatique de vos services sans effort supplémentaire de votre part.

Rappelons-le, il ne s'agit pas d'une solution pouvant convenir à tous les cas d'utilisations. De nombreuses contraintes sur le FaaS, liées à sa nature de communication basée sur l'événementiel, peuvent rapidement devenir bloquantes et ajouter de la complexité dans le séquencement de vos échanges. En revanche ces services pourront rapidement vous apporter des bénéfices tant financiers qu'en terme d'évolutivité et d'adaptabilité.

Le dernier point possiblement bloquant à l'heure actuelle est le manque de maturité de certains types de services. Maturité autant en terme d'outillage qu'en terme de process. Certaines briques technologiques sont encore récentes (3 ans pour AWS Lambda). Il vous faudra donc bien prendre en compte la dette technique générée par ce type de solutions.

Mais que serait une nouvelle technologie sans son lot de contraintes, surtout quand elle offre autant de promesses. De nombreuses entreprises commencent d'ores et déjà à l'utiliser et nous commençons à avoir des premiers REX lors de conférences.

Enfin, ce type de services est amené à évoluer rapidement et le catalogue des

produits 100% managés à croître. Chacune des parties (Cloud Providers et utilisateurs) sont gagnantes dans l'utilisation de tels services et garantissent ainsi un avenir florissant qui apportera de l'homogénéité et de la sécurité dans le monde de l'infrastructure et de l'hébergement informatique.

IPPON
Discovery to Delivery

PARIS - BORDEAUX - NANTES - LYON

RICHMOND, VA - WASHINGTON, DC - NEW YORK

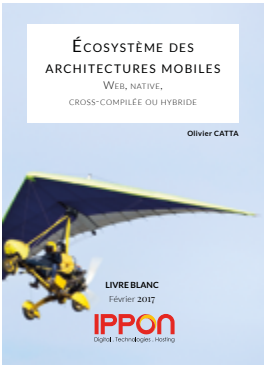
MELBOURNE

MARRAKECH

FRANCE :
www.ippon.fr/contact
contact@ippon.fr
blog.ippon.fr
+33 1 46 12 48 48
@ippontech

US :
www.ippon.tech/contact
blog.ippon.tech
844-477-6687
@ipponusa

Découvrez aussi nos
autres publications
sur http://www.ippon.fr/ressources/



IPPON
Discovery to Delivery

www.ippon.fr
blog.ippon.fr

Paris
Nantes
Bordeaux
Lyon
Washington, DC
Richmond, VA
New York, NY
Melbourne
Marrakech