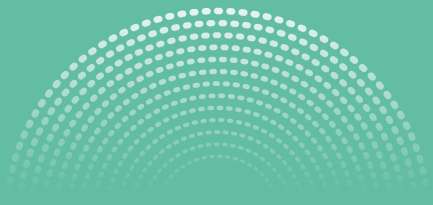
SOLAT

Web API

Web Application Programming Interface

LIVRE BLANC



SOAT en quelques mots

Fondé par Michel Azria et David-Eric Levy, SOAT est un cabinet de conseil IT qui fait grandir depuis 16 ans une communauté de consultants spécialisés dans le développement et l'optimisation de systèmes d'information.

Au quotidien, ce sont plus de 370 collaborateurs de talent qui accompagnent nos clients sur des problématiques technologiques et organisationnelles complexes : développement d'applications spécifiques, performance et qualité du code, déploiement continu, choix technologiques, conception d'architectures IT, transformation agile et modernisation du système d'information.

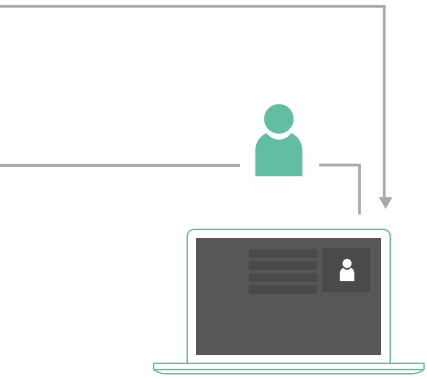
Pour garantir à nos clients des prestations de haute qualité et une expertise toujours à la pointe, nous avons placé l'innovation technologique et la capitalisation de nos savoir-faire au cœur de notre stratégie.

Conférences, livres blancs, avis d'expert, blog, communautés techniques et de pratiques… Nous favorisons le partage des connaissances pour faire continuellement progresser nos consultants et apporter toujours plus de valeur ajoutée à nos clients. Cette démarche de partage et de capitalisation ancrée dans les gènes de SOAT participe à la diffusion des savoirs.

Allier savoir-faire et savoir-être reste notre défi au quotidien pour rendre notre organisation toujours plus performante et épanouissante.

Table des matières

introduction



"Les APIs ouvrent le potentiel d'innovation stocké dans les données des entreprises"

Le concept d'interface de programmation (API - Application Programming Interface) existe depuis plus de trente ans dans l'univers de l'ingénierie logicielle mais a évolué avec l'avènement des systèmes distribués, du protocole HTTP et du web qui ont accéléré le partage de l'information et des données. Cet essor, qui a eu lieu au milieu des années 2000 avec l'arrivée du web 2.0 (ou web social) et des entreprises telles que Facebook, Flickr, Twitter, Delicious, n'a depuis, cessé de s'amplifier.

C'est au travers des Web APIs qu'a eu lieu cette interconnexion et ces échanges massifs de données entre les applications et les systèmes d'information, devenant par la suite une composante essentielle du cloud computing.

Au-delà de ses impacts technologiques, une évolution s'est aussi opérée dans l'approche économique du développement des systèmes d'information. "API centric", "API first" sont autant d'approches qui agissent comme des paradigmes de développement. Elles modélisent et structurent l'architecture des applications ainsi que l'interconnexion entre celles-ci.

La modernisation des systèmes d'information est devenue un enjeu majeur dans les entreprises. Une majorité d'entre elles souhaitent désormais exposer leurs fonctionnalités et leurs données, pour créer de nouveaux usages. Les Web APIs se retrouvent ainsi dans des domaines aussi larges que la finance, le marketing, la logistique ou les technologies de l'information, répondant toutes au même besoin : l'interconnexion des systèmes d'information dans les écosystèmes de données. Les APIs offrent en effet la possibilité à toutes ces entités de mieux travailler entre elles et à tous les niveaux ; que ce soit avec les partenaires ou le monde entier.

Les APIs s'intègrent donc naturellement dans une économie de partage, d'exposition et de monétisation des données. Cette économie des APIs (API economy) représente la valeur que propose le cycle de vie d'une API ainsi que le Business Model associé. Les entreprises, via l'exposition de leurs APIs, vont pouvoir intégrer des écosystèmes pour atteindre de nouvelles audiences et accéder à de nouvelles places de marché. La mise en place d'une API peut aussi permettre d'augmenter un retour sur investissement d'actifs logiciels déjà existants.

Enfin, les APIs ouvrent le potentiel d'innovation stocké dans les données des entreprises, que ce soit au sein d'un système d'information et des équipes internes d'une organisation ou vers des développeurs et sociétés tierces.

Ce livre a été écrit pour toutes les personnes concernées par cette révolution en cours des Web APIs, que vous soyez développeur, leader technique, manager technique ou de produit. Il aborde différentes approches et techniques de conception, et fournit un ensemble de bonnes pratiques à respecter pour une adoption facilitée des APIs par des clients. Vous trouverez ainsi dans cet ouvrage l'ensemble des sujets qu'il est nécessaire d'examiner lorsque l'on souhaite mettre en oeuvre la connexion des applications au travers des APIs, à savoir : leur conception, leur design, leur sécurisation, leur documentation ainsi que leurs possibilités de monétisation.

PARTIE I :

Les Web APIs

Les APIs sont « *un ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels* »
(Source : Wikipedia).

Le principe premier d'une API est donc de fournir une interface réutilisable sur laquelle plusieurs applications peuvent interagir facilement. Les APIs n'offrent pas d'interface graphique, ce sont les développeurs d'applications qui utilisent les APIs dans le cadre de leur réalisation.

Les APIs ne sont pas récentes dans le monde du logiciel et de l'IT mais elles ont évolué au cours des dernières années et plus particulièrement dans le domaine du web.

Une Web API est, comme son nom l'indique, une API accessible sur le web via principalement le protocole HTTP. Elle représente un concept et non une technologie en soi. Il est tout à fait possible de créer des Web APIs en utilisant des technologies et langages aussi différents que Java, C#, Python, Ruby, Go...

Les APIs n'offrent pas seulement un mécanisme simple d'intégration d'applications, mais permettent d'exposer des fonctionnalités et des données existantes d'une application pour créer de nouveaux usages à partir de celles-ci.

Les Web APIs au cœur du système d'information

Devenir toujours plus agile et plus réactif est aujourd'hui devenu un mantra dans les directions des systèmes d'information. L'adaptation aux besoins des clients, aux exigences réglementaires et à la concurrence représente un ensemble de défis qu'il est difficile d'adresser tout en gardant une capacité à délivrer de la valeur de manière continue.

La mise en place d'un système d'information basé sur des APIs a pour but de casser les silos liés aux organisations pyramidales et de rendre l'information accessible aux différents départements et entités d'une entreprise. Elle permet, en outre, de mettre en valeur les actifs logiciels les plus importants au sein d'une organisation.

Cette stratégie permet également un découplage des applications ainsi qu'une séparation des responsabilités.

Les GAFA (Google, Apple, Facebook et Amazon) l'ont compris depuis longtemps, allant jusqu'à imposer à leurs équipes de développer et déployer une API pour chaque application de l'entreprise. On peut se rappeler de la lettre envoyée par Jeff Bezos (CEO d'Amazon) à ses employés en 2002 :

"*Toutes les équipes exposeront dorénavant leurs données et fonctionnalités par l'intermédiaire d'API. Les équipes ont l'obligation de communiquer les unes avec les autres à travers ces APIs. Aucun autre mode de communication entre équipes ne sera autorisé. Pas de liens directs, pas d'accès direct en lecture à un entrepôt de données d'une autre équipe, pas de modèle de partage de mémoire [...] Cela signifie que chaque équipe doit planifier et concevoir de façon à pouvoir exposer son API à des développeurs du monde extérieur. Aucune exception ne sera tolérée. Quiconque ne suivra pas cette consigne sera renvoyé.*

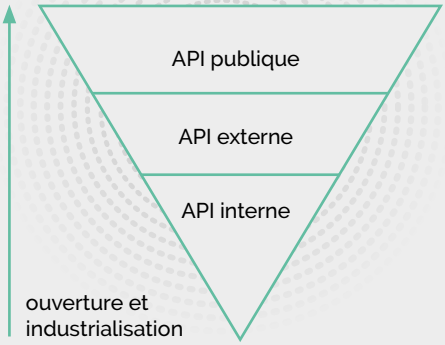
Merci, bonne journée !"

Par cette transformation imposée par son CEO, Amazon a franchi une étape en passant d'un vendeur de livre en ligne à une entreprise mettant à disposition des services d'infrastructure.

Les Web APIs peuvent et doivent agir comme des acteurs du changement au sein de votre système d'information et permettent d'accompagner la transformation digitale de votre entreprise afin de rester plus compétitif.

Niveaux d'industrialisation et ouverture

Il existe plusieurs niveaux d'industrialisation relatifs à la stratégie définie par l'entreprise pour valoriser ses APIs et pour lesquels un niveau de maturité (tant au niveau technique que des compétences de vos équipes) est nécessaire.



Au sein du premier niveau, l'usage des APIs reste interne à l'organisation. Elles permettent de stimuler l'innovation et de rendre les équipes plus agiles. Déployer des APIs en interne est un très bon point de départ pour une entreprise avant de les proposer à l'externe (aux partenaires privilégiés ou au public) et d'intégrer un écosystème dans lequel elle devra rester compétitive.

Le second niveau d'industrialisation comprend un usage interne des APIs mais s'étend également aux développeurs ou aux entreprises partenaires. Elles permettent de générer de nouvelles sources de revenu, les APIs étant alors des produits proposés par l'entreprise.

La troisième et dernière phase d'industrialisation permet, au-delà d'un usage plus restreint entre l'interne et les partenaires, de rendre accessible les APIs au plus grand nombre. Elles permettent d'accélérer l'innovation externe (open innovation) et d'intégrer de nouveaux écosystèmes.

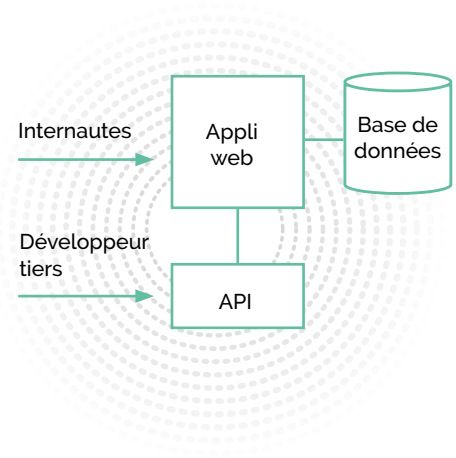
Plus le niveau est élevé, plus l'API est considérée comme ouverte (on utilise alors le terme d'Open API). Pour répondre aux contraintes de cette ouverture, il est nécessaire d'être accompagné par des équipes possédant des compétences techniques, fonctionnelles et de monétisation.

Approche de conception

La façon de concevoir une API ou d'aborder un projet de développement d'une API a changé. Autrefois vu comme des projets connexes, elles sont désormais au centre des stratégies dans le développement d'applications et des systèmes distribuées.

L'approche de conception historique

Historiquement, beaucoup d'entreprises commençaient par créer leur application web puis, pour des besoins liés à leur future application mobile, développaient une API permettant d'exposer les données nécessaires à cette application. Les APIs étaient souvent vues comme des projets tiers et secondaires, développés de manière séparée. Les fonctionnalités développées pour l'application web n'étaient pas immédiatement répercutées sur l'API. En résultait un système peu modulaire et peu agile non adapté à l'évolution des besoins clients.



L'approche de conception API First

Pour pallier ces problématiques de couplage entre l'application web et l'API liées à l'approche historique, la stratégie API First consiste à exposer au plus tôt les données et les fonctionnalités d'une application à travers une ou plusieurs APIs.

Vous développez ensuite vos applications web et mobiles sur ces APIs. Les développeurs tiers consommeront de la même manière cette API pour utiliser les données et construire de nouvelles applications.

Ce type de stratégie correspond à une pratique de développement couramment appelée "dogfooding" (aussi appelée "eating your own dog food") qui désigne l'utilisation de ses propres produits et services pour être directement confronté à ses qualités et ses défauts. Cette méthode atteste de la confiance d'une entreprise en ses propres produits.

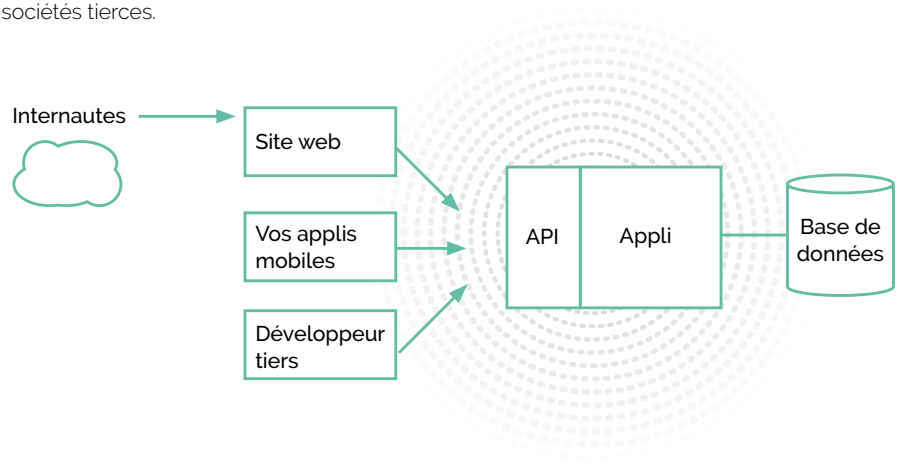
Dans ce cas précis, vos APIs seront utilisées par vos propres équipes ainsi que par des sociétés tierces.

Une séparation claire des responsabilités

Les APIs créent une séparation entre le fournisseur du service et les consommateurs de ce service. Elles permettent de distinguer les responsabilités entre la production des données, la mise à disposition des fonctionnalités d'un côté et de l'autre la consommation de ces APIs.

La première responsabilité d'une API est de collecter les données nécessaires pour satisfaire une demande cliente (requête), que ce soit auprès de serveurs applicatifs, d'un système distribué ou de solutions de stockage. L'API est ensuite chargée de mettre en forme les données pour le besoin des consommateurs. Cela peut être un format JSON, XML, etc.

Afin de garantir une consommation stable de l'API par les clients, celle-ci doit posséder dès sa conception une capacité à évoluer et à prendre en compte les impacts possibles pour le client.



Les propriétés d'une API

Les APIs sont dotées de nombreuses responsabilités comme la collecte, la structuration, la livraison et la sécurisation de données en provenance des systèmes d'information. Ces propriétés ne sont pour autant pas suffisantes pour concevoir des APIs qui donnent envie d'être consommées par les clients.

Les APIs doivent également répondre à un certain nombre d'exigences et se doter de propriétés additionnelles pour être véritablement attractives. Une API doit donc être avant tout :

- **Centrée sur le client**
  La conception d'une API doit être en priorité dirigée par l'usage que vont en faire les développeurs.
- **Simple**
  L'API doit être facilement et rapidement exploitable à sa juste valeur par les développeurs.
- **Intuitive et Prédictible**
  Les développeurs doivent comprendre par eux-mêmes les URIs (Uniform Resource Identifier), les paramètres et les données des objets.
- **Explorable**
  Les développeurs doivent être capables d'explorer les APIs sans nécessairement avoir à lire la documentation et doivent pouvoir se reposer sur les conventions de code actuelles.
- **Correctement documentée**
  Bien qu'aisément explorable, une API doit, malgré tout, être correctement documentée.
- **Atomique**
  Chaque API est vue comme un seul point d'entrée cohérent.
- **Sécurisée**
  Des règles de sécurité concernant l'authentification des utilisateurs, leurs droits ainsi que leurs accès doivent être intégrées.
- **Performante, Scalable et Disponible**
  Les caractéristiques de performance d'une API sont fondamentales pour favoriser l'adoption de celle-ci. Conçue comme une application à part entière, une API doit répondre à des critères de SLA (performance, scalabilité et disponibilité) pré-établies par les équipes selon les exigences métier.
- **Réutilisable**
  Afin de minimiser les coûts de conception et de maintenance, une API doit être utilisée par un maximum de consommateurs sur différents projets.
- **Compatible de manière ascendante**
  La suppression ou la modification d'APIs existantes est souvent tentante au profit de l'intégration de nouvelles fonctionnalités. Une API publiée est avant tout une API utilisée. Pour éviter de forcer les clients à se mettre à jour, il est crucial de prendre en compte la compatibilité ascendante tout au long du cycle de vie d'une API.

L'État de l'art : les APIs REST

REST signifie Representational State Transfer, un terme inventé par Roy Fielding en 2000 pour désigner un style d'architecture pour la conception d'applications à couplage faible, souvent utilisé dans le développement de services web. REST n'applique aucune règle sur la façon dont l'implémentation doit être effectuée, seules existent des lignes directrices de conception laissant une certaine liberté pour sa mise en œuvre.

REST est décrit par un ensemble de contraintes architecturales qui tentent de minimiser la latence et les communications réseau, tout en maximisant l'indépendance et l'évolutivité des implémentations de composants. Les six contraintes de REST sont :

1. Client-Server
   Nécessite qu'un service offre une ou plusieurs opérations et qu'il attende que les clients requêtent ces opérations.
2. Stateless
   Nécessite que la communication entre le consommateur de service (client) et le fournisseur de service (serveur) soit sans état.
3. Cache
   Nécessite de qualifier les réponses comme pouvant être mises en cache ou non.
4. Uniform Interface
   Nécessite que tous les fournisseurs de services et les consommateurs d'une architecture conforme à REST partagent une seule et même interface commune pour toutes les opérations.
5. Layered System
   Nécessite la possibilité d'ajouter ou de supprimer des intermédiaires au moment de l'exécution sans perturber le système.
6. Code-on-Demand
   Seule contrainte optionnelle, elle propose que le serveur puisse fournir du code exécutable dynamiquement au client (tel que du Javascript à un navigateur web) pour étendre les fonctionnalités standard de ce client et lui permettre ainsi de gérer des données qui lui sont inconnues.

Design d'API REST

Le protocole HTTP possède des conventions qui fournissent les bases du couplage faible, de la tolérance aux pannes et de la mise à l'échelle. REST étant construit sur ces principes de bases, il est nécessaire de comprendre le fonctionnement d'HTTP et d'en connaître les bonnes pratiques.

Voici un ensemble de bonnes pratiques et l'état de l'art dans le design d'API REST. Au-delà de cette liste, les conventions peuvent évoluer au fil du temps. Le plus important est donc de choisir des conventions en début de projet et de s'y tenir tout au long de celui-ci, de manière à garder une cohérence dans le design de votre API.

Pour commencer, il est nécessaire de faire un rappel des méthodes HTTP :

- **HEAD :** accéder aux métadonnées d'une ressource
- **GET :** accéder à une ressource
- **POST :** ajouter une ressource
- **PUT :** mettre à jour une ressource complète en la remplaçant par une nouvelle version
- **PATCH :** mettre à jour une partie d'une ressource en envoyant un différentiel
- **DELETE :** supprimer une ressource
- **OPTIONS :** lister toutes les ressources disponibles

Ces verbes agissent de manières différentes en fonction du type de ressource : item ou collection.

	http://api.soat.fr/articles	http://api.soat.fr/articles/8765
GET	Liste les articles	Récupère l'article spécifié par l'ID est 8765
POST	Crée un nouvel article	Non supporté
PUT	Méthode non autorisée	Mise à jour de l'article spécifié
DELETE	Supprime tous les articles	Supprime l'article

Une API REST peut être abordée de deux manières.

La première consiste à se concentrer sur les ressources exposées et est appelée *Resource Oriented Design*.

Elle se concentre sur la définition du type de ressources que l'API doit fournir, leurs noms et méthodes ainsi que la relation entre chacune de ses ressources.

Exemple avec une API pour un petit système de blog :

- **GET** /authors/<idAuteur>
  => Récupère l'Auteur dont l'identifiant est <idAuteur>
- **GET** /authors/<idAuteur>/articles
  => Récupère les articles de l'auteur
- **GET** /authors/<ididAuteur>/articles/<ididArticle>
  => Récupère un article précis de l'auteur
- **GET** /users/<idUser>
  => Récupère l'utilisateur associé à l'identifiant <idUser>
- **GET** /users/<idUser>/comments
  => Récupère les commentaires de l'utilisateur
- **GET** /users/<idUser>/comments/<ididComment>
  => Récupère un commentaire précis de l'utilisateur

La seconde approche, E*xperience Oriented Design*, se concentre sur l'expérience utilisateur et/ou sur le type d'appareil qui va consommer l'API.

En effet, une entreprise qui distribue des données pouvant être consommées sur un grand nombre d'appareils différents (smartphone haut de gamme avec un forfait 4G, smartphone bas de gamme avec un forfait 3G, télévision connectée, console de jeux…) souhaitera en premier plan optimiser l'expérience de l'utilisateur. C'est le cas notamment de Netflix qui fournit des applications sur des supports extrêmement divers.

Une solution est de proposer des points d'accès aux APIs (API endpoints) en fonction de la catégorie de l'appareil :

- /ps4/homescreen
  => Contient toutes les APIs propres aux consoles de type ps4
- /smartphone
  => Contient toutes les APIs utilisables sur smarphone

Bonnes pratiques des APIs REST

Privilégier le format JSON au détriment du XML :

Le format XML est par nature plus verbeux que le JSON, provoquant une augmentation du trafic réseau, enjeu non négligeable dans le domaine de la mobilité.

Utiliser les codes HTTP appropriés pour les messages d'erreur :

Les codes commençant par le chiffre 4 concernent les erreurs client et par le chiffre 5 celles du serveur.

Exemple :

- **400 :** Bad Request
- **401 :** Unauthorized
- **404 :** Not Found
- **405 :** Method Not Allowed
- **406 :** Not Acceptable
- **429 :** Too Many Requests
- **500 :** Internal Server Error
- **501 :** Not Implemented
- **502 :** Bad Gateway

Préférer l'usage des noms plutôt que des verbes :

Les noms sont utilisés pour les ressources contrairement aux verbes qui sont utilisés pour les actions et les calculs :

- pour une ressource : `/users/123456` pour accéder à l'utilisateur d'ID 123456
- pour une action : `/calculDistance`

Exemple du cas de la suppression d'une ressource

Une approche pourrait consister à créer explicitement une action REST dédiée à la suppression et tenter d'y accéder. Cette approche héritée de l'approche SOA n'exploite pas le langage uniforme de REST, consistant à appliquer des verbes HTTP sur des objets.

`GET /articles/123/delete_resource`

Il est important de préférer cette approche :

`DELETE /articles/123`

Utiliser des UUID plutôt que des nombres qui se suivent :

Il est préférable de générer des identifiants uniques (Universal Unique Identifier (UUID)) car ils permettent d'éviter la découverte de ressources déjà existantes par des attaquants ou personnes malveillantes.

Préférer l'usage du pluriel pour adresser les collections :

exemple : GET /article**s**

Utiliser un identifiant unique pour spécifier une instance :

exemple : GET /articles/**693463567**

Accéder aux propriétés d'une ressource :

GET /articles/693463567/comments

On récupère ici les commentaires (comments) de l'article (articles) d'identifiant 693463567.

Préférer une casse lowerCamelCase ou snake_case :

En effet, plusieurs types de casse existent : CamelCase (qui possède deux variantes : lowerCamelCase et UpperCamelCase), snake_case ou spinal-case. Le choix de la casse utilisée vous appartient. Le plus important est d'être cohérent, une fois le type de casse choisi, il faudra l'utiliser à tous les endroits nécessaires.

Éviter d'envelopper les collections :

Au lieu de :

```
GET / articles
  {
    data:  [
      {id: 1, ...},
      {id: 2, ...}
    ]
  }
```

Préférer :

```
GET / articles
  [
    {id: 1, ...},
    {id: 2, ...}
    ]
```

Fournir un système de pagination :

- Par nombre de page
  GET /articles?page=42
- Par curseur
  GET /articles?cursor=a8374a8
- Par filtre
  GET /articles?title=API

Fournir un système de filtrage :

Exemple pour récupérer les articles à l'état de brouillon :

GET /articles?state=brouillon

Fournir un système de tri :

Exemple pour trier des articles par date de création :

GET /tickets?sort=created-date

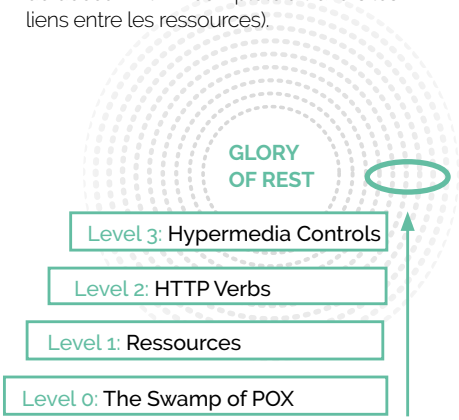
Un mauvais design d'API peut être préjudiciable pour son évolution ainsi que son utilisation. Il est donc important d'apporter un soin particulier à certains points :

- **Absence de cohérence dans le design de l'API.** Une fois l'API exposée, il peut être difficile de revenir sur certains choix effectués préalablement. Vous devez, avant toute conception, avoir une vision précise du besoin et de la valeur proposée.
- **Difficulté de compréhension.** Il faudra veiller à ce que le design de votre API soit le plus compréhensible possible par un développeur et qu'il traduise le fonctionnement de celle-ci.
- **Estimation de l'usage d'une API,** de manière à prendre en compte les problèmes qu'il est possible de rencontrer, notamment lors de la mise à l'échelle. Un choix de données trop volumineuses, lorsqu'elles sont exposées peut entraîner des problèmes de performance par la suite. Bien qu'il soit difficile d'anticiper l'utilisation d'une API par des développeurs tiers, il est possible d'être au plus proche du besoin en adoptant une démarche de conception itérative.

Le modèle de maturité de Richardson

Le modèle de maturité REST développé par Leonard Richardson, tente de classer une API en fonction des contraintes imposées par REST. Plus votre API répondra à ces exigences plus elle sera performante. Il y a quatre niveaux d'exigence. Le niveau le plus bas est le niveau 0, il désigne la mise en œuvre la moins conforme.

- Niveau 0 : Une seule URI, le protocole HTTP est utilisé comme système de transport (SOAP, XML, RPC).
- Niveau 1 : Plusieurs URI, un seul verbe HTTP.
- Niveau 2 : Plusieurs URI, plusieurs verbes HTTP (CRUD).
- Niveau 3 : Utilisation d'hypermedia (ajout de métadonnées dans les réponses permettant de découvrir l'API complète à travers les liens entre les ressources).



Le niveau 2 est le plus régulièrement rencontré sur les différents projets. Le niveau 3 demande un effort supplémentaire et peut se révéler plus difficile à atteindre en fonction de la maturité de vos équipes de développement.

Gestion du changement

"Successful software always gets changed" - Frederic P. Brooks

Une solution logicielle constituée d'un ensemble d'APIs est un système distribué de composants faiblement couplés. Au-delà de la distribution des différents composants, c'est la responsabilité de ces composants qui est distribuée.

L'évolution des APIs est rendue possible à travers la mise en place d'un système de gestion de versions des APIs. Ainsi, lorsqu'une API change, les clients la consommant doivent toujours être en mesure de l'utiliser. Le cas échéant, il est nécessaire de mettre en place des stratégies de versioning pérennes et tangibles.

Compatibilité ascendante

Afin de gérer les différentes versions d'une API, un mécanisme de compatibilité doit être mis en place. Lorsqu'un client peut continuer à utiliser l'ensemble des fonctionnalités de la version précédente de l'API, on appelle cela la compatibilité ascendante. Le développeur de l'API doit alors continuer à maintenir, sécuriser et corriger toutes ses anciennes versions.

Certains changements sont généralement permis :

- Ajout de paramètres, d'en-têtes d'une URI, etc.
- Ajout de champs supplémentaires dans le contenu XML ou JSON, en portant une attention particulière aux librairies de sérialisation/désérialisation qui nécessitent une structure figée.
- Ajout de endpoints supplémentaires (tels que l'ajout de ressources de type REST).

A contrario, d'autres génèrent des incompatibilités :

- Le changement dans la structure de données (modification ou suppression de champs).
- Suppression d'un champ dans la requête. Il est nécessaire dans ce cas de le rendre optionnel.
- Changement d'URIs.

Ces contraintes doivent être respectées pour assurer la pérennité de vos APIs ainsi que la compréhension qui en est faite par les développeurs amenés à les utiliser.

Différentes pratiques existent aujourd'hui pour la gestion des versions d'une API :

- Dans l'URL : `https://api.soat.fr/v2/articles/17238723`
- Dans un header custom : `X-API-Version: 2`, `GData-Version: 2.0`, `X-MS-Version: 2017-01-12`
- Avec le media type : il est possible de spécifier la version en spécifiant dans le header Accept la version souhaitée `application/vnd.status.v2+json`

L'Hypermedia comme solution de secours

L'approche Hypermedia possède des caractéristiques qui peuvent contribuer à la longévité des logiciels. Elle apporte une certaine flexibilité en séparant l'implémentation de l'API de la manière dont elle est consommée par un client. Les clients suivent dynamiquement des liens fournis en réponse aux requêtes..

Provisioning

Une approche alternative consiste à mettre à disposition un composant logiciel de configuration contenant la logique d'accès à l'API et à l'extraction de données. Ce composant de configuration logicielle est mis à jour par le fournisseur de l'API à chaque nouvelle version. Le client consommant l'API interroge à chaque requête le composant de configuration pour disposer de l'ensemble des informations d'utilisation. Ce mécanisme est en particulier très utilisé dans le monde de l'IoT (Internet of Things) où la mise à jour du composant logiciel déployé sur le matériel est difficile voire impossible.

Sécurisation d'une API

Une Web API, comme toute application web, est vulnérable aux attaques de sécurité les plus fréquentes notamment celles définies dans le top 10 de l'OSWAP[1] : injection, XSS, CSRF, exposition de données sensibles, authentification défectueuse…

Par ailleurs, une Web API est exposée à des problématiques supplémentaires comme une utilisation frauduleuse suite à la récupération d'une clé privée ou d'un identifiant utilisateur - obtenu par exemple à la suite d'une rétro-ingénierie d'une application mobile - ou encore, suite au vol d'une suite de clés après une intrusion dans votre système. L'injection de schémas JSON ou XML est une autre forme d'attaque qu'il est possible de rencontrer. La liste est longue et l'innovation florissante des attaquants peut s'avérer redoutable. Il est, dès lors, indispensable d'inclure la sécurisation de votre API au cours du développement de celle-ci, de manière à profiter au plus tôt du retour d'expérience des utilisateurs. Il est malheureusement trop fréquent de voir aborder la sécurité sur la fin des projets de développement, lorsque les échéances sont trop proches et les budgets serrés.

Pour ce faire, nous vous proposons d'aborder la sécurisation de votre API au travers d'une liste de points à examiner, tout au long de votre projet :

- Authentification du client de l'API (ex : application mobile)
- Authentification de l'utilisateur final
- Gestion des quotas et du trafic
- Validation du contenu : schéma JSON, schéma XML
- Sécurisation de la couche de transport
- Chiffrement du contenu
- Scan du contenu pour identifier de potentielles attaques (SQL injection)
- Stockage des journaux de logs
- Détection des attaques automatiques ou bots
- Outils de monitoring et d'alerte
- Outil de gestion des clés d'API

Sécuriser une API est extrêmement important, mais il est préférable de ne pas ajouter une complexité qui ne serait pas nécessaire. Aujourd'hui, plusieurs possibilités existent, avec différents niveaux de protection.

1. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Sécurisation par HTTP

HTTP propose Basic Access Authentification : une requête fournit, via les entêtes HTTP, un couple identifiant/mot de passe. C'est un système d'authentification très simple, mais qui n'est pas sécurisé. Quiconque écoute le trafic pourra intercepter identifiant et mot de passe. Le trafic se doit alors d'être chiffré, en ayant par exemple recours à HTTPS, rendant ainsi le trafic illisible par vos attaquants.

JSON Web Token

JSON Web Token (JWT) est un standard ouvert (RFC 7519) pour échanger de l'information de manière sécurisée via un jeton signé (définition Wikipédia : https://fr.wikipedia.org/wiki/JSON_Web_Token). JWT permet de s'affranchir d'une authentification / identification systématique. Une fois le token généré, le client peut le fournir à différents services qui se baseront uniquement sur ces informations pour autoriser ou non l'accès.

Même s'il est conservé par le client, le token étant signé, la donnée est sécurisée. Une modification de son contenu en révoquerait la validité. JWT est donc autoporté : votre serveur d'authentification n'est plus le goulot d'étranglement de votre application.

En revanche, vous devez porter une attention toute particulière à votre politique de répudiation des tokens. Avec une durée de validité trop longue, un utilisateur non autorisé continuera à accéder à votre API, tant que le token n'aura pas été invalidé.

OAuth 2.0

Le framework OAuth 2.0 permet à une application d'obtenir un accès limité aux comptes utilisateurs d'un service HTTP tel que Facebook, Google, GitHub ou Twitter. Il fonctionne en déléguant l'authentification d'un utilisateur et en autorisant les applications tierces à accéder à son compte. OAuth 2.0 fournit des flux d'autorisation pour les applications web ou mobiles.

Les nouveaux nés

Produire une API qui répond à toutes les problématiques et tous les besoins est très complexe, car le contenu que l'on souhaite afficher sur une page web n'est pas strictement le même que sur une application mobile (taille d'écran, visibilité, expérience utilisateur...).

Face à cette situation, deux géants du web ont conçu deux produits : GraphQL pour Facebook et Falcor pour Netflix. Ces produits permettent d'interroger leurs APIs, tout en ayant des réponses adaptées aux supports.

GraphQL

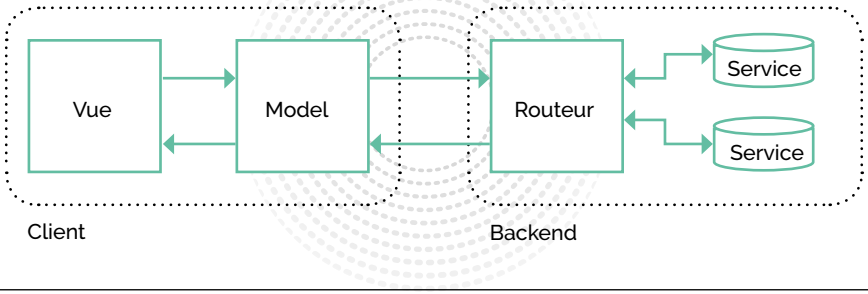
GraphQL est orienté produit : une requête contient l'ensemble des champs dont le client a besoin en réponse. C'est donc le client qui contrôle le format de la réponse qui sera spécifiquement adaptée au contexte. De plus, l'ensemble des données nécessaires à un écran transite à travers une requête unique. Il est ici inutile de devoir interroger plusieurs services, avec des formats différents, pour obtenir toutes les informations nécessaires.

GraphQL préconise l'utilisation d'un point d'entrée unique et n'impose pas d'utilisation des méthodes HTTP. La présence, en réponse, d'un champ d'erreur indique un problème lors du traitement de la réponse.

Cela peut entraîner des incohérences telles que la présence d'une erreur malgré une réponse HTTP OK 200 ou encore utiliser un HTTP POST pour lire des données. Ces problèmes sont semblables à ceux déjà rencontrés avec SOAP. Si vous voulez implémenter GraphQL sur vos projets, vous devez prendre en compte ces manques pour mitiger ces futurs problèmes.

GraphQL propose donc une solution pour générer des réponses adaptées à différents contextes. Toutefois, la présence de certaines incohérences techniques empêchera GraphQL de devenir le remplaçant de REST. Reste qu'une coopération entre ces domaines vous aidera à construire des APIs efficaces, robustes et souples.

Des implémentations de GraphQL existent en C#, Java, Javascript, PHP, Ruby ou encore Clojure. Cela ne devrait pas être un problème pour l'adopter sur votre prochain projet.

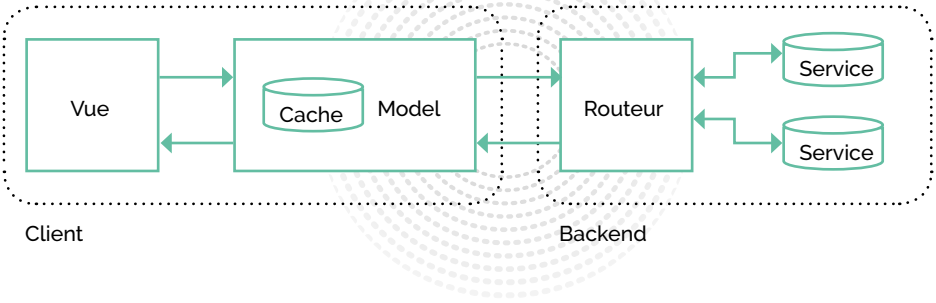


Falcor

Netflix est disponible sur quasi tous les supports possibles. L'entreprise doit alors gérer une multitude de clients hétérogènes : application web, smartphone, tablette, télévision, système embarqué, etc. Pour récupérer ces données spécifiques au contexte client, sans multiplier les APIs, Netflix a conçu Falcor.

Si, fonctionnellement, Falcor semble très proche de GraphQL, il est techniquement différent. Falcor propose un cache local et optimise les requêtes réseaux soit en dé-dupliquant les requêtes soit en fusionnant différentes requêtes pour n'émettre qu'une requête réseau effective.

Mais tout le monde ne pourra profiter de ces fonctionnalités. Netflix a conçu Falcor pour répondre principalement à ses propres besoins et propose uniquement un client Javascript et des implémentations serveur qui s'opèrent via NodeJS. Si vous souhaitez utiliser Falcor dans un autre contexte (Java, C#…), vous aurez à vous charger des implémentations clientes. De même, côté serveur, vous pourrez faire communiquer vos applicatifs avec le serveur Falcor, mais vous aurez tout de même besoin de mettre en production une technologie NodeJS, qui peut sortir de votre catalogue technologique habituel.



PARTIE II :

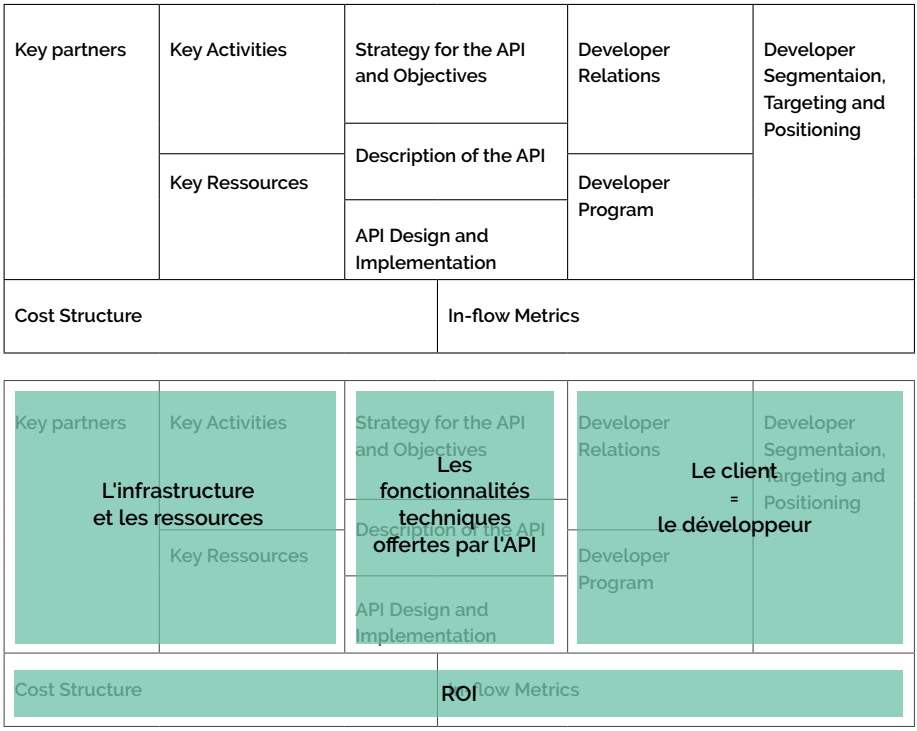
API as Product

Comme énoncé dans les chapitres précédents, les Web APIs relèvent en grande partie d'enjeux et de problématiques techniques. Il serait cependant réducteur de ne les envisager que sous un prisme technologique. Une Web API peut devenir une opportunité stratégique de revenu et dans ce cas, un produit à part entière. Celle-ci est donc exposée aux mêmes contraintes que tout produit commercial. Elle doit être accompagnée d'un ensemble de pratiques permettant son adoption rapide et facile, et ainsi produire une expérience utilisateur unique, satisfaisante et sans friction.

Mais transformer une API en produit ou la présenter en tant que tel n'est pas chose aisée. Cela peut en effet entraîner des questions sur les modalités d'exposition des données, la facturation de l'API et par conséquent sur le business model inhérent à cette transformation. Pour travailler sur son business model, il existe des solutions bien connues des startups qui permettent une modélisation synthétique : le Business Model Canvas créé par Alexander Osterwalder qui se concentre sur les partenaires, les ressources et les activités, et le Lean Canvas d'Ash Maurya qui se concentre sur l'adéquation du produit au marché. Pour les APIs, un modèle dédié a émergé : l'API Model Canvas.

API Model Canvas

À première vue, l'API Model Canvas peut sembler similaire aux modèles énoncés précédemment. Cependant, celui-ci se concentre sur l'économie des APIs et la transformation d'une API en produit.



Comme on l'observe très clairement sur le schéma ci-dessus, le client est désormais un développeur. Trois sections sont donc dédiées ici à la mise en avant de votre client : Developer Relation, Developer Program et Developer Segmentation. Ces trois sections sont aujourd'hui regroupées pour représenter l'expérience du développeur.

Au même titre que les interfaces graphiques, qui possèdent un recueil de bonnes pratiques réuni sous le terme User Experience (UX), les Web APIs disposent elles aussi d'une suite de pratiques qui vise à améliorer l'expérience du développeur, réunie sous le terme Developer Experience (DX).

Developer Experience

La notion d'expérience développeur doit être traitée de la même façon que l'expérience utilisateur. Il est donc primordial de bâtir une API optimisée pour vos clients cibles et de bien comprendre son audience ainsi que ses besoins. Pour que vos clients puissent utiliser au plus vite votre produit, et de la manière la plus efficiente, vous devez lui fournir une expérience d'onboarding sans friction (frictionless).

Quatre grands axes doivent être adressés pour fournir un bon niveau d'expérience :

- **La communication**
- **Les outils et techniques**
- **Le support et l'assistance**
- **La documentation**

La communication

La valeur business de votre API doit être exposée clairement aux développeurs qui se rendent sur l'interface de présentation du produit. Vous devez mettre en avant ses avantages, sa simplicité d'usage, le gain de temps ou tout autre avantage concurrentiel qui lui est propre. Son pricing doit être clairement défini pour que le développeur puisse se projeter et anticiper les coûts associés à l'usage de votre service.

L'usage du produit doit être le plus limpide possible et un ensemble de données et d'usages doivent être fournis aux développeurs. À ce titre, les statistiques d'utilisation sont primordiales car elles permettent aux développeurs de votre API d'appréhender l'utilisation qui en est faite et le cas échéant, d'estimer son coût.

Cette exposition des usages se fait généralement via un portail dédié aux développeurs (Developer portal) offrant un accès à la documentation, la FAQ (voir ci-après), aux statistiques et à la facturation. Ce type de portail est aujourd'hui inclus dans les logiciels d'API Management.

Les outils et techniques

Il est primordial de proposer rapidement des SDK (Software Development Kit) dans les langages les plus populaires, afin d'optimiser la montée en compétence des développeurs tiers et la rapidité d'adoption de votre API. Dans le cas contraire, ces SDK seront probablement développés par une communauté, rendant éparse, diluée et hors de contrôle la Developer Experience.

Par exemple, un éditeur de solutions de paiement sur mobile proposera un SDK de son API sous Android pour adresser les développeurs d'applications Android et un SDK en langage Swift pour adresser les développeurs d'applications iOs (plateforme Apple).

"Il est primordial de bâtir une API optimisée pour fournir une expérience sans friction"

Le support et l'assistance

Vous devez fournir une roadmap très claire et explicite sur le support de votre API ainsi que sur ses différentes versions.

Le support dédié à vos utilisateurs doit être pris en charge par des personnes ayant un bon niveau technique et possédant une certaine empathie envers les clients, pour qu'ils se sentent soutenus dans leur utilisation et que leurs interrogations soient comprises.

La prise en charge de l'assistance peut s'effectuer au travers de différents outils tels que le chat, les logiciels de gestion d'incidents, un forum ou désormais des chatbots. Il est également recommandé de mettre à disposition une FAQ qui sera régulièrement mise à jour et enrichie avec les retours clients. Elle offrira ainsi un gain de temps considérable dans la prise en charge des demandes et aidera les clients à régler les problèmes les plus fréquemment rencontrés.

Il est aussi très important de présenter des métriques concernant l'état de santé de votre système (API Health) sur une page dédiée de votre portail de développeurs. Cette page devra contenir les différents statuts (alive, error...), les temps de réponse moyen, l'historique des incidents ou encore les statuts de mise à jour.

La documentation

"If It Isn't Documented, It Doesn't Exist"
Jeff Atwood, Coding Horror[1]

Utiliser une API Web sans documentation, c'est un peu comme se retrouver au milieu de nulle part, seul, sans carte ni GPS. La documentation, quelle que soit sa forme, fournit un cadre, une direction et les intentions derrière une API web. Pour qu'une API soit exploitable, et correctement utilisée, une documentation précise et exhaustive doit l'accompagner. Dans le cas contraire, vos utilisateurs auront beaucoup de peine à implémenter les interactions avec vos services. Les acteurs majeurs du web (GitHub, Amazon, Facebook…) ne s'y sont pas trompés : une documentation de qualité est fournie pour chacune de leurs APIs, favorisant ainsi leur adoption.

Pour répondre aux critères d'une documentation de qualité, il est important que celle-ci contienne un ensemble de contenus, décrit sous trois catégories :

- La documentation de référence
- Le workflow
- Les tutoriels

La documentation de référence

La documentation de référence répond à la question : Que fait cette API ?

Cette documentation décrit les endpoints ainsi que les requêtes et réponses de votre API. Elle peut être écrite ou générée à partir de votre application. La documentation manuelle offre un contexte, un fil rouge ou des explications détaillées qu'une documentation générée automatiquement n'offrira pas. Le coût d'écriture, selon l'API, peut être conséquent et complexe : la documentation expose-t-elle bien les utilisations possibles de votre API ? Est-elle simplement à jour ? Plus l'API devient conséquente, plus la mise à jour de la documentation va devenir complexe et coûteuse.

L'automatisation de la documentation se présente comme la solution la plus évidente à ces problèmes. Mais il existe plusieurs niveaux d'automatisation. Une documentation peut être générée à partir des commentaires contenus dans votre API, comme une Javadoc, à partir d'une spécification ou encore directement à partir de vos APIs, en s'appuyant sur les éléments techniques, comme le font les outils de langage de description API RAML, Blueprint et Swagger.

La documentation vivante (living documentation) permet de lier une documentation manuelle et une documentation automatique. Elle fera le lien à partir d'éléments différents : texte, code ou d'une autre documentation générée automatiquement. Le livrable généré contiendra tous les éléments nécessaires à la bonne compréhension et utilisation de votre API.

1. https://blog.codinghorror.com/if-it-isnt-documented-it-doesnt-exist/

Le workflow

Le workflow répond à la question : Comment dois-je procéder pour utiliser cette API ?

Le workflow fournit une vue d'ensemble des appels à effectuer vers l'API pour une fonctionnalité donnée. Il met en avant cette orchestration et permet d'expliciter, par exemple, le lien entre différentes méthodes REST.

Si le workflow n'est pas respecté, le risque d'erreur sera plus élevé. Pour limiter ce risque, votre API doit expliciter techniquement l'ordre dans lequel elle peut être appelée afin de faciliter sa bonne utilisation. Par exemple :

- L'invocation d'une API génère en sortie une valeur précise. Celle-ci doit être un paramètre d'entrée requis à l'invocation de l'API suivante dans le workflow.
- En réponse d'une méthode, fournir la suite possible du workflow, avec la liste des méthodes suivantes possibles. Cette technique s'appuie généralement sur l'approche Hypermedia.

Tutoriel

Les tutoriels répondent à la question : Par où puis-je commencer à utiliser cette API ?

Les tutoriels doivent être basés sur les use cases de votre API. Ces derniers sont importants car ils permettent aux développeurs de se projeter dans une utilisation concrète de votre produit.

Prenons l'exemple d'une API mettant à disposition des outils facilitant des processus de communication (SMS, notification…), plusieurs use cases peuvent être présentés aux développeurs :

- Use case 1 : Envoi de notification
- Use case 2 : Validation de numéro de téléphone
- Use case 3 : Support client par SMS
- Use case 4 : Authentification sans mot de passe

Chaque use case permettra d'expliciter l'utilisation et la prise en main de votre API ainsi que le workflow et les interactions nécessaires entre elle et son consommateur.

Les outils de documentation

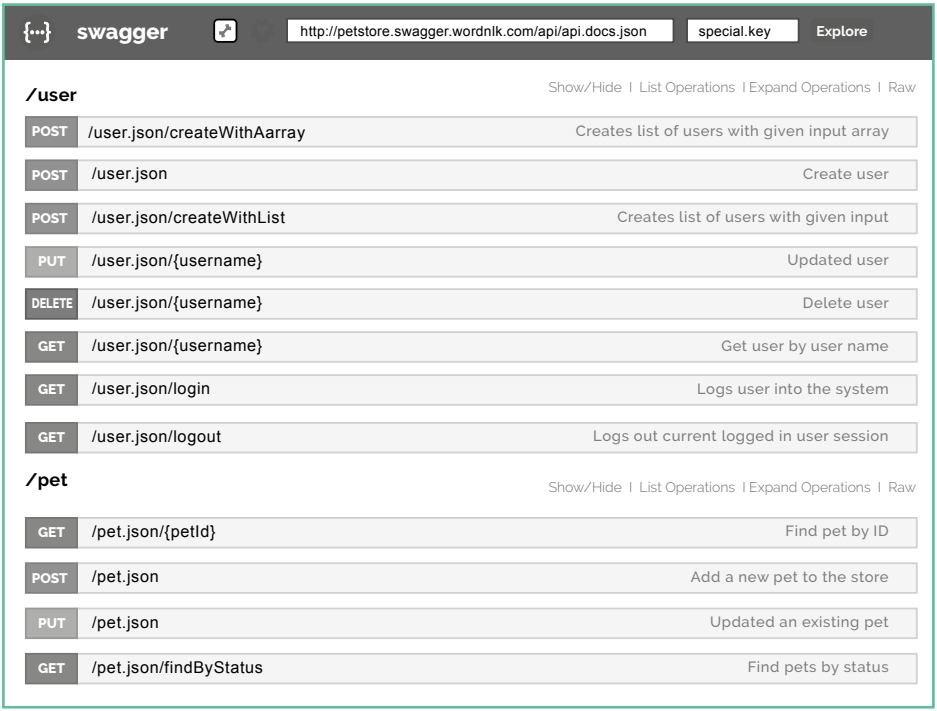
Open API (Swagger)

Swagger est un outil conçu par l'entreprise Worknik pour documenter ses propres APIs. Les spécifications ont été offertes par la société SmartBear à l'organisation Open API Initiative sous le nom d'Open API Specification.

Swagger est composé de plusieurs outils et, grâce à sa popularité, une liste conséquente d'intégrations pour une grande variété de langage/framework.

Si vous utilisez Swagger, n'oubliez pas de mettre en place swagger-ui pour afficher votre API sous forme de documentation directement testable depuis un navigateur web. À partir d'une spécification Swagger, une page web avec tous les points d'accès de votre API sera disponible avec un exemple de requête et de réponse.

La spécification Swagger doit être écrite en JSON ou en YAML et représenter votre API. Vous devez utiliser un module d'intégration pour générer votre API technique. Si au contraire, vous voulez que votre API technique serve de base à la génération de votre documentation - ce que l'on vous préconise - alors vous devez utiliser un autre module d'intégration, spécifique à la technologie que vous utilisez. Cette génération automatique assure une documentation toujours à jour et cohérente de votre API.



RAML

RAML est aussi un langage de description des APIs. Créé par la société MuleSoft, RAML est l'abréviation de RESTful API Modeling Language. Au-delà de ce langage, un ensemble d'outils est disponible pour faciliter la description, la visualisation, la production et la consommation des APIs. D'un point de vue plateforme d'API, les plateformes MuleSoft Anypoint, 3scale et Restlet supportent RAML.

RAML est basé sur le langage YAML.

Une description d'API en RAML contient les éléments suivants :

- Des informations basiques sur les APIs comme son nom, son titre et la personne rédigeant la documentation.
- Les ressources incluant les méthodes, les schémas et les paramètres.
- Des éléments réutilisables comme les types de ressources ou des éléments de sécurité.

YAML décrit des données hiérarchiques (similaire au XML) en utilisant des espaces. Comparé au XML, YAML est considéré généralement comme plus concis et plus lisible par les développeurs. Un fichier YAML est un fichier de propriétés d'objets de types clés-valeur ou de liste. Une majorité d'outils et de frameworks utilise désormais le langage YAML pour décrire les fichiers de configuration. L'émergence de YAML se fait au détriment du langage JSON, réservé à l'échange de données entre composants logiciels et dont l'analyse est extrêmement facile par les technologies JavaScript.

API Blueprint

API Blueprint est l'outsider des outils de documentation : les spécifications des APIs sont dans le format Markdown, les rendant très compréhensibles, même pour une personne non-technique. Le choix de ce format offre la possibilité d'y insérer également de la documentation, des images, etc.

API Blueprint propose quelques outils, mais la gamme est moins large que RAML ou Swagger. Par exemple, vous ne trouverez pas systématiquement de module de génération de code à destination de votre API technique. La puissance d'API Blueprint se retrouve à travers la plateforme de son éditeur : apiary.io. Cette plateforme de très bonne qualité offre une gestion d'équipe, un générateur de documentation et le partage d'API. Avant de considérer API Blueprint pour votre prochain projet, vous devrez évaluer les impacts possibles de cette plateforme sur votre projet (dépendances, coûts).

Son récent rachat par Oracle devrait l'aider à améliorer et faciliter son intégration technologique, ce qui fera naturellement grandir sa communauté.

Quelle solution choisir ?

Il n'y a pas de solution idéale pour tous les projets. Même si Swagger est l'outil le plus populaire actuellement, RAML et API Blueprint sont des solutions à ne pas négliger. Elles peuvent, par leurs formats de description, par les modules d'intégration ou par les outils, être plus intéressantes selon votre contexte. Le tableau ci-dessous se veut être une aide dans votre prise de décision.

Mesure et facilité d'usage

Au-delà de la qualité de service (SLA), il est important de mesurer la facilité d'usage de votre API. Plusieurs indicateurs vous permettent de la suivre et ceci de plusieurs manières :

- **Time To First Call** : Quel est le temps de compréhension nécessaire avant qu'un développeur fasse le premier appel à votre API ?
- **Concept** : Quels sont les concepts que le développeur a besoin de connaître pour utiliser votre API ?
- **Gestion des erreurs** : votre API et le tableau de bord associé doivent remonter des erreurs explicites et compréhensibles.

Ces indicateurs doivent vous permettre de mesurer la facilité avec laquelle un développeur va pouvoir prendre en main votre API. Les points abordés précédemment - documentation, communication, support et outils - lui permettront de rapidement expérimenter des utilisations de votre API dans des cas concrets et pratiques.

	Swagger	RAML	API Blueprint
Sponsor	Smartbear	MuleSoft	Apiary
Format de spécification	JSON	YAML	Markdown
Plateforme online	Swagger Hub	API Portal	Apiary.io
Communauté	Large	Grandissante	Petite
Générateur de code	Oui	Oui	Oui mais choix restreint

Monétisation d'une API

Les Web APIs sont passées d'un outil d'interconnexion entre applications à un produit à part entière qu'il est possible de commercialiser et donc, de monétiser. Auparavant gérées par des équipes entièrement composées d'ingénieurs, celles-ci sont désormais pluridisciplinaires (marketing, produit, développement...) pour mieux appréhender le contexte business associé à ce type de produit technique.

Étant considérées comme un produit, les Web APIs peuvent agir comme un effet levier pour votre business model. Vous devez définir si vous facturez l'usage et/ou le volume, si c'est le développeur qui paie ou si c'est vous qui le payez. Voici un petit panorama des différentes stratégies de monétisation d'une API.

Dans le cas où le développeur paie

Cette stratégie représente typiquement le cas où le développeur paie pour le service rendu par l'API. Plusieurs déclinaisons existent telles que :

- **Facturer le nombre d'appels ou d'utilisateurs de l'API.** Cette méthode, aussi appelée "Pay as You Go", est la plus communément utilisée, car elle est facilement applicable et quantifiable. Plus le nombre d'appels à l'API augmente, plus la facturation va elle-même augmenter. Il peut être difficile de se projeter dans le nombre de clients qui va consommer votre API mais si leur nombre augmente, vos frais seront couverts par ce type de pricing. Pour aider à la prise en main de l'API et à la compréhension de sa valeur par les développeurs, il est conseillé de mettre en place un modèle freemium limité dans le temps ou dans le nombre de requêtes.
- **Facturation à la transaction.** Chaque transaction, à l'aide de l'API, est facturée par un prix fixe ou proportionnel au montant. Cette méthode est particulièrement employée par les sociétés de paiement en ligne telles que Paypal ou Stripe.
- **Le modèle freemium.** Le développeur dispose d'un nombre de requêtes pour lequel le service est gratuit. Dans le cas où le développeur souhaite obtenir des services supplémentaires comme des SLA plus importantes ou un nombre de requêtes plus élevé, il devra alors s'affranchir d'un montant supplémentaire. Cette méthode est notamment utilisée par Google pour l'API de Google Maps.

Dans le cas où le développeur est payé

Ce type de monétisation concerne l'affiliation. Si vous disposez d'un site d'E-commerce, vous pouvez décider d'exposer une API qui permettra à d'autres développeurs de placer certains de vos articles sur leurs sites. Une fois la vente d'un article en provenance du site affilié, plusieurs modes de rémunération sont alors possibles. Lorsque vous mettez en place ce type de monétisation, il faut choisir celui qui convient le mieux à l'intégration de votre API par un développeur :

- **CPA (Coût Par Action).** Utilisé majoritairement sur les sites de E-commerce, le développeur est rémunéré à chaque fois qu'un internaute clique sur le lien d'affiliation et effectue un achat sur le site marchand. Un pourcentage de l'achat est alors reversé au développeur. On retrouve ce type de rémunération chez Amazon.
- **CPC (Coût par Clic).** La rémunération du développeur se fait à chaque fois qu'un internaute clique. Le cas le plus connu est la régie publicitaire de Google : AdSense.
- **Revenue sharing.** Vous rémunérez le développeur en fonction de chaque dépense qu'un consommateur fait sur votre site après s'être inscrit depuis le site tiers. On retrouve ce type de rémunération sur les sites de jeux en ligne.

La stratégie de partenariat B2B

L'exposition d'une API peut aussi se faire dans le cas d'une stratégie de partenariat B2B. Elle permet l'intégration de votre service par une entreprise tierce dans le cadre d'un marché défini.

La stratégie d'Upselling

Enfin et au-delà des types de monétisation que nous avons vus précédemment, l'exposition d'une API peut être intégrée dans une stratégie d'Upselling de votre produit. Pour rappel, l'Upselling est une pratique qui consiste à proposer une montée en gamme sur un produit à un consommateur clé.

Salesforce (logiciel de CRM) est un exemple de société qui propose son API à partir d'une certaine tarification. Le client doit donc choisir cette tarification pour pouvoir intégrer Salesforce aux autres outils présents dans son système d'information.

Conclusion

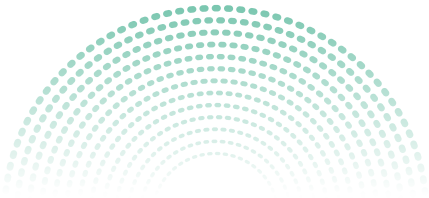
Parce qu'elles interconnectent les systèmes dans un monde de plus en plus ouvert et décloisonné, les APIs constituent l'un des principaux moteurs aux projets de modernisation des SI.

En apportant plus d'interopérabilité, de rapidité et une optimisation des échanges de données, elles permettent de gérer une multitude de projets SI avec une efficience maximale et participent ainsi à la réussite de la transformation digitale des organisations.

Reste que la conception d'une API implique de bonnes pratiques de développement, doit répondre à de nombreuses exigences de qualité, et suppose de bien anticiper ses impacts organisationnels ainsi que sa sécurité. L'adoption elle-même d'une API ne peut se faire que de manière méthodique et progressive, car elle induit un changement profond d'architecture.

Mais dès lors qu'elles sont conçues selon les règles de l'art et utilisées à bon escient, les APIs peuvent libérer le potentiel d'innovation des organisations et créer de nouvelles synergies : le Big Data, l'Open Data, la Mobilité, la Sécurité, l'IoT... toutes ces disciplines transversales vont en effet pouvoir être fédérées par leur mise en place.

Si aujourd'hui, il est donc impératif de réfléchir aux APIs lors de la création d'un produit, dans certains cas, il sera même plus stratégique de créer une API qui est un produit. Culturellement, cette approche reste très novatrice mais prendra de l'ampleur dans les prochaines années. Le fait que les APIs deviennent en elles-mêmes « des services », avec leur propre système de monétisation, va contribuer à la propagation de leurs activités et bénéfices, conférer un rôle de plus en plus majeur aux développeurs et ouvrir de nouvelles possibilités d'utilisation de leurs ressources.



Auteurs

Web API : Edition 2017

Grégory Boissinot

Bertrand Lehurt

David Wursteisen

CONCEPTION GRAPHIQUE ET RÉALISATION : MAP ADVERTISING

www.soat.fr - blog.soat.fr

Sequana 1
89 quai Panhard et Levassor
75013 Paris
01 44 75 42 55