Sur le back-end

les problèmes d'évolutivité, de fiabilité et de maintenabilité sont traités par le

intégration de logiciels dans des conteneurs virtualisés en cours d'exécution

microservices. Ces services exposent la programmation d'applications

des interfaces (API) pouvant être utilisées par les développeurs pour créer des applications.

Pour simplifier l'utilisation des API, un certain nombre de documentations sur les API

des normes ont été proposées ; pour les services Web basés sur REST, le

La spécification OpenAPI [10], anciennement connue sous le nom de Swagger, est devenue

le leader apparent du marché. Un écosystème d'ensembles d'outils connexes a

évolué autour de la spécification offrant la génération de code et la sim-

ple frontaux Web. Bien que ces outils soient conviviaux pour les développeurs, ils

ne sont pas vraiment destinés à être utilisés par des non-développeurs comme les concepteurs

et les utilisateurs finaux. Par exemple, les interfaces utilisateur générées contiennent de nombreux

détails techniques comme les schémas JSON et les codes d'erreur possibles. À

cette fin, notre objectif principal est d'automatiser le processus fastidieux

de la création d'interface utilisateur de l'API à l'interface Web. Nous voulons

être capable de générer des prototypes d'applications dont la conception peut être

adapté aux demandes spécifiques de l'utilisateur dans une étape consécutive. Par là, nous

libérer en outre des ressources de développement pour s'adresser à d'autres utilisateurs

conditions.

https://eudatasharing.eu/fr/technical-aspects/api-guidance-partie-1

https://mina7.net/pin/#google\_vignette

http://www.christian-faure.net/2016/02/07/architecture-des-apis-dans-les-systemes-distribues/

username = ‘guest’ et mot de passe = ‘guest’

Pour la manipulation de ce serveur à travers une interface utilisateur, nous avons activés les plugins de gestions en saisissant dans le terminal RabbitMQ : « rabbitmq-plugins enable rabbitmq\_management ».

Dans cet exemple nous avons utilisés rabbitmq-server-3.8.16.exe, Erlang (otp\_win64\_23.3.exe) et Eclipse (nécessite l’importation des deux librairies amqp-client.jar, slf4j-simple.jar et slf4j-api-1.7.26).

Sans serveur central il se connecte au serveur le plus proche.

Les sites :

<https://apluseduc.com/381-un-apercu-sur-les-differents-reseaux-informatiques>

<https://apluseduc.com/381-un-apercu-sur-les-differents-reseaux-informatiques>

Pour envoyer les données dans le Cloud les objets connectés passe par un réseau de communication particulier, ces réseaux font intervenir différent protocole de communication pour définir le contenu, le transport et le format des données. AMQP, MQTT sont des protocoles le plus utilités pour ce processus.

Mqtt faible consommation en énergie, rapidité d’exécution et sa capacité de chiffrement et lourd pour les réseaux cellulaire Iots NB-IoT et ne prend pas en compte les services du divices management.

Les API REST sont utilisées comme mécanisme d'intégration d'applications prédominant sur Internet.

Dans un environnement aux ressources limitées, il est important de comprendre la valeur.

La documentation d'une API REST est importante pour son adoption réussie. Les API exposent les données et les services que les consommateurs souhaitent utiliser. Une API doit être conçue avec une interface que le consommateur peut comprendre. La documentation de l'API est essentielle pour que les développeurs d'applications comprennent l'API. La documentation doit aider le développeur à découvrir la fonctionnalité de l'API et lui permettre de commencer à l'utiliser facilement. Ce chapitre examine les aspects de la documentation d'une API et certains des outils et technologies disponibles pour la documentation des API, notamment RAML, Swagger, API Blueprint et autres.

Documenting a REST API is important for its successful adoption. APIs expose data and services that consumers want to use. An API should be designed with an interface that the consumer can understand. API documentation is key to the app developers comprehending the API. The documentation should help the developer to learn about the API functionality and enable them to start using it easily. This chapter looks at the aspects of documenting an API and some of the tools and technologies available for API documentation, including RAML, Swagger, API Blueprint, and others.

Alors que de plus en plus d'appareils Internet des objets (IoT) sont connectés aux services cloud, le fog computing est devenu un nouveau modèle informatique pour fournir des services analytiques et intelligents aux utilisateurs finaux pour un temps de réponse rapide. Dans les applications IoT, le modèle de publication/abonnement a été considéré comme un meilleur modèle en raison d'une bande passante réseau plus faible et d'un traitement moindre des messages, ce qui prolonge la durée de vie des appareils fonctionnant sur batterie.

A different way for devices to communicate on a network is called publish-subscribe, or pub-sub. In a pub-sub architecture, a central source called a broker (also sometimes called a server) receives and distributes all data. Pub-sub clients can publish data to the broker or subscribe to get data from it—or both.

Clients that publish data send it only when the data changes (report by exception, or RBE). Clients that subscribe to data automatically receive it from the broker/server, but again, only when it changes.

The broker does not store data; it simply moves it from publishers to subscribers. When data comes in from a publisher, the broker promptly sends it off to any client subscribed to that data.

In our truck analogy, there are no empty trucks. A client publishing data sends a full truck to the broker. The broker sees the truck come in but doesn’t unload it; it simply routes it intact to a subscriber (cloning the truck if there’s more than one subscriber).

In the diagram below, the client on the left publishes data that the clients on the right subscribe to. In addition, the client in the lower right also publishes data needed by other clients not shown.

Le terme **décrire** n’est pas utilisé par hasard car implémenter ces spécifications permet entre autres :

* d’obtenir une documentation ([Swagger UI](http://swagger.io/swagger-ui/)) ;
* et de générer des clients permettant d’interagir avec notre API ([Swagger Codegen](http://swagger.io/swagger-codegen/)).

Les spécifications permettent de créer un fichier JSON qui décrit l’ensemble des éléments d’une API (URL des ressources, code de statut des réponses, verbes HTTP utilisés, etc.). Par convention, ce fichier est souvent nommé ***swagger.json***.

À l’instar de la méthode de connexion, nous utilisons aussi le paramètre in pour désigner l’identifiant du token. Cet attribut peut valoir :

* path : le paramètre est extrait de l’URL de la ressource ;
* query : le paramètre est un query string ;
* header : le paramètre est une entête HTTP ;
* body : le paramètre est dans le payload ;
* form : le paramètre est dans le payload qui est encodé au format *application/x-www-form-urlencoded* ou *multipart/form-data* (c’est le format utilisé par un formulaire classique).

L’entête HTTP X-Auth-Token est utilisée par plusieurs requêtes de notre API. En le déclarant dans l’attribut parameters, cela nous permet de le réutiliser dans les appels API qui nous intéressent.

* Lors de la livraison d’une trame, elle est reçue dans l’un des cinq états différents qui sont :
* Accepted : indique le succès du traitement du récepteur.
* Rejected : indique l’échec du traitement du message
* Released : indique message non traité

Modified : un message non traité et non modifié

Pour établir une connexion l’une des pairs doit envoyer une frame header contenant sa version de protocole préféré à l’autre pair et l’autre pair (cas d’un serveur il doit répondre avec une version inférieure ou égale à celle du client) doit répondre avec sa version de protocole supporté.

https://www.youtube.com/watch?v=cvu53CnZmGI