# Dev Web

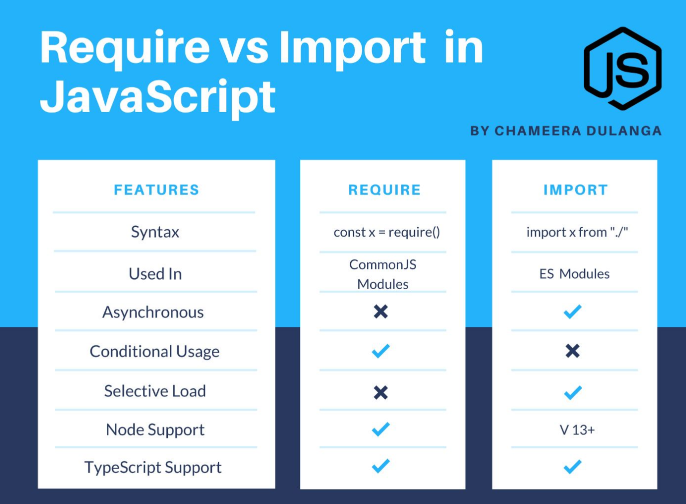
## côté serveur

S3 - R3.01 - 2023/2024

IUT Belfort-Montbéliard

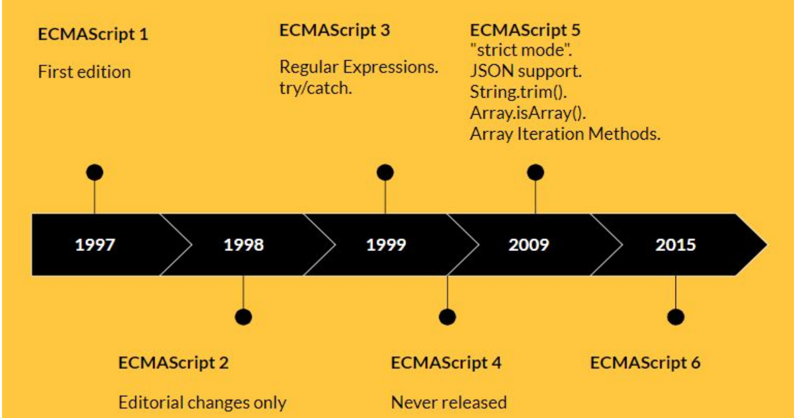
# CommonJS & ESM

- CommonJs est le système de module original et par défaut de Node.js qui utilise require et module.exports.
- Les modules ECMAScript sont relativement récents et utilisent import & export.




| FEATURES | REQUIRE | IMPORT |
|---|---|---|
| Syntax | const x = require() | import x from "./" |
| Used In | CommonJS Modules | ES Modules |
| Asynchronous | ✗ | ✓ |
| Conditional Usage | ✓ | ✗ |
| Selective Load | ✗ | ✓ |
| Node Support | ✓ | V 13+ |
| TypeScript Support | ✓ | ✓ |

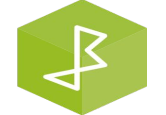
# CommonJS & ESM



| ECMAScript modules | CommonJS modules |
|---|---|
| **Files ending in .js** if there is a top-level field "type" with a value of "module" in the nearest parent package.json file | **Files ending in .js** f there is a top-level field "type" with a value of "commonjs" in the nearest parent package.json file |
| **Files ending in .mjs** | **Files ending in .cjs** |
| **Strings passed in as an argument to --eval or --print, or piped to node via STDIN, with the flag --input-type=module** | **Strings passed in as an argument to --eval or --print, or piped to node via STDIN, with the flag --input-type=commonjs** |
| | **Files ending in .js** if there is no top-level field "type" in the nearest parent package.json file |
| | **Strings passed in as an argument to --eval or --print, or piped to node via STDIN, without the flag --input-type** |

# CommonJS & ESM

- Les modules ECMAScript (également appelés modules ES ou ESM) ont été introduits dans le cadre de la spécification ECMAScript 2015 dans le but de donner à JavaScript un système de modules officiel adapté à différents environnements d'exécution. La spécification ESM essaie de conserver quelques bonnes idées des systèmes de modules existants précédents comme CommonJS. La syntaxe est très simple et compacte. Il est possible de charger des modules de manière asynchrone.
- Le différenciateur le plus important entre ESM et CommonJS est que les modules ES sont statiques, ce qui signifie que les importations sont décrites au niveau supérieur de chaque module et en dehors de toute instruction de flux de contrôle.



# CommonJS & ESM

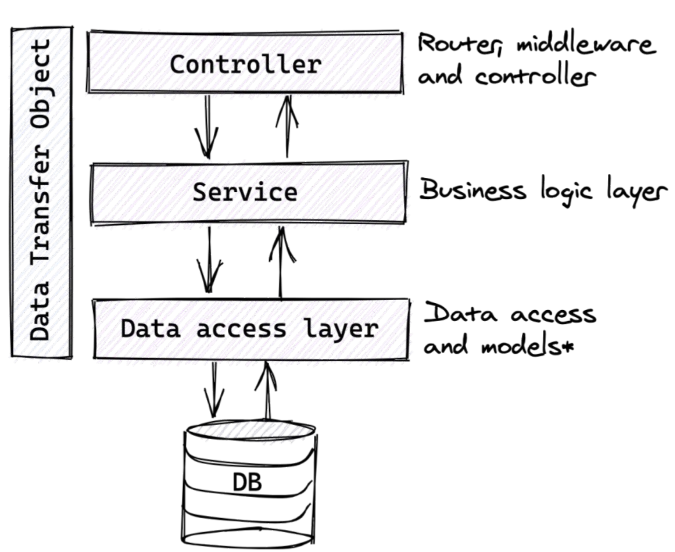
Par exemple, le code suivant ne serait pas valide lors de l'utilisation de modules ES :

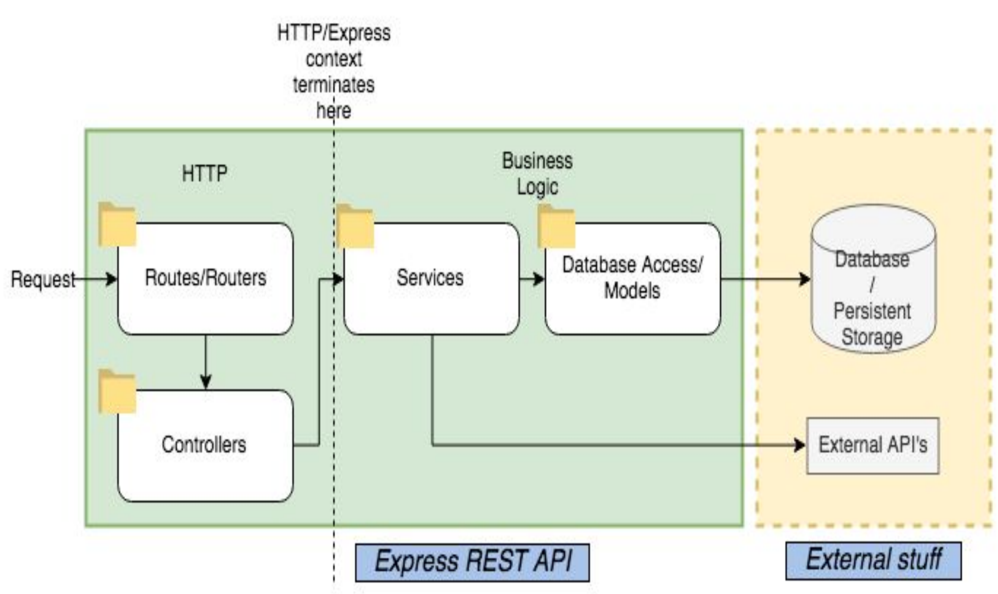
```
if (condition) {
  import module1 from 'module1'
} else {
  import module2 from 'module2'
}
```

Alors que dans CommonJS, il est parfaitement bien d'écrire quelque chose comme ceci :

```
let module = null
if (condition) {
  module = require('module1')
} else {
  module = require('module2')
}
```

# Router - Controller - Service - Model/DAO



# Middlewares

- La fonction **app.use** monte les fonctions middleware. Ceci est un élément important du jargon Express.
- Les fonctions du middleware sont impliquées dans le traitement des requêtes et l'envoi des résultats aux clients HTTP.
- Ils ont accès aux objets de requête et de réponse et sont censés traiter leurs données et éventuellement ajouter des données à ces objets.

```
const express = require('express')
const app = express()

const myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

# Middlewares

```
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

- HTTP method for which the middleware function applies.
- Path (route) for which the middleware function applies.
- The middleware function.
- Callback argument to the middleware function, called "next" by convention.
- HTTP response argument to the middleware function, called "res" by convention.
- HTTP request argument to the middleware function, called "req" by convention.

# Middlewares

- Les fonctions middleware prennent trois arguments. Les deux premiers (request et response) sont équivalents aux objets de requête et de réponse de l'objet HTTP request Node.js.
- Le dernier argument, **next**, est une fonction de rappel qui contrôle la fin du cycle requête-réponse, et il peut être utilisé pour envoyer des erreurs dans le pipeline middleware.

```
const express = require('express')
const app = express()

const myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

# Middlewares

- Les fonctions middleware appellent next et, dans un cas normal, ne fournissent aucun argument en appelant **next()**. S'il y a une erreur, la fonction middleware indique l'erreur en appelant **next(err)**, comme illustré ici.

```
app.use(function(req, res, next) {
  const err = new Error('Not found');
  err.status = 404;
  next(err);
});
```

# Architectures d'API

L'architecture de l'API fait référence à la conception d'une API et à son organisation.

Il englobe la structure d'une API et la façon dont elle est construite, y compris les points de terminaison, les méthodes et les paramètres.

RapidAPI
@Rapid_API

API architecture refers to the design of an API and how it is organized.

It encompasses the structure of an API and how it is built, including the endpoints, methods, and parameters.

6:01 PM · Sep 20, 2022 · FeedHive.io

# Architectures d'API

Il existe différents types d'architectures d'API que les développeurs peuvent utiliser pour créer des API.

Le type d'architecture d'API le plus populaire est l'API REST.

Les autres types d'architectures d'API populaires incluent SOAP, GraphQL, XML-RPC et JSON-RPC.

RapidAPI
@Rapid_API

There are a few different types of API architectures that developers can use to create APIs.

The most popular type of API architecture is the REST API.

Other popular types of API architectures include SOAP, XML-RPC, and JSON-RPC.

6:01 PM · Sep 20, 2022 · FeedHive.io

@Rapid_API

## API Architecture Types

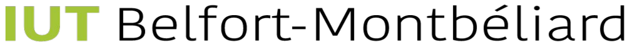
**REST** (Representational State Transfer)
- Follows six REST architectural constraints
- Can use JSON, XML, HTML, or plain text
- Flexible, lightweight, and scalable
- Most used API format on the Web
- Uses HTTP

**GraphQL**
- A query language for APIs
- Uses a Schema to describe data
- Functions using queries and mutations
- Uses a single endpoint to fetch specific data
- Used in apps requiring low bandwidth

**SOAP** (Simple Object Access Protocol)
- Strictly defined messaging framework that relies on XML
- Protocol independent
- Secure and extensible
- Used in secure enterprise environments

**RPC** (Remote procedure Call)
- Action-based procedure great for command-based systems
- Uses only HTTP GET and POST
- Has lightweight payloads that allow for high performance
- Used for distributed systems

**Apache Kafka**
- Used for live event streaming
- Communicates over TCP protocol
- Can publish, store, and process data as it occurs
- Captures and delivers real-time data e.g. stock markets



# Architectures d'API

L'API SOAP fait référence à l'interface de programmation d'application pour la communication via Simple Object Access Protocol.

Il définit les règles et les normes d'interaction entre deux applications sur Internet.

Ce type d'API est souvent utilisé dans les services Web.

**RapidAPI**
@Rapid_API

2. SOAP

SOAP API refers to the application programming interface for communication in Simple Object Access Protocol.

It defines the rules and standards for how two applications can interact with each other over the internet.

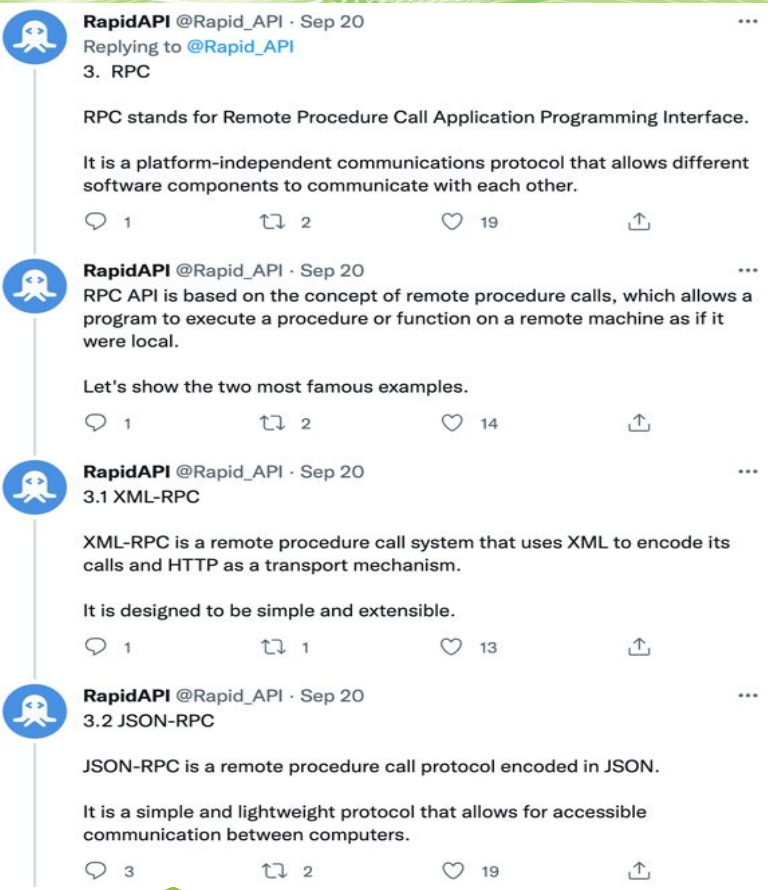
This type of API is often used in web services.

# Architectures d'API

RPC signifie Remote Procedure Call.

Il s'agit d'un protocole de communication indépendant de la plate-forme qui permet à différents composants logiciels de communiquer entre eux.

L'API RPC est basée sur le concept d'appels de procédure à distance, qui permet à un programme d'exécuter une procédure ou une fonction sur une machine distante comme si elle était locale.



# Introduction au REST

- Dans le développement Web, les API REST jouent un rôle important pour assurer une communication fluide entre le client et le serveur.
- La communication entre le client (frontend) et le serveur (backend) n'est généralement pas très directe. Nous utilisons donc une interface appelée Application Programming Interface (ou API) pour servir d'intermédiaire entre le client et le serveur.

**RapidAPI**
@Rapid_API

1. REST

A REST (Restful) API is a web service that uses HTTP requests (GET, PUT, POST, and DELETE) to manipulate data.

A REST API can access resources such as HTML pages, images, and other resources identified by a URL.

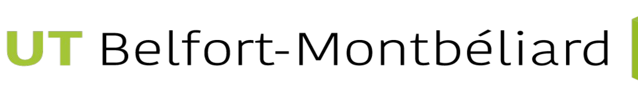
6:01 PM · Sep 20, 2022 · FeedHive.io

# Introduction au REST

- REST est un style architectural logiciel créé par Roy Fielding en 2000 pour guider la conception de l'architecture du Web. Toute API qui suit le principe de conception REST est dite RESTful.
- En termes simples, une API REST est un moyen permettant à deux ordinateurs de communiquer via HTTP (Hypertext Transfer Protocol), de la même manière que les clients et les serveurs communiquent.

REST: Representational State Transfer





# Introduction au REST

**RapidAPI**
@Rapid_API

RESTful APIs are the most famous type of API.

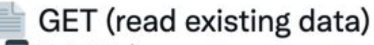
REST APIs are APIs that follow standardized principles, properties, and constraints.

You can access resources in the REST API using HTTP verbs. 👇

11:38 AM · Oct 26, 2021 · FeedHive.io

**RapidAPI**
@Rapid_API

Here are a few common HTTP verbs:



📄 GET (read existing data)
📲 POST (create a new response or data)
♻️ PATCH (update the data)
🗑️ DELETE (delete the data)

11:38 AM · Oct 26, 2021 · FeedHive.io

**RapidAPI**
@Rapid_API

HTTP has a fixed number of methods that the client can use to indicate what type of operation it wants to perform via the request.

- GET
- POST
- PUT
- PATCH
- DELETE
- HEAD
- TRACE
- OPTIONS
- CONNECT

3:20 PM · Dec 3, 2021 · FeedHive.io

# Introduction au REST

Si une méthode HTTP ne modifie pas l'état du serveur en dehors de la journalisation, on l'appelle une méthode “safe”.

Les méthodes GET, HEAD ou OPTIONS sont des méthodes “safe”.

Il est important de noter que toutes les méthodes HTTP “safe” sont également idempotentes, mais toutes les méthodes HTTP idempotentes ne sont pas “safe”.

Lorsque plusieurs appels d'une méthode HTTP donnent le même résultat et laissent le serveur dans le même état, la méthode HTTP est appelée idempotente.

**RapidAPI**
@Rapid_API

If an HTTP method doesn’t alter the server's state apart from logging, it is called a safe method.

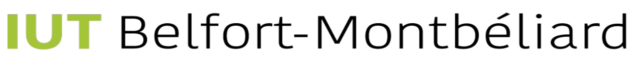
GET, HEAD, or OPTIONS methods are safe methods.

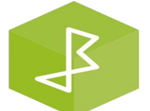
3:20 PM · Dec 3, 2021 · FeedHive.io

**RapidAPI** @Rapid_API · Dec 3, 2021
Replying to @Rapid_API
It is important to note that all the safe HTTP methods are also idempotent, but not all idempotent HTTP methods are safe.

1 6

**RapidAPI** @Rapid_API · Dec 3, 2021
When multiple calls of an HTTP method yield the same result and leave the server in the same state, then the HTTP method is called idempotent.





# Caractéristiques de l'API RESTful

Dans une API RESTful, le client et le serveur sont toujours indépendants, garantissant que le client et le serveur peuvent être mis à l'échelle indépendamment.

An actual RESTful API will follow the following constraints. 👇

1. Client-Server Architecture

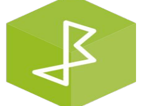
The client requests the data from the server with no third-party interpretation.

11:38 AM · Oct 26, 2021 · FeedHive.io

1️⃣ Client-server Separation

In a RESTful API, the client and server are always kept independent, ensuring that both the client and the server can be scaled independently.

3:21 PM · Dec 3, 2021 · FeedHive.io



# Caractéristiques de l'API RESTful

Dans une API RESTful, le client et le serveur sont toujours indépendants, garantissant que le client et le serveur peuvent être mis à l'échelle indépendamment.

**RapidAPI**
@Rapid_API

An actual RESTful API will follow the following constraints. 👇

1. Client-Server Architecture

The client requests the data from the server with no third-party interpretation.

11:38 AM · Oct 26, 2021 · FeedHive.io

**RapidAPI**
@Rapid_API

1️⃣ Client-server Separation

In a RESTful API, the client and server are always kept independent, ensuring that both the client and the server can be scaled independently.

3:21 PM · Dec 3, 2021 · FeedHive.io

# Caractéristiques de l'API RESTful

Les serveurs ne sont pas autorisés à stocker des données relatives au client. Aucun état de session ou d'authentification n'est stocké sur le serveur.

Statelessness exige que chaque requête du client au serveur contienne toutes les informations nécessaires pour comprendre et compléter la requête.

Le serveur ne peut pas tirer parti des informations de contexte précédemment stockées sur le serveur.

Pour cette raison, l'application cliente doit conserver entièrement l'état de la session.

**RapidAPI**
@Rapid_API

2. Statelessness

Statelessness means that every HTTP request happens in complete isolation. The client and the server don't need to store any information about each other, and there is no state.

11:38 AM · Oct 26, 2021 · FeedHive.io

**RapidAPI**
@Rapid_API

2 Stateless

Servers aren't allowed to store any data related to the client. No session or authentication state is stored on the server.

If the client requires authentication, then the client needs to authenticate itself before sending a request to the server.

3:21 PM · Dec 3, 2021 · FeedHive.io

# Caractéristiques de l'API RESTful

La contrainte "cacheable" exige qu'une réponse s'étiquette implicitement ou explicitement comme cachable ou non cachable.

Si la réponse peut être mise en cache, l'application cliente obtient le droit de réutiliser les données de réponse ultérieurement pour des requêtes équivalentes et une période spécifiée.

RapidAPI
@Rapid_API

3. Cacheability

The response can be cacheable, and it can improve the performance and scalability of an API. Also, cacheability allows the client to get the data even quicker.

11:38 AM · Oct 26, 2021 · FeedHive.io

RapidAPI
@Rapid_API

Responses can be explicitly or implicitly defined as cacheable or non-cacheable to improve scalability and performance.

The main idea of caching is to improve the performance of the client by reducing the bandwidth required to load the resource.

3:21 PM · Dec 3, 2021 · FeedHive.io

# Caractéristiques de l'API RESTful

Le style de système en couches permet à une architecture d'être composée de couches hiérarchiques en contraignant le comportement des composants.

Par exemple, dans un système en couches, chaque composant ne peut pas voir au-delà de la couche immédiate avec laquelle il interagit.

**RapidAPI**
@Rapid_API

4. Layering

Different layers of the API architecture should work together, creating a scalable system that is easy to update or adjust.

11:38 AM · Oct 26, 2021 · FeedHive.io

**2** Retweets **45** Likes

**RapidAPI**
@Rapid_API

4 Layered System

It isn't always necessarily true that the client connects directly to the server and requests a resource. There can be multiple systems in between them that are responsible for handling security, traffic, balancing the load, redirection, etc.

3:21 PM · Dec 3, 2021 · FeedHive.io

**13** Likes

# Caractéristiques de l'API RESTful

Request
Response
Presentation
Component
Client Interactions
Request
Domain Object
Business
Component
Business Processing
Domain Object
Domain Object
Persistence
Component
Data Persistence
Domain Object
Domain Object
Database
Database Storage

# Caractéristiques de l'API RESTful

56

L'utilisation d'interfaces pour dissocier les classes de l'implémentation de leurs dépendances est un concept assez ancien. Dans REST, vous utilisez le même concept pour découpler le client de l'implémentation du service REST. Pour définir une telle interface (un contrat entre le client et le service), il faut utiliser des standards. En effet, si vous voulez un réseau de services REST de la taille d'Internet, vous devez appliquer des concepts globaux, comme des normes, pour qu'ils se comprennent.

- **Identification des ressources** - Vous utilisez la norme URI (IRI) pour identifier une ressource. Dans ce cas, une ressource est un document Web.
- **Manipulation des ressources à travers ces représentations** - Vous utilisez le standard HTTP pour décrire la communication. Ainsi, par exemple, GET signifie que vous souhaitez récupérer des données sur la ressource identifiée par l'URI. Vous pouvez décrire une opération avec une méthode HTTP et un URI.
- **Messages auto-descriptifs** - Vous utilisez des types MIME standard et des vocabulaires RDF (standard) pour rendre les messages auto-descriptifs. Ainsi, le client peut trouver les données en vérifiant la sémantique, et il n'a pas besoin de connaître la structure de données spécifique à l'application utilisée par le service.
- **Hypermédia en tant que moteur de l'état de l'application** (alias HATEOAS) - Vous utilisez des hyperliens et éventuellement des modèles d'URI pour découpler le client de la structure d'URI spécifique à l'application. Vous pouvez annoter ces hyperliens avec la sémantique, par exemple les relations de lien IANA , afin que le client comprenne ce qu'ils signifient.

Partager Améliorer cette réponse
Suivre

modifié le 18 mai 2021 à 10:46
Parzh d'Ukraine
6 777 • 3 • 28 • 57

répondu le 25 septembre 2014 à 23:29
inf3rno
23.3k • dix • 108 • 188

RapidAPI
@Rapid_API

5. Uniform Interface

Communication between the client and the server must be done in a standardized language that is independent of both. This improves scalability and flexibility.

11:38 AM · Oct 26, 2021 · FeedHive.io

1 Retweet 43 Likes

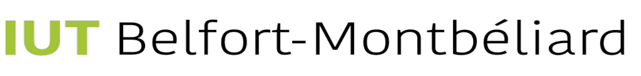
RapidAPI
@Rapid_API

5 Uniform Interface

The client and server can interact with each other in a single language irrespective of the architecture that they are based upon.

3:21 PM · Dec 3, 2021 · FeedHive.io

12 Likes



# Caractéristiques de l'API RESTful

## REST — Uniform interface

- Identification of **resources** → Resource as URIs http://api.co/cars/123
- Manipulation of resources through **representations** → JSON, XML…
- Self-descriptive messages → HTTP GET, POST, PUT, DELETE media types, cacheability…
- **HATEOAS** *(Hypermedia As The Engine Of Application State)* → Hypermedia APIs HAL, JSON-LD, Siren…

# Caractéristiques de l'API RESTful

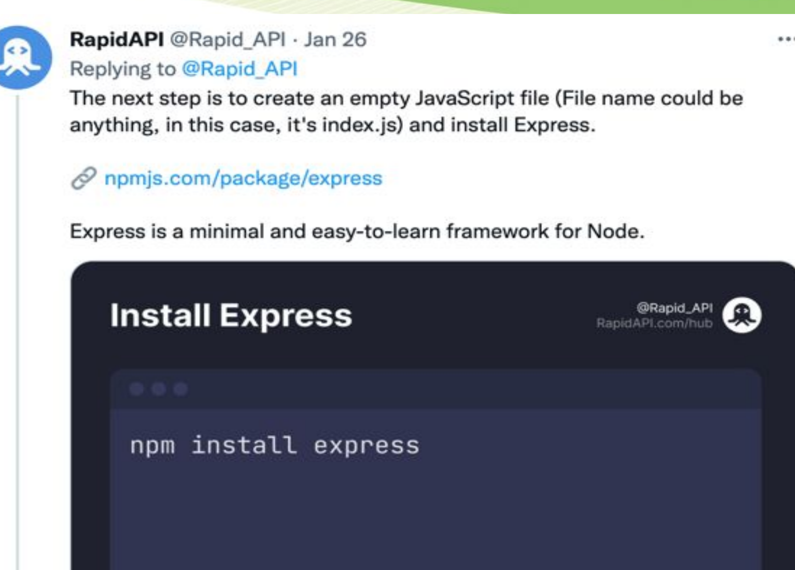
**What is HATEOAS?**

- Hypermedia As The Engine Of Application State
- The client doesn't have a built in knowledge of how to navigate and manipulate the model
- Instead server provides that information dynamically to the user
- Implemented by using media types and link relations

```
{
    "FirstName": "Razvan",
    "LastName": "Stetcu",
    "BirthDate": "07-08-1994",
    "Links": [
        {
            "Href": "http://localhost:62782/Employee/4",
            "Rel": "get_employee",
            "Method": "GET"
        },
        {
            "Href": "http://localhost:62782/Employee/4",
            "Rel": "delete_employee",
            "Method": "DELETE"
        },
        {
            "Href": "http://localhost:62782/Employee",
            "Rel": "edit_employee",
            "Method": "PUT"
        }
    ]
}
```

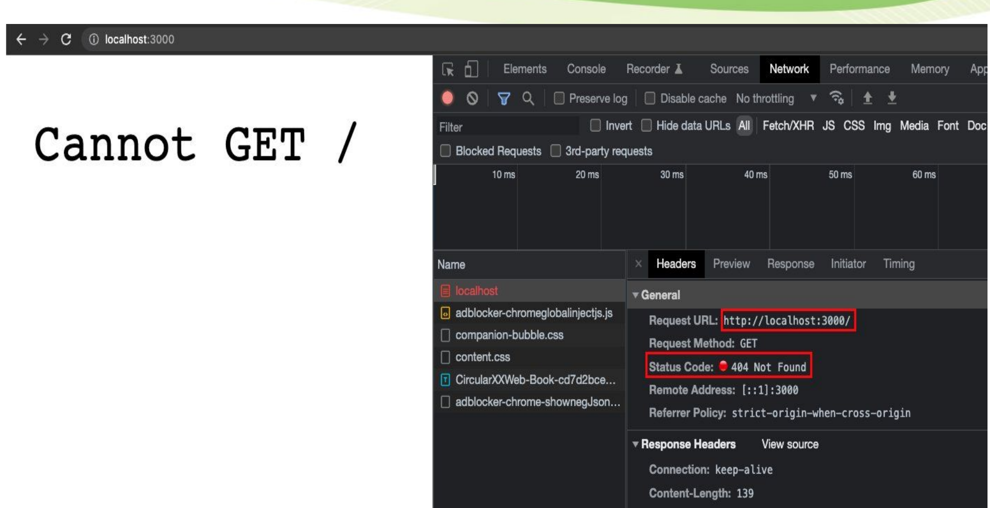
```
{
    "links": {
        "self": { "href": "http://api.com/items" },
        "item": [
            { "href": "http://api.com/items/1" },
            { "href": "http://api.com/items/2" }
        ]
    "data": [
            {"itemName":"a"},
            {"itemName":"b"}
     ]
}
```

# Construire une API REST à partir de zéro en utilisant Node.js et Express



# Construire une API REST à partir de zéro en utilisant Node.js et Express

Si nous essayons d'accéder à "localhost:3000" dans le navigateur, nous voyons que nous obtenons une réponse "404 Not Found" qui est correcte car nous n'avons pas encore défini de points de terminaison.



# Construire une API REST à partir de zéro en utilisant Node.js et Express

# Construire une API REST à partir de zéro en utilisant Node.js et Express

RapidAPI
@Rapid_API

The `get` method allows us to create HTTP GET requests.

It accepts two params:

- The first is path/route.

- The second is a callback function that handles the request to the specified route.

8:37 PM · Jan 26, 2022 · FeedHive.io

RapidAPI
@Rapid_API

The callback function itself accepts two arguments:

- The first is the request which is the data to the server.

- The second is the response which is the data to the client.

**get method** — @Rapid_API RapidAPI.com/hub

```
app.get('/path', (req, res) ⇒ {

});
```

ALT

8:37 PM · Jan 26, 2022 · FeedHive.io

RapidAPI
@Rapid_API

Suppose you want to display all the users whenever the client requests the "/users" endpoint.

To return the data from the server to the client, we have the `send` method.

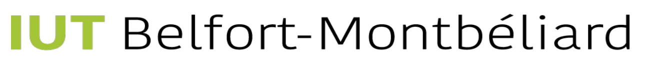
In the code snippet below, we send an array of objects with `name` and `id` fields.

**GET Request** — @Rapid_API RapidAPI.com/hub

```
app.get('/users', (req, res) ⇒ {
  res.status(200).send([
    { name: 'John', id: 1 },
    { name: 'Doe', id: 2 },
  ]);
});
```

ALT

8:37 PM · Jan 26, 2022 · FeedHive.io



# Construire une API REST à partir de zéro en utilisant Node.js et Express

RapidAPI
@Rapid_API

As POST request is to create or add new data to the server. We first need to add middleware so that we can parse JSON body.

**Add middleware**

@Rapid_API
RapidAPI.com/hub

```
const express = require("express");
const app = express();
app.use(express.json());
```

ALT

8:37 PM · Jan 26, 2022 · FeedHive.io

RapidAPI
@Rapid_API

We are defining a POST endpoint at the "/user/3" route.

We implemented the logic of throwing a "400 Bad Request" status code if the user forgets to pass the name value in the request body. 👇

**POST Endpoint**

@Rapid_API
RapidAPI.com/hub

```
app.post("/user/3", (req, res) => {
  const { name } = req.body;
  if (!name) {
    res.status(400).send({ message: "Please add name of the user" });
  }
  res.send({ name: name, id: 3 });
});
```

ALT

8:37 PM · Jan 26, 2022 · FeedHive.io

# Meilleures pratiques lors de la création de l'API REST

## Acceptez et répondez avec JSON

Dans le passé, l'acceptation des / et la réponse aux requêtes API se faisaient principalement en XML et même en HTML. Mais de nos jours, JSON (JavaScript Object Notation) est largement devenu le format de facto pour l'envoi et la réception de données API.

En effet, avec XML par exemple, le décodage et l'encodage des données sont souvent un peu compliqués. XML n'est donc plus largement pris en charge par les frameworks.

JavaScript, par exemple, a une méthode intégrée pour analyser les données JSON via l'API fetch, car JSON a été principalement conçu pour cela. Mais si vous utilisez un autre langage de programmation tel que Python ou PHP, ils disposent désormais tous de méthodes pour analyser et manipuler également les données JSON.

# Meilleures pratiques lors de la création de l'API REST

## Acceptez et répondez avec JSON

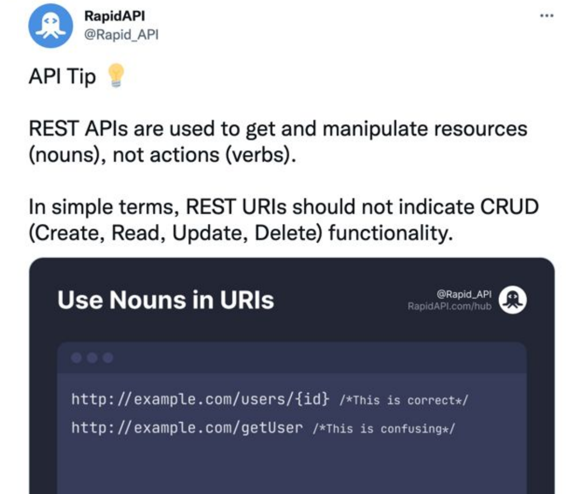
Pour nous assurer que lorsque notre application API REST répond avec JSON, les clients l'interprètent comme tel, nous devons définir **Content-Type** dans l'en-tête de réponse sur **application/json** une fois la requête effectuée. De nombreux frameworks d'applications côté serveur définissent automatiquement l'en-tête de réponse. Certains clients HTTP examinent l'en-tête de réponse **Content-Type** et analysent les données en fonction de ce format.

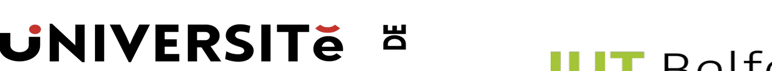
```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.post('/', (req, res) => {
  res.json(req.body);
});

app.listen(3000, () => console.log('server started'));
```

# Meilleures pratiques lors de la création de l'API REST

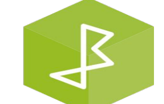
## Utiliser des noms au lieu de verbes dans les chemins de point de terminaison

Lorsque vous concevez une API REST, vous ne devez pas utiliser de verbes dans les chemins de point de terminaison. Les points de terminaison doivent utiliser des noms, signifiant ce que chacun d'eux fait. En effet, les méthodes HTTP telles que GET, POST, PUT, PATCH et DELETE sont déjà sous forme verbale pour effectuer des opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) de base.





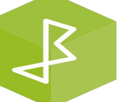



# Meilleures pratiques lors de la création de l'API REST

## Utiliser des noms au lieu de verbes dans les chemins de point de terminaison

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.get('/articles', (req, res) => {
  const articles = [];
  // code to retrieve an article...
  res.json(articles);
});

app.post('/articles', (req, res) => {
  // code to add a new article...
  res.json(req.body);
});
```

```
app.put('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to update an article...
  res.json(req.body);
});

app.delete('/articles/:id', (req, res) => {
  const { id } = req.params;
  // code to delete an article...
  res.json({ deleted: id });
});

app.listen(3000, () => console.log('server started'));
```



# Meilleures pratiques lors de la création de l'API REST

## Nommez les collections avec des noms au pluriel

Vous pouvez considérer les données de votre API comme une collection de différentes ressources de vos consommateurs.

Si vous avez un point de terminaison comme https://mysite.com/post/123, il peut être correct de supprimer un “post” avec une requête DELETE ou de mettre à jour un “post” avec une requête PUT ou PATCH, mais cela ne dit pas à l'utilisateur qu'il y a pourrait être d'autres "posts" dans la collection. C'est pourquoi vos collections doivent utiliser des noms au pluriel.

Ainsi, au lieu de https://mysite.com/post/123, il devrait s'agir de https://mysite.com/posts/123.

# Meilleures pratiques lors de la création de l'API REST

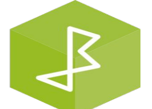
## Utiliser l'imbrication logique sur les terminaux

Lors de la conception des endpoints, il est logique de regrouper ceux qui contiennent des informations associées.

Autrement dit, si un objet peut contenir un autre objet, vous devez concevoir le point de terminaison pour refléter cela.

Il s'agit d'une bonne pratique, que vos données soient structurées de cette manière dans votre base de données ou non. En fait, il peut être conseillé d'éviter de refléter la structure de votre base de données dans vos points de terminaison pour éviter de donner aux attaquants des informations inutiles.

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.get('/articles/:articleId/comments', (req, res) => {
  const { articleId } = req.params;
  const comments = [];
  // code to get comments by articleId
  res.json(comments);
});


app.listen(3000, () => console.log('server started'))
```



# Meilleures pratiques lors de la création de l'API REST

## Utiliser l'imbrication logique sur les terminaux

Souvent, différents points de terminaison peuvent être liés entre eux, vous devez donc les imbriquer pour qu'il soit plus facile de les comprendre.

Par exemple, dans le cas d'une plate-forme de blogs multi-utilisateurs, différents articles pourraient être écrits par différents auteurs, donc un point de terminaison tel que https://mysite.com/posts/author ferait une imbrication valide dans ce cas.

Dans le même esprit, les publications/posts peuvent avoir leurs commentaires individuels, donc pour récupérer les commentaires, un point de terminaison comme https://mysite.com/posts/postId/comments aurait du sens.

# Meilleures pratiques lors de la création de l'API REST

## Utiliser l'imbrication logique sur les terminaux

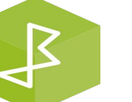
Cependant, l'imbrication peut aller trop loin. Après environ le deuxième ou le troisième niveau, les points de terminaison imbriqués peuvent devenir difficiles à manier. Envisagez plutôt de renvoyer l'URL à ces ressources, en particulier si ces données ne sont pas nécessairement contenues dans l'objet de niveau supérieur.

Vous devez éviter une imbrication de plus de 3 niveaux car cela peut rendre l'API moins élégante et moins lisible.

Par exemple, supposons que vous vouliez renvoyer l'auteur de commentaires particuliers. Vous pouvez utiliser **/articles/:articleId/comments/:commentId/author**. Mais cela devient incontrôlable. Au lieu de cela, renvoyez plutôt l'URI de cet utilisateur particulier dans la réponse JSON :

**"author": "/users/:userId"**

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.get('/articles/:articleId/comments', (req, res) => {
  const { articleId } = req.params;
  const comments = [];
  // code to get comments by articleId
  res.json(comments);
});


app.listen(3000, () => console.log('server started'))
```



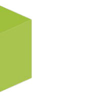
# Meilleures pratiques lors de la création de l'API REST

## Utiliser les codes d'état (status codes) dans la gestion des erreurs

Vous devez toujours utiliser des codes d'état HTTP normaux dans les réponses aux requêtes adressées à votre API. Cela aidera vos utilisateurs à savoir ce qui se passe - si la requête est réussie, si elle échoue, ou autre chose

Vous trouverez ci-dessous un tableau présentant différentes plages de codes d'état HTTP et leur signification:

| STATUS CODE RANGE | MEANING |
|---|---|
| 100 – 199 | Informational Responses.<br>For example, 102 indicates the resource is being processed |
| 300 – 399 | Redirects<br>For example, 301 means Moved permanently |
| 400 – 499 | Client-side errors<br>400 means bad request and 404 means resource not found |
| 500 – 599 | Server-side errors<br>For example, 500 means an internal server error |



# Meilleures pratiques lors de la création de l'API REST

## Utiliser les codes d'état (status codes) dans la gestion des erreurs

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// existing users
const users = [
  { email: 'abc@foo.com' }
]

app.use(bodyParser.json());

app.post('/users', (req, res) => {
  const { email } = req.body;
  const userExists = users.find(u => u.email === email);
  if (userExists) {
    return res.status(400).json({ error: 'User already exists' })
  }
  res.json(req.body);
});


app.listen(3000, () => console.log('server started'));
```

# Meilleures pratiques lors de la création de l'API REST

## Autoriser le filtrage, le tri et la pagination

Parfois, la base de données d'une API peut devenir incroyablement volumineuse. Si cela se produit, la récupération des données d'une telle base de données peut être très lente.

Le filtrage, le tri et la pagination sont autant d'actions qui peuvent être effectuées sur la collection d'une API REST. Cela lui permet uniquement de récupérer, de trier et d'organiser les données nécessaires afin que le serveur ne soit pas trop occupé par les requêtes.

Un exemple de point de terminaison filtré est celui ci-dessous:

https://mysite.com/posts?tags=javascript

Ce point de terminaison récupère tout message contenant un “tag” javascript.

# Meilleures pratiques lors de la création de l'API REST

## Autoriser le filtrage, le tri et la pagination

Nous avons également besoin de moyens de paginer les données afin de ne renvoyer que quelques résultats à la fois. Nous ne voulons pas immobiliser les ressources trop longtemps en essayant d'obtenir toutes les données demandées en même temps.

Le filtrage et la pagination augmentent tous deux les performances en réduisant l'utilisation des ressources du serveur. Plus les données s'accumulent dans la base de données, plus ces fonctionnalités deviennent importantes.

`/employees?lastName=Smith&age=30`

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// employees data in a database
const employees = [
  { firstName: 'Jane', lastName: 'Smith', age: 20 },
  //...
  { firstName: 'John', lastName: 'Smith', age: 30 },
  { firstName: 'Mary', lastName: 'Green', age: 50 },
]

app.use(bodyParser.json());

app.get('/employees', (req, res) => {
  const { firstName, lastName, age } = req.query;
  let results = [...employees];
  if (firstName) {
    results = results.filter(r => r.firstName === firstName);
  }

  if (lastName) {
    results = results.filter(r => r.lastName === lastName);
  }

  if (age) {
    results = results.filter(r => +r.age === +age);
  }
  res.json(results);
});

app.listen(3000, () => console.log('server started'));
```

# Meilleures pratiques lors de la création de l'API REST

## Autoriser le filtrage, le tri et la pagination

De même, nous pouvons accepter le paramètre de requête “page” et renvoyer un groupe d'entrées dans la position de (page - 1) * 10 à page * 10.

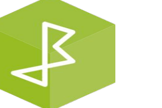
Nous pouvons également spécifier les champs à trier dans la chaîne de requête. Par exemple, nous pouvons obtenir le paramètre d'une chaîne de requête (queryString) avec les champs pour lesquels nous voulons trier les données. Ensuite, nous pouvons les trier par ces champs individuels.

Par exemple, nous pouvons souhaiter extraire la chaîne de requête d'une URL telle que :

http://example.com/articles?sort=+author,-datepublished

Où **+ signifie ascendant** et **- signifie descendant**. Nous trions donc par nom d'auteur par ordre alphabétique et date de publication du plus récent au moins récent.





# Meilleures pratiques lors de la création de l'API REST

## Utiliser SSL pour la sécurité

SSL signifie Secure Socket Layer. C'est crucial pour la sécurité dans la conception de l'API REST. Cela sécurisera votre API et la rendra moins vulnérable aux attaques malveillantes.

D'autres mesures de sécurité que vous devriez prendre en considération incluent: rendre la communication entre le serveur et le client privée et s'assurer que toute personne utilisant l'API n'obtient pas plus que ce qu'elle demande.

Les certificats SSL ne sont pas difficiles à charger sur un serveur et sont disponibles gratuitement principalement pendant la première année. Ils ne sont pas chers à l'achat dans les cas où ils ne sont pas disponibles gratuitement.

La différence claire entre l'URL d'une API REST qui s'exécute sur SSL et celle qui ne l'est pas est le « s » en HTTP:

https://mysite.com/posts fonctionne sur SSL

http://mysite.com/posts ne fonctionne pas sur SSL

# Meilleures pratiques lors de la création de l'API REST

## Soyez clair avec la gestion des versions

Les API REST doivent avoir des versions différentes, vous ne forcez donc pas les clients (utilisateurs) à migrer vers de nouvelles versions. Cela peut même être négatif pour l'application si vous ne faites pas attention

L'un des systèmes de gestion de versions les plus courants dans le développement Web est la gestion de versions sémantique

Un exemple de versionnage sémantique est 1.0.0, 2.1.2 et 3.3.4. Le premier nombre représente la version majeure, le deuxième nombre représente la version mineure et le troisième représente la version du correctif

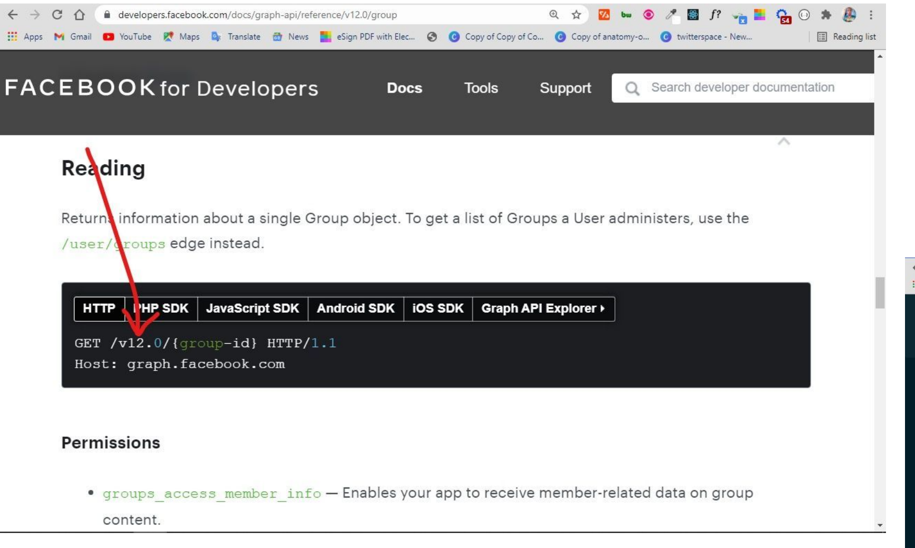
De nombreuses API RESTful de géants de la technologie et d'individus se présentent généralement comme suit :

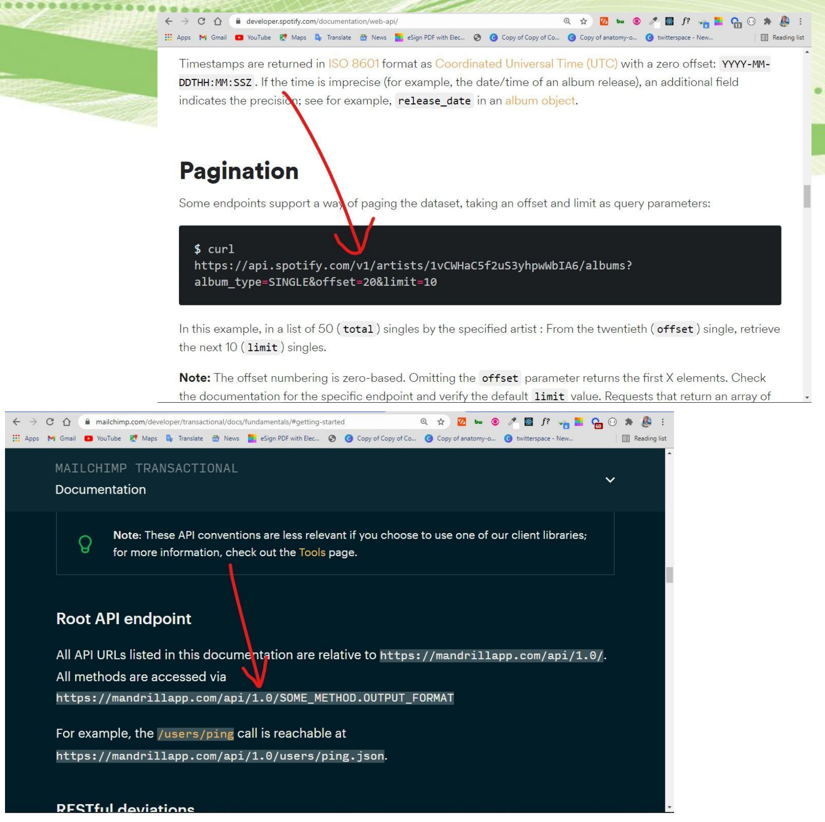
https://mysite.com/v1/ pour la version 1

https://mysite.com/v2/ pour la version 2

# Meilleures pratiques lors de la création de l'API REST

## Soyez clair avec la gestion des versions



# Meilleures pratiques lors de la création de l'API REST

## Fournir une documentation API précise

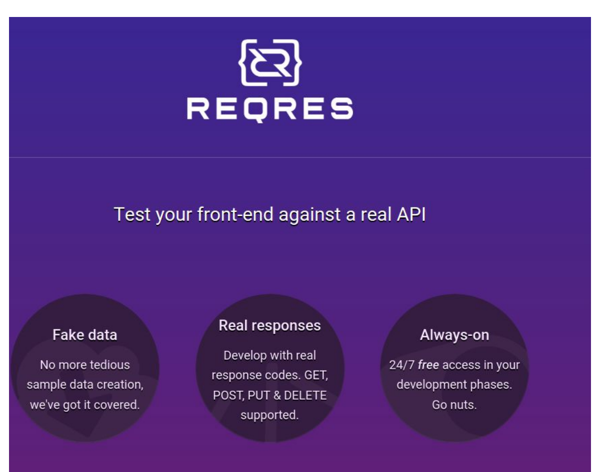
Lorsque vous créez une API REST, vous devez aider les clients (consommateurs) à apprendre et à comprendre comment l'utiliser correctement. La meilleure façon de le faire est de fournir une bonne documentation pour l'API

La documentation doit contenir:

- points de terminaison pertinents de l'API
- exemples de requêtes des points de terminaison
- implémentation dans plusieurs langages de programmation
- messages pour différentes erreurs avec leurs codes de statut
- l'un des outils les plus courants que vous pouvez utiliser pour la documentation de l'API est Swagger

# Meilleures pratiques lors de la création de l'API REST

## Fournir une documentation API précise




| Method | Endpoint |
|---|---|
| GET | LIST USERS |
| GET | SINGLE USER |
| GET | SINGLE USER NOT FOUND |
| GET | LIST <RESOURCE> |
| GET | SINGLE <RESOURCE> |
| GET | SINGLE <RESOURCE> NOT FOUND |
| POST | CREATE |
| PUT | UPDATE |
| PATCH | UPDATE |
| DELETE | DELETE |
| POST | REGISTER - SUCCESSFUL |
| POST | REGISTER - UNSUCCESSFUL |
| POST | LOGIN - SUCCESSFUL |
| POST | LOGIN - UNSUCCESSFUL |
| GET | DELAYED RESPONSE |

**Request**
/api/users?page=2

**Response**
200


```
{
    "page": 2,
    "per_page": 6,
    "total": 12,
    "total_pages": 2,
    "data": [
        {
            "id": 7,
            "email": "michael.lawson@reqr
            "first_name": "Michael",
            "last_name": "Lawson",
            "avatar": "https://reqres.in/
        },
        {
            "id": 8,
            "email": "lindsay.ferguson@re
            "first_name": "Lindsay",
            "last_name": "Ferguson",
            "avatar": "https://reqres.in/
        },
        {
            "id": 9,
            "email": "tobias.funke@reqres
            "first_name": "Tobias",
            "last_name": "Funke",
            "avatar": "https://reqres.in/
        },
        {
            "id": 10,
            "email": "byron.fields@reqres
            "first_name": "Byron",
            "last_name": "Fields",
            "avatar": "https://reqres.in/
        },
        {
            "id": 11,
            "email": "george.edwards@reqr
            "first_name": "George",
            "last_name": "Edwards",
            "avatar": "https://reqres.in/
```

# Meilleures pratiques lors de la création de l'API REST

## Choisissez le bon framework pour votre API REST Node.js

Il est important de choisir le framework qui convient le mieux à votre cas d'utilisation.

### Express, Koa ou Hapi

Express, Koa et Hapi peuvent être utilisés pour créer des applications de navigateur et, à ce titre, ils prennent en charge la création de modèles et le rendu, pour ne citer que quelques fonctionnalités. Si votre application doit également fournir le côté utilisateur, il est logique de les utiliser.

### Restify

D'autre part, Restify se concentre sur vous aider à créer des services REST. Il existe pour vous permettre de créer des services API "stricts" qui sont maintenables et observables. Restify est utilisé en production dans des applications majeures comme npm ou Netflix.

# Meilleures pratiques lors de la création de l'API REST

## Utiliser les méthodes HTTP et les routes API

Les informations de la méthode doivent être exprimées dans le verbe HTTP.

| | |
|---|---|
| Not RESTful | **GET** api/users/delete/:userId HTTP/1.1 |
| RESTful | **DELETE** api/users/:userId HTTP/1.1 |