Node Api

Louis Harang - Wizards Technologies

Sommaire du cours

Les API

- C’est quoi une API ?
- HTTP(s), méthodes et codes
- Les headers
- Requête et réponse
- Formats, sérialisation et désérialisation
- Les clients API

Node JS

- Historique
- Moteur V8 et javascript server side
- Installation & bases de node
- Common JS vs ESM
- Packages
- Les promesses

Express JS

- Présentation & installation
- Créer sa première application express
- Le routeur
- Récupérer le payload d’une requête
- L’objet response

Sequelize :

- Rappel sur les ORM
- Installation et configuration
- Définition des modèles
- sequelize-cli
- Associations
- Manipuler la donnée

Express JS avancé

- Sérialisation, désérialisation
- Validation des données
- Dotenv
- Gestion des erreurs
- Organiser son code
- Middleware
- Authentification

Open api

- Présentation
- Exemples

Semantic Versionning

Les API

C’est quoi une API ?

Application Programming Interface
ou
Interface de programmation *en français*

Définition :

Ensemble normalisé de classes, de méthodes, de fonctions et de constantes qui sert de **façade** par laquelle un logiciel offre des services à d'autres logiciels.
Une API est servie un service web ou une bibliothèque logicielle, le plus souvent accompagnée d'une description qui spécifie comment des programmes consommateurs peuvent se servir des fonctionnalités du programme fournisseur.

De manière plus générale, on parle d'API à partir du moment où une entité informatique cherche à agir avec ou sur un système tiers, et que cette interaction se fait de manière normalisée en respectant les contraintes d'accès définies par le système tiers.

HTTP(s), méthodes et codes

HTTP & HTTPs

HTTP : L'Hypertext Transfer Protocol (HTTP, littéralement « protocole de transfert hypertexte ») est un protocole de communication client-serveur développé pour le World Wide Web.

HTTPS : L'Hypertext Transfer Protocol Secure (HTTPS, littéralement « protocole de transfert hypertexte sécurisé ») est la combinaison du HTTP avec une couche de chiffrement comme SSL ou TLS.

- permet au visiteur de vérifier l'identité du site web, via un certificat d'authentification émis par une autorité tierce, réputée fiable.
- Il garantit **théoriquement** la confidentialité et l'intégrité des données envoyées par l'utilisateur.
- Initialement utilisé pour les transactions financières en ligne : commerce électronique, banque en ligne, courtage en ligne, etc. Il est aussi utilisé pour la consultation de données privées, comme les courriers électroniques, par exemple.

A savoir : Le SSL est un critère SEO pour Google

HTTP(s), méthodes et codes

Les endpoints

Un endpoint c'est quoi ? C'est tout simplement la route sur laquelle vous allez appeler l'API ainsi que la méthode HTTP utilisée.

Exemple :

POST /articles

GET /users

PATCH /users/15f2b2fc-7b84-49a0-9fb7-b4b37bcd0e4b

DELETE /articles/62c1d5ae-bcfd-4669-a1ab-3fdd49334951

HTTP(s), méthodes et codes

Les méthodes HTTP

HTTP a plusieurs verbes (HTTP request methods), les plus courants sont :

GET : pour récupérer une ressource

POST : pour créer une ressource

PATCH : pour mettre à jour partiellement une ressource

PUT : pour mettre à jour totalement (remplacer) une ressource

DELETE : pour détruire une ressource

HTTP(s), méthodes et codes

Les codes HTTP

HTTP possède aussi de nombreux codes :

200: OK
201: CREATED
204: EMPTY
301: MOVED PERMANENTLY
302: FOUND
400: BAD REQUEST
401: UNAUTHORIZED
403: FORBIDDEN
404: NOT FOUND
405: METHOD NOT ALLOWED
500: INTERNAL ERROR
502: BAD GATEAWAY
503: SERVICE UNAVAILABLE

Il faut toujours faire attention comment on choisit ses codes de HTTP car cela permet de facilement identifier si notre appel API a réussi ou si il y a eu un problème.
Renvoyer 200 alors qu'il y a une erreur = **très mauvaise pratique**

Requête et réponse

Requête

Lorsque vous voulez appeler une api, vous faites une **requête HTTP** via votre application serveur ou client.

Une requête se compose de plusieurs informations :

- d'une méthode HTTP
- d'un endpoint
- d'un ou plusieurs headers
- d'un payload (ou body)

Le payload est tout simplement un objet JSON (en général) composée de donnée que vous voulez envoyer à l'API. Par exemple, le payload d'une requête lors de l'inscription sur un site :

```
{
  "firstname": "Louis",
  "lastname": "Harang",
  "email": "cours@narah.io",
  "password": "password"
}
```

Requête et réponse

Réponse

La réponse HTTP est tout simplement ce que va répondre l'API.

- d'un code HTTP indiquant le statut de la réponse
- d'un ou plusieurs headers
- d'un body

Le body est peut être directement récupéré par l'application appelante et désérialiser pour pouvoir être manipulé.

Les headers

Définition

Les headers (en-têtes) HTTP permettent au client et au serveur de transmettre des informations supplémentaires avec la requête ou la réponse.

Il existe plusieurs types d'en-têtes avec différentes utilisations :

- Authentification
- Mise en cache
- Conditionnels
- Gestion de connexion
- Négociation de contenu
- Cookies
- Cross-Origin Resource Sharing (CORS)

En utilisant un framework, les headers essentiels sont générés automatiquement, rien besoin de faire. Vous pouvez bien sûr en rajouter ou modifier ceux générés selon vos besoins.

Les headers

Content-Type

Selon la configuration de votre API, il est possible qu'elle accepte plusieurs format.

Pour vous assurez que vous appelez l'API dans le bon format il y a un header à connaître par coeur et à préciser à chaque fois : **Content-Type**

Dans chaque requête vous devrez préciser le type de contenu que vous envoyez / que vous voulez en réponse.

Il existe beaucoup de valeur possible pour **Content-Type**, pour du JSON on utilise *application/json*

Formats, sérialisation et désérialisation

Les formats de données 1/5

Les APIs sont donc *languages agnostics*, qu'elle soit en PHP, Javascript, Go ou Ruby, une API renverra toujours un seul format précis qui sera désérialisé par l'application appelante.

Ils existent plusieurs formats : **JSON, FormData, Binaire** (Messagepack), **XML**

On utilise en règle général le format JSON (JavaScript Object Notation), la structure de ce format est quasiment identique aux objets javascripts et permet d'être facilement désérialiser par l'application appelante.

Un document JSON comprend 2 types composés :

- des objets
- des tableaux
-

et 4 types scalaires :

- des booléens
- des nombres
- des chaînes de caractères
- la valeur null

Formats, sérialisation et désérialisation

Les formats de données 2/5

```
{
    "data": {
        "id": "15f2b2fc-7b84-49a0-9fb7-b4b37bcd0e4b",
        "firstname": "Louis",
        "lastname": "Harang",
        "age": 26,
        "skills": [
          "javascript",
          "php",
          "golang",
          "ruby"
        ],
        "active": true,
        "bio": null,

    }
}
```

Un exemple de document JSON

Formats, sérialisation et désérialisation

Les formats de données 3/5

Au delà des formats de données, il y a la manière de structurer les APIs. Il y a de nombreux standards, les plus connus étant **SOAP** (ancien et contraignant), **REST**, **RPC** ou encore **GraphQL**. Nous allons nous concentrer principalement sur **RPC & REST** pendant ce cours.

RPC : *remote procedure call,* c'est un peu la version alpha des APIs, permet d'exécuter une fonction sur un serveur web. Par exemple je veux enregistrer un email à la newsletter de mon site web, j'enverrais une requête **POST** sur **/newsletter/register** avec dans le corps de ma requête l'e-mail que je souhaite inscrire. Côté API, je vais récupérer l'email dans la requête et faire les traitements classiques par la suite. Pour des applications avec un besoin plus gros (beaucoup de table en base de donnée par exemple), on se tournera plus vers le **REST**.

Formats, sérialisation et désérialisation

Les formats de données 4/5

REST : *representational state transfer,* un service web RESTful permet aux applications appelantes d'effectuer des requêtes de manipulation de **ressource** via leurs représentations textuelles (principalement des opérations de CRUD). Une API REST suivra toujours la même trame. Chaque action de CRUD suivra la même logique. Prenons l'exemple d'un article sur un blog.

GET /articles : on récupère tous les articles
GET /articles/{id} : on récupère un article identifié via son id
POST /articles : on créer un nouvel article
DELETE /articles/{id} : on supprime un article identifié via son id

Pour mettre à jour il y a plusieurs écoles :

PUT /articles/{id} : on écrase l'article identifié via son id
PATCH /articles/{id} : on met à jour uniquement les données de la requête d'un article identifié via son id

Une API **REST** suit en règle général ce format de route, il y a parfois des exceptions comme par exemple pour les relations entre les ressources.

Formats, sérialisation et désérialisation

Les formats de données 5/5

Bien sûr pour la création d'application web il y a toujours une partie de conception et d'architecture à prévoir, mais très souvent le format **REST** est utilisé pour des raisons pratiques (90% du web = des formulaires).

Cependant le **REST** peut parfois sembler beaucoup trop rigide, il est donc recommandé d'utiliser en plus des appels **RPC** permettant d'avoir une plus grande liberté au niveau de la création de vos **APIs.**

Formats, sérialisation et désérialisation

Sérialisation et désérialisation

La sérialisation et désérialisation sont deux processus liés :

Sérialisation : Codage d'une information sous la forme d'une suite d'informations plus petites. Par exemple quand on va persister en base de donnée notre modèle, les données vont être sérialisées. Il en va de même pour les APIs, avant de retourner la donnée à l'application appelante, on sérialise la donnée (un tableau par exemple) au format défini dans l'API (JSON en général)

Désérialisation : C'est le processus inverse de la sérialisation, on prend de la donnée dans un certain format pour la transformer en donnée manipulable dans notre application. Par exemple, transformer un tableau JSON en tableau JS (si on est dans une API en javascript).

Dans la plupart des cas, on utilise des **frameworks** qui font tout ce travail en amont et en aval de chaque appel et réponse de l'API.

Les clients API

Présentation

POSTMAN

Les clients API

Se faciliter la vie avec les collections APIs

Ici nous allons parler du cas de **postman,** leader des clients APIs.

Un point critique sur tous les projets API est **la spécification** (qu'on verra plus tard avec Open API).

La spécification permet de donner la description tous les endpoints de votre API, la méthode pour chaque endpoints, le format de requête ou encore le format de réponse.

Dans ce même esprit, postman vous permet de créer des collections API que vous pouvez exporter / importer. Ces collections vous permettent de sauvegarder et d'organiser vos requêtes postman facilement vous facilitant ainsi le développement de vos **APIs.**

TP

Grâce à l'API swapi et Postman :

1. Trouver l'endpoint pour afficher les informations du vaisseau avec l'id 10
2. Trouver le total de planètes (uniquement la valeur)
3. L'année de naissance de Dark Vador (valeur et endpoint)
4. Le personnage avec l'id 13 traduit en wookiee (endpoint uniquement)
5. Les urls des résidents de la planètes d'origine de R2-D2
6. Faire une collection postman de tous les endpoints ci-dessus

https://swapi.py4e.com/documentation

Node JS

Historique

Node est créé par **Ryan Dahl** en **2009**.
Dahl eu l'idée de créer Node.js après avoir observé la barre de progression d'un chargement de fichier sous
Flickr : le navigateur ne savait pas quel pourcentage du fichier était chargé et devait adresser une requête au serveur web. Dahl voulait développer une méthode plus simple.
Node est basé sur le **moteur V8** de Google.
En 2010, un package manager pour node est apparu : **npm**

Ryan Dahl, créateur de Node.js

Moteur V8 et javascript server side

Moteur V8

V8 est un **moteur JavaScript open-source** développé par le projet Chromium pour les navigateurs Web Google Chrome et Chromium. La première version du moteur V8 a été publiée en même temps que la première version de Chrome, le 2 septembre 2008.

V8 compile directement le code JavaScript en code machine natif avant de l'exécuter. Le code compilé est en outre optimisé (et ré-optimisé) dynamiquement au moment de l'exécution, en fonction du profil d'exécution du code (JIT).

V8 peut compiler vers x86, ARM ou MIPS dans leurs éditions 32 bits et 64 bits.

Moteur V8 et javascript server side

Javascript server side & client side

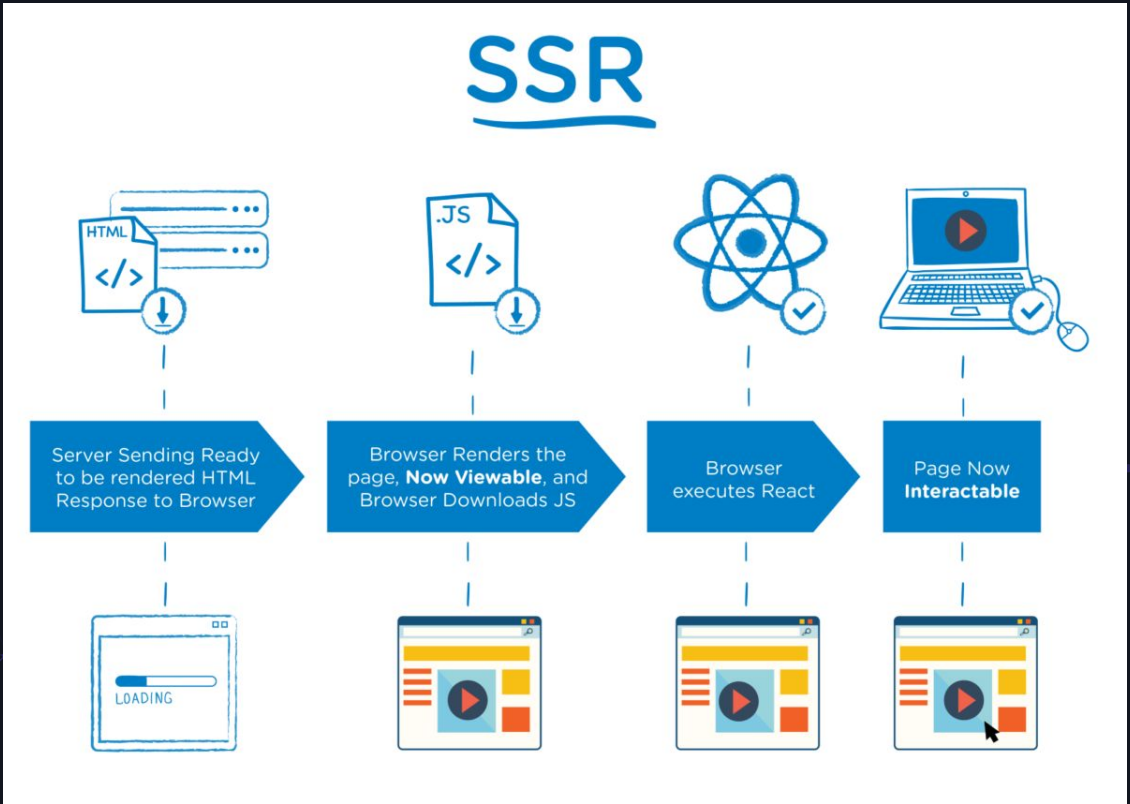
JavaScript a été créé en 1995 par Brendan Eich. Il a été standardisé sous le nom d'ECMAScript en juin 1997 par Ecma International. La version actuellement en vigueur de ce standard, depuis juin 2020, est la 11ème édition.

Javascript est à la base uniquement un langage interprété par le navigateur, donc côté client.

Avec l’arrivée du **moteur V8** il est donc désormais possible d'exécuter du javascript côté serveur.

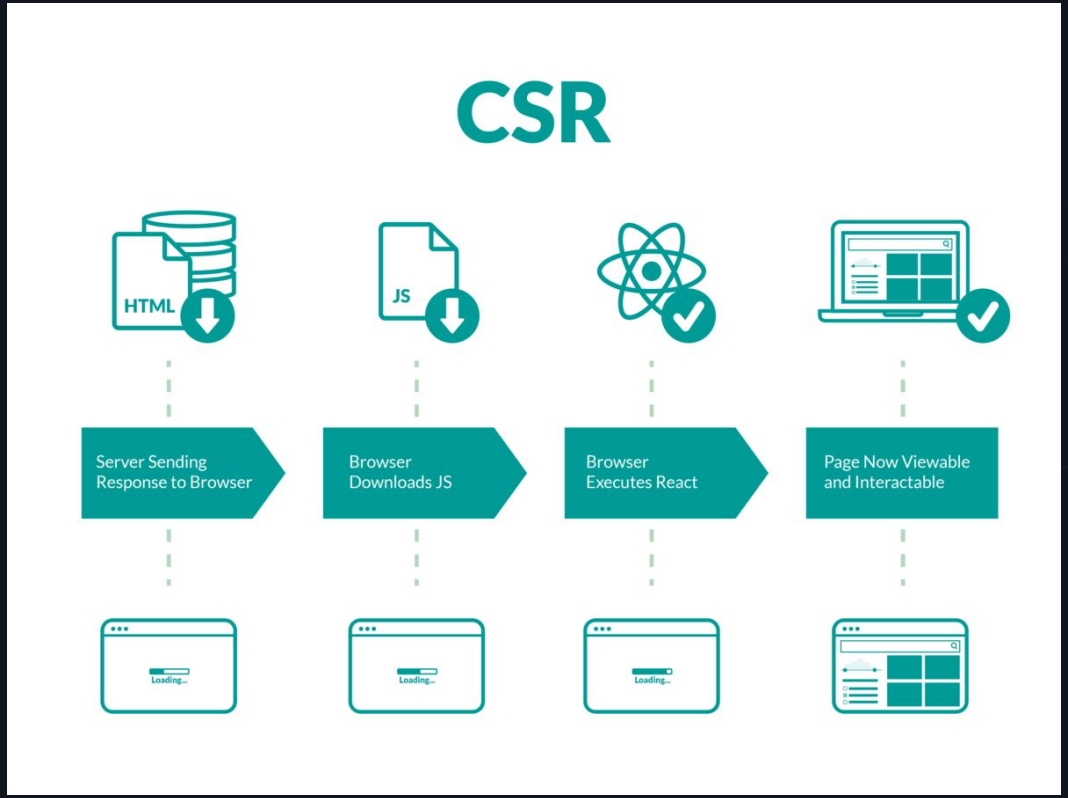
Moteur V8 et javascript server side

SSR



Moteur V8 et javascript server side

CSR



Installation & bases de node

- Node est, à l’instar de PHP, un langage interprété qui permet de créer des applications backend qui s'exécutent sur des serveurs.
- Le langage est 100% asynchrone et performant
- Node utilise Common JS (explications plus loin dans ce chapitre)
- Node s'exécute dans le terminal. On peut directement utiliser node dans le terminal pour exécuter des commandes ou alors lancer des fichiers.

Installation & bases de node

Installation

Pour ce cours nous avons besoin d'au minimum :

- Node js 16
- **npm** ou **yarn** ou **pnpm** (celui que vous maitrisez)

Installation & bases de node

Lancer node dans le terminal 1/2

```
Welcome to Node.js v14.17.0.
Type ".help" for more information.
>
```

Installation & bases de node

Lancer node dans le terminal 2/2

```
> console.log('toto');
toto
undefined
```

```
> const message = "Hello, world!"
undefined
> console.log(message);
Hello, world!
undefined
```

On peut exécuter du code javascript directement dans le terminal

Installation & bases de node

Exécuter des fichiers js

```
const message = "Hello, world!";
console.log(message);
```

hello.js

```
node hello.js # J'éxecute le code en tapant node {le nom de mon fichier}
Hello, world! # Node éxecute mon fichier et m'affiche Hello, World
```

je lance le fichier dans mon terminal

Installation & bases de node

Le serveur node

```
const http = require('http');

const server = http.createServer((req, res) => {
    res.end('Hello, world !');
});

server.listen(3000, () => console.log('Le serveur tourne sur http://localhost:3000'));
```

Ces 5 lignes de code permettent de lancer un serveur node js sur le port 3000.

Installation & bases de node

Requêtes et réponses

Dans la méthode http.createServer prend en paramètre la fonction *requestListener* qui a pour signature :

function (request, response) { }

Les deux paramètres de *requestListener* sont injectés directement dans la closure et on peut les récupérer pour voir le contenu.

La requête est un objet de type **IncomingMessage** qui possède plusieurs méthodes et propriétés : docs

La réponse est un objet de type **ServerResponse** qui possède plusieurs méthodes et propriétés : docs

Même si node a un server intégré, il n'est pas aisé de récupérer les données des requêtes :

- Pour récupérer la valeur d'une query string, vous devez récuperer la propriété url dans IncomingMessage, la parser et récupérer les valeurs
- Pour récupérer la valeur d'un payload, vous devez utiliser des listeners node sur l'event *"data"* (peu utilisé).

TP

Pour chaque exercice créez un fichier et rangez les dans un dossier nommé TP1.

1. Renvoyer une réponse avec un code HTTP 400.
2. Renvoyez une réponse au format JSON avec comme schema :

```
{
  "message": "Hello world",
  "status": 200,
}
```

3. Renvoyez une réponse JSON, en 200, avec comme schema :

```
{
  "firstname": "Votre prénom",
  "lastname": "Votre nom",
  "birthdate": "Votre date de naissance",
  "color": "Votre couleur préférée,
}
```

4. Envoyez en query string le paramètre “message” et affichez “Your message: {le message récupéré}”. Pour vous aider, utilisez le package URL de node (ou cherchez sur internet :)
5. Bonus : faire un routeur qui renvoie “hello world” quand vous essayez d’accéder à la route “/welcome” et “Not found” quand vous essayez d'accéder à n’importe quelle autre route.

CommonJS vs ESM

CommonJS est un projet de développement d'une API pour écrire des programmes JavaScript s'exécutant ailleurs que dans un navigateur Web, et portables sur les différents interpréteurs et environnements d'exécution implémentant CommonJS. CommonJS ne fait pas partie de la spécification ECMAScript officielle contrairement aux ECMAScript modules (ESM), qui est une implémentation récente qui permet d’unifier et de standardiser la manière dont sont chargés les modules javascript.

De plus, ESM est chargé de manière asynchrone et CommonJS de manière synchrone.

CommonJS vs ESM

ESM

```
// Importer un module
import Vue from 'vue';

// Exporter un module

export default Vue;

// Exporter plusieurs modules

export const Router = () => {};
export const Auth = () => {};

// Importer plusieurs modules

import { Router, Auth } from 'myfile.js';
```

CommonJS vs ESM

CommonJS

```
// Importer un module
const package = require('module-name');

// Exporter un module
// uppercase.js

exports.uppercase = (str) => str.toUpperCase();

// Importer mon module

const uppercaseModule = require('uppercase.js');
uppercaseModule.uppercase('test');

// ou

const { uppercase } = require('uppercase.js');
uppercase('test');
```

```
// Utiliser module.exports pour exporter vos modules

const anotherThing = () => {};
const something = "this string is exported";

module.exports = {
  something,
  anotherThing,
};
```

Packages

Les packages managers

Il y a deux principaux packages managers :

- **NPM** qui est est le gestionnaire de paquets officiel de Node.js. Depuis la version 0.6.3 de Node.js, npm fait partie de l'environnement et est donc automatiquement installé par défaut.
- **Yarn** qui est le plus gros concurrent de NPM, qui est plus rapide et permet de travailler plus facilement en monorepo notamment.

Il existe de nombreux packages managers moins connus, je recommande l'utilisation de **pnpm** qui est extrêmement rapide, il possède **exactement** la même api que npm (même commandes etc) et il permet d'économiser de place sur sa machine.

Packages

Lancer un projet

Pour pouvoir installer des packages dans le dossier de votre projet javascript, vous devez init un projet grâce à la commande “npm init”. Ces informations vont permettre notamment de pouvoir push son code (package) sur npmjs.org vous permettant alors de pouvoir installer votre package depuis npm directement.

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (api)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```

Packages

Publier un package

Publier un package sur npmjs.org est plutôt simple :

1. Créez vous un compte sur npmjs.org
2. une fois inscrit, connectez vous sur votre terminal avec la commande : **npm login**
3. Une fois la connexion effectuée, créer dossier avec un bout de code javascript (avec un readme à la racine c'est plus sympa)
4. A la racine du dossier, lancez la commande : **npm publish**
5. Et voilà !

Attention : le nom du package est important, il faut qu'il soit unique sinon npm refusera de publier le paquet.

TP

Exo 1 :

1. Créer un package dans un nouveau dossier, donnez lui un nom unique
2. Dans le fichier d'entrypoint, créez une méthode qui permet de transformer une chaîne de caractère en slug (utilisez un package)
3. Créez ensuite une méthode qui tri par ordre alphabétique un tableau de mot aléatoire
4. Toujours dans le fichier d'entrypoint, créez une méthode qui permet de mettre une majuscule à chaque mot d'une chaîne de caractère
5. Enfin créez une méthode qui retourne un élément aléatoire d'un tableau.

Exo 2 :

1. Créez deux fichiers arrays.js & strings.js, mettez les méthodes des strings dans strings.js et celle sur les arrays dans arrays.js
2. Exportez les méthodes dans chaque fichier
3. Exportez les méthodes dans le fichier d'entrypoint pour qu'on puisse utiliser les méthodes quand on les importes
4. Publiez votre package sur npm

Exo 3 :

1. Une fois fini, installer le package d'un de vos camarades et exécutez le !

Les promesses

Rappel sur les promesses

Une promesse est un objet (Promise) qui représente la complétion ou l'échec d'une opération asynchrone. La plupart du temps, on « consomme » des promesses.

En résumé, une promesse est un objet qui est renvoyé et auquel on attache des callbacks plutôt que de passer des callbacks à une fonction. Ainsi, au lieu d'avoir une fonction qui prend deux callbacks en arguments

Les promesses

Rappel sur les promesses

```
function faireQqcALAncienne(successCallback, failureCallback){
  console.log("C'est fait");

  if (Math.random() > .5) {
    successCallback("Réussite");
  } else {
    failureCallback("Échec");
  }
}

function successCallback(résultat) {
  console.log("L'opération a réussi avec le message : " + résultat);
}


function failureCallback(erreur) {
  console.error("L'opération a échoué avec le message : " + erreur);
}

faireQqcALAncienne(successCallback, failureCallback);
```

Sans promesse

Les promesses

Rappel sur les promesses

```
function faireQqc() {
  return new Promise((successCallback, failureCallback) => {
    console.log("C'est fait");

    if (Math.random() > .5) {
      successCallback("Réussite");
    } else {
      failureCallback("Échec");
    }
  })
}

function successCallback(résultat) {
  console.log("L'opération a réussi avec le message : " + résultat);
}

function failureCallback(erreur) {
  console.error("L'opération a échoué avec le message : " + erreur);
}

const promise = faireQqc();
promise.then(successCallback, failureCallback);
```

Avec promesse

Les promesses

Try catch, await / async

Pour gérer les erreurs en JS, vous pouvez utilisez un try catch. Les try catch fonctionnent très bien avec les promesses et permet de rendre son code plus lisible.

Ici on exécute une promesse basique, et si jamais elle échoue on affiche l'erreur.

```
function execute() {
  try {
    myPromise(1).then(res => console.log(res));
  } catch(e) {
    console.log(e);
  }
};

const myPromise = (value) => {
  return new Promise((resolve, failure) => {
    if(value === 1) {
      resolve('Hey ! This is my promise');
    } else {
      failure('Oops');
    }
});
}

execute();
```

Les promesses

Try catch, await / async

Utiliser des then et des catch après une promesse est compliqué et pas forcément pratique quand on veut récupérer la valeur de cette promesse. Pour cela, on peut utiliser la syntaxe **async / await.**

Await permet d'attendre que la promesse s'exécute avant de continuer à exécuter la suite du code.

Pour pouvoir utiliser **await**, il faut que la fonction soit marqué comme "async" (sinon vous aurez une erreur).

```
async function execute() {
  try {
    const firstPromise = myPromise(1);
    console.log(firstPromise);
    const secondPromise = myPromise(2);
    console.log(secondPromise);
  } catch(e) {
    console.log(e);
  }
};

const myPromise = (value) => {
  return new Promise((resolve, failure) => {
    if(value === 1) {
      resolve('Hey ! This is my promise');
    } else {
      failure('Oops');
    }
});
}

execute();
```

TP

1. Créez une fonction qui prend en paramètre une chaîne de caractère, si la chaîne de caractère fait plus de 20 caractères, la promesse échoue, sinon la fonction renvoie **true**
2. Créez une fonction qui prend en paramètre deux int, si la première variable est supérieure à la seconde, la promesse renvoie la différence entre les deux variables, sinon elle échoue.
3. Créez une fonction qui prend en paramètre une date de naissance d'une personne au format DD/MM/YYYY. Si la personne est mineure, la promesse échoue, sinon elle renvoie **true**
4. Exécutez toutes les fonctions créées ci-dessus avec des then & catch
5. Exécutez toutes les fonctions créées ci-dessus dans un try catch avec des awaits
6. Faites un init de package, installez axios, et refaites tous les appels du TP sur les clients API avec axios. Tous vos appels doivent être dans un try catch avec un await. A la fin renvoyez un objet avec les informations récupérées.

Express JS

Présentation et installation

“Fast, unopinionated, minimalist web framework for Node.js”

Express.js est un framework qui permet de construire des applications web basées sur Node.js. C'est de fait le framework standard pour le développement de serveur en Node.js.

```
npm install express --save
```

Créez un dossier et installer express

Créer sa première application express

```
const express = require('express');
const app = express();
const port = 3000;

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

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

Le routeur

Router 101

Express vous permet de décrire facilement et de manière claire vos routes. La syntaxe est toujours la même : le chemin et le handler. Vous pouvez écrire directement le handler sous forme de closure ou alors l'importer depuis un autre fichier.

```
app.METHOD(PATH, HANDLER);
```

```
app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.post('/', function (req, res) {
  res.send('Got a POST request');
});

app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user');
});

app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user');
});
```

Le routeur

Les paramètres de route

Pour mettre des paramètres dans vos endpoints, rien de plus simple : préfixez le paramètre dans la route par **deux points**.

Les paramètres de route sont ensuites automatiquement injectés dans la propriété **params** de la request.

```
app.get('/users/:id', function (req, res) {
  res.send(req.params);
});
```

Le routeur

Les queries strings

La gestion des queries strings en express est très simple : les paramètres sont automatiquement injectés dans la propriété **query** de la request.

```
// on tape /users?message="hello"

app.get('/users', function (req, res) {
  res.send(req.query.message);
});
```

Récupérer le payload d'une requête

Par défaut, on ne peut pas récupérer le payload d'une requête express. Pour pouvoir le faire, il faut installer **body-parser** et le register dans express.

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

const port = 3000;
const app = express();

app.use(bodyParser.json());

app.get('/', async function(req, res) {
    console.log(req.body);
    console.log(req.query.message);
    res.send('Hello world');
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

L’objet response

L’objet response qui est injectée dans le handler d’une route possède plusieurs méthodes et propriétés. On a vu **body** et **query**, mais il y en a d’autres.

- **res.send() :** permet de renvoyer une réponse
- **res.status() :** permet de saisir le code HTTP renvoyé
- **res.end() :** permet de terminer le processus de réponse
- **res.json() :** permet de retourner une réponse au format JSON. On peut directement retourner un objet JS qui se sérialisé en JSON.
- **res.redirect() :** permet de faire une redirection
- **res.set() :** permet de set un header de réponse

Il est possible de chaîner les méthodes :

```
res.status(404).json({ message: "Not Found" });
```

TP

Faites tous les exos dans la même application express

1. Créez une route **GET** sur ‘/hello-world’ qui renvoie un json { “message”: “Hello world !” }
2. Créez une route **GET** sur “/message” qui renvoie la valeur de la query string “message”. Si la query string fait plus de 20 caractères, l’api renvoie 400 et comme messag { message: “Bad Request” }
3. Créez une route **POST** sur ‘/infos/headers’ qui renvoie tous les headers de la request sous format json
4. Créez une route **POST** qui récupère un payload de type { "firstname": "Toto", "birthdate": "10/01/1990" }, si l’user est majeur retourner 200 avec comme message { message: "Welcome :firstname" } sinon afficher une 403 avec comme message { message: "Forbidden" }
5. Créer une route **GET** sur **“**/rick-roll” qui redirige l’utilisateur sur **https://youtu.be/dQw4w9WgXcQ**
6. Créez une route **DELETE** sur “/custom-header” qui renvoie un header de response Message : “Hello world !”
7. Prenez tous les handlers des routes précédentes et mettez les dans un autre fichier.
8. Créez une route **GET** sur “/params/:id/:key/:slug”, renvoyez les paramètres de route sous format JSON
9. Créez un dossier **handlers**, à l’intérieur créez un fichier “users”. Dans ce fichier, créez 5 handlers :
    a. Une handler **getUsers** sur ‘/users’ qui renvoie “All users”
    b. Une handler **getUser** sur ‘/users/:id’ qui renvoie “User id :id”
    c. Une handler **createUser** sur ‘/users’ qui renvoie “Create user”
    d. Une handler **deleteUser** sur ‘/users/:id’ qui renvoie “Delete user id :id”

Sequelize

Rappel sur les ORM

Un mapping objet-relationnel (en anglais object-relational mapping ou ORM) est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet. Ce programme définit des correspondances entre les schémas de la base de données et les classes du programme applicatif.

On pourrait le désigner par là, « comme une couche d'abstraction entre le monde objet et monde relationnel ».

Sequelize

“Sequelize is a promise-based Node.js ORM for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. It features solid transaction support, relations, eager and lazy loading, read replication and more.”

C’est l’ORM le plus généraliste et accessible sur javascript.

La documentation est par **ici**

Installation et configuration

```
npm install --save sequelize

npm install --save sqlite3
```

Installation et configuration

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

const { User } = require('./sequelize.js');

const port = 3000;
const app = express();

app.use(bodyParser.json());

app.get('/', async function(req, res) {
    const jane = await User.create({
      username: 'janedoe',
      birthday: new Date(1980, 6, 20)
    });
    res.json(jane);
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

index.js

```
const { Sequelize, Model, DataTypes } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:');

(async () => {
  await sequelize.sync();
})();

class User extends Model {}

User.init({
  username: DataTypes.STRING,
  birthday: DataTypes.DATE
}, { sequelize, modelName: 'user' });


module.exports = {
  User,
}
```

sequelize.js

Définition des modèles

Organiser son code

```
const { Sequelize } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:');

(async () => {
  await sequelize.sync();
})();

module.exports = {
  sequelize,
};
```

On déplace le code de base de sequelize dans ./core/sequelize.js

```
const { sequelize } = require('../core/sequelize.js');
const { Model, DataTypes } = require('sequelize');

class User extends Model {}

User.init({
  username: DataTypes.STRING,
  birthday: DataTypes.DATE
}, { sequelize, modelName: 'user' });


module.exports = {
  User,
}
```

Pour chaque modèle, on va créer un fichier dans ./models avec comme nomenclature Model.model.js

sequelize-cli

Pour une utilisation plus performante et fidèle à la documentation de sequelize, on va utiliser sequelize-cli.
Ce cli permet de :

- Gérer les migrations
- Créer des seeders
- Résoudre les problèmes de circle dependencies

sequelize-cli

Dans votre projet **express** fraîchement créé, lancez la commande ci-dessous pour initialiser sequelize dans votre projet.

```
npx sequelize-cli init
```

sequelize-cli

Pour préciser la database et le driver, on va dans config/config.js et on change la config

```
 "development": {
    "dialect": "sqlite",
    "storage": "./database.sqlite"
},
```

sequelize-cli

```
Sequelize CLI [Node: 14.17.0, CLI: 6.3.0, ORM: 6.12.0-beta.3]

sequelize-cli <commande>

Commandes :
  sequelize-cli db:migrate                        Run pending migrations
  sequelize-cli db:migrate:schema:timestamps:add  Update migration table to have timestamps
  sequelize-cli db:migrate:status                 List the status of all migrations
  sequelize-cli db:migrate:undo                   Reverts a migration
  sequelize-cli db:migrate:undo:all               Revert all migrations ran
  sequelize-cli db:seed                           Run specified seeder
  sequelize-cli db:seed:undo                      Deletes data from the database
  sequelize-cli db:seed:all                       Run every seeder
  sequelize-cli db:seed:undo:all                  Deletes data from the database
  sequelize-cli db:create                         Create database specified by configuration
  sequelize-cli db:drop                           Drop database specified by configuration
  sequelize-cli init                              Initializes project
  sequelize-cli init:config                       Initializes configuration
  sequelize-cli init:migrations                   Initializes migrations
  sequelize-cli init:models                       Initializes models
  sequelize-cli init:seeders                      Initializes seeders
  sequelize-cli migration:generate                Generates a new migration file
[alias : migration:create]
  sequelize-cli model:generate                    Generates a model and its migration
    [alias : model:create]
  sequelize-cli seed:generate                     Generates a new seed file
     [alias : seed:create]
```

sequelize-cli

Pour créer un modèle, rien de plus simple : utiliser la commande model:generate

```
npx sequelize-cli model:generate --name User --attributes firstName:string,lastName:string,email:string
```

sequelize-cli

Création de modèle

Le cli va générer automatiquement un modèle d'après les paramètres que vous avez saisi dans la commande. Tous les modèles seront disponibles dans le dossier models.

A noter que le cli va vous créer en parallèle un fichier de migration. Si vous voulez ajouter / supprimer / modifier des champs de votre modèle, il faudra modifier la migration aussi (ou alors en faire une nouvelle)

```
'use strict';
const {
  Model
} = require('sequelize');
module.exports = (sequelize, DataTypes) => {
  class User extends Model {
    /**
     * Helper method for defining associations.
     * This method is not a part of Sequelize lifecycle.
     * The `models/index` file will call this method automatically.
     */
    static associate(models) {
      // define association here
    }
  };
  User.init({
    firstName: DataTypes.STRING,
    lastName: DataTypes.STRING,
    email: DataTypes.STRING
  }, {
    sequelize,
    modelName: 'User',
  });
  return User;
};
```

sequelize-cli

Création de seeds

Pour créer des seeds (concept identique à celui de Laravel), utilisez la commande seed:generate

```
npx sequelize-cli seed:generate --name fake-users
```

Modifier le fichier de seed puis lancer la commande ci dessous pour exécuter le code

```
npx sequelize-cli db:seed:all
```

Associations

Sequelize met à disposition 4 types d'association qui peuvent être combinée :

- **HasOne**
- **BelongsTo**
- **HasMany**
- **BelongsToMany**

Les relations sont à déclarer dans la méthode **associate** dans votre modèle.

```
User.hasOne(Profile);
Post.belongsTo(User);
User.hasMany(Post);
Role.belongsToMany(User);
```

Associations

On peut préciser des options pour chaque associations, en second paramètre de la fonction.

```
Foo.hasOne(Bar, {
  onDelete: 'RESTRICT',
  onUpdate: 'RESTRICT'
});
Bar.belongsTo(Foo);
```

On peut configurer les comportements
on delete & on update

Associations

```
// Option 1
Foo.hasOne(Bar, {
  foreignKey: 'myFooId'
});
Bar.belongsTo(Foo);

// Option 2
Foo.hasOne(Bar, {
  foreignKey: {
    name: 'myFooId'
  }
});
Bar.belongsTo(Foo);
```

On peut préciser le nom de la clé étrangère (conseillé)

```
const { DataTypes } = require("Sequelize");

Foo.hasOne(Bar, {
  foreignKey: {
    // name: 'myFooId'
    type: DataTypes.UUID
  }
});
Bar.belongsTo(Foo);
```

On peut configurer le type de la clé étrangère

Associations

```
Foo.hasOne(Bar, {
  foreignKey: {
    allowNull: false
  }
});
```

On peut préciser si on autorise une relation null

Associations

Attention, il faut bien préciser la référence de la FK dans la migration.

Pour cela rajoutez “references” dans la colonne utilisée comme foreign key.

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Post', {
      name: Sequelize.DataTypes.STRING,
      authorId: {
        type: Sequelize.DataTypes.INTEGER,
        references: {
          model: {
            tableName: 'users',
            schema: 'schema'
          },
          key: 'id'
        },
        allowNull: false
      },
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Post');
  }
}
```

Manipuler la donnée

Insert

```
const jane = await User.create({ firstName: "Jane", lastName: "Doe" });
```

Pour ajouter des lignes en BDD, rien de plus simple, importez votre modèle et utilisez la méthode Model.create avec en paramètre les infos à sauvegarder

Manipuler la donnée

Update

```
await User.update({ lastName: "Doe" }, {
  where: {
    lastName: null
  }
});
```

Pour faire un UPDATE, importez votre modèle, ajoutez les champs à mettre à jour puis précisez la condition.

Manipuler la donnée

Select : findAll

```
// Pour récupérer tous les users
const users = await User.findAll();

// Pour récuprérer tous les users avec uniquement leur nom
const userNames = await User.findAll({
  attributes: ['name']
});
```

Pour récupérer toutes les lignes d'une table, on utilise Model.findAll() (= **SELECT * from users**)
On peut préciser les attributs qu'on veut sélectionner (= **SELECT name from users**)

Manipuler la donnée

Select : findOne

```
const project = await Project.findOne({ where: { title: 'My Title' } });
```

Manipuler la donnée

Select : findByPk

```
const project = await Project.findByPk(123);
```

Manipuler la donnée

Select : conditions where

```
User.findAll({
  where: {
    role: 'admin'
  }
});
```

Comme pour UPDATE, on peut appliquer des conditions **WHERE** sur nos select, ici on veut filtrer remonter les utilisateurs avec le rôle *admin*

Manipuler la donnée

Delete

```
await User.destroy({
  where: {
    id: 24025024,
  }
});
```

Pour supprimer une ligne, rien de plus simple, utilisez Model.destroy() avec une condition *where*

TP

Partie 1 :

1. Créez un nouveau projet express from scratch avec sqlite3, sequelize, sequelize-cli.
2. Créez un modèle user avec : lastname, firstname, email, username, lien vers github
3. Créez un modèle post avec : title, content, date, author (relation vers user)
4. Créez un modèle comment avec : content, date, author (relation vers user)
5. Créez un modèle role avec : name
6. Rajoutez une relation “role” à l’utilisateur
7. Rajoutez une relation “post” à commentaire

Partie 2 :

1. Créez une route qui récupère un payload json qui permet de créer un utilisateur
2. Créez une route qui récupère un payload json qui permet de modifier un utilisateur (en patch, c’est à dire qu’il y a une modification uniquement des champs précisés dans le payload)
3. Créez une route qui permet de supprimer un utilisateur
4. Créez une route qui permet d’afficher un utilisateur par son id
5. Créez une route qui permet d’afficher tous les utilisateurs

TP

Partie 3 :

1. Faites la même chose que pour l'utilisateur mais pour les rôles, les commentaires et les posts
2. Sur la route qui affiche le post par id, rajoutez une query string qui permet d'afficher (ou non) les commentaires du post
3. Sur la route qui affiche un utilisateur par son id, rajoutez une query string qui permet d'afficher (ou non) ses posts

Partie 4 :

1. Créez des seeds pour remplir votre application avec de données
2. Faites une collection postman de toutes vos routes et exporter là dans votre repository
3. Si ce n'est pas déjà fait, vos routes doivent être scopés par modèle (ex : un fichier user.routes.js, post.routes.js) et chaque handler de route doit être scopé par modèle (user.handler.js, post.handler.js)