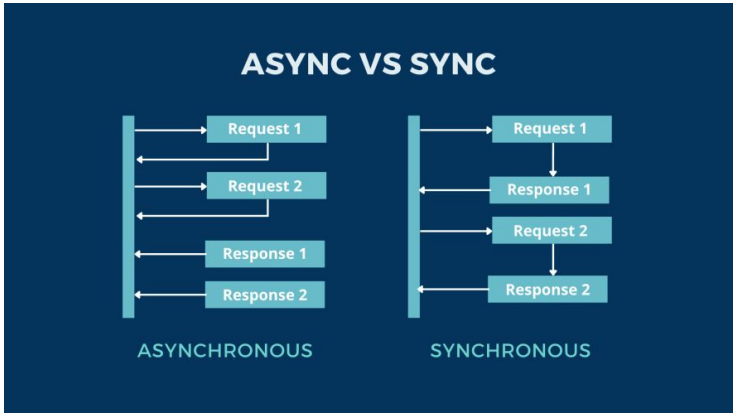
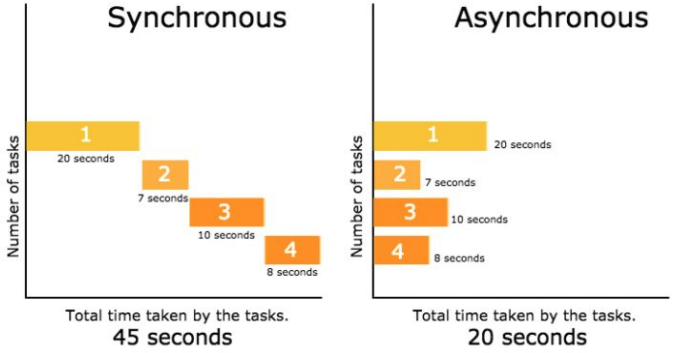
# Node.js

La programmation asynchrone & orientée événements

plb consultant

# Quel intérêt de développer en asynchrone ?

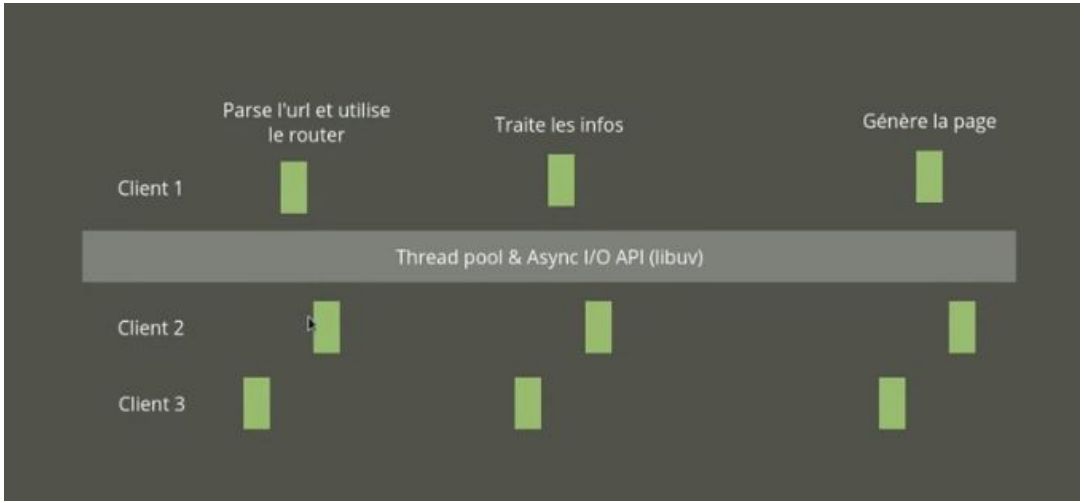


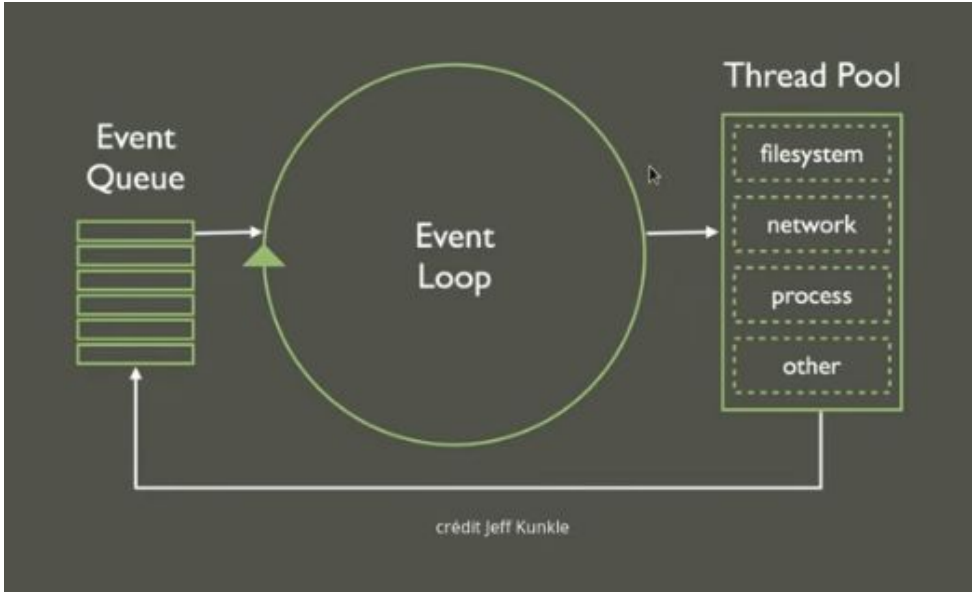
# Plan

1. La gestion événementielle
2. Flux de lecture
3. Flux d’écriture
4. Lecture d'une ressource en ligne
5. Principaux modules de l'API
6. Gestion des requêtes/réponses
7. Processus fils
8. TCP/UDP
9. API REST
10. Express
11. Postman
12. Le moteur de template EJS

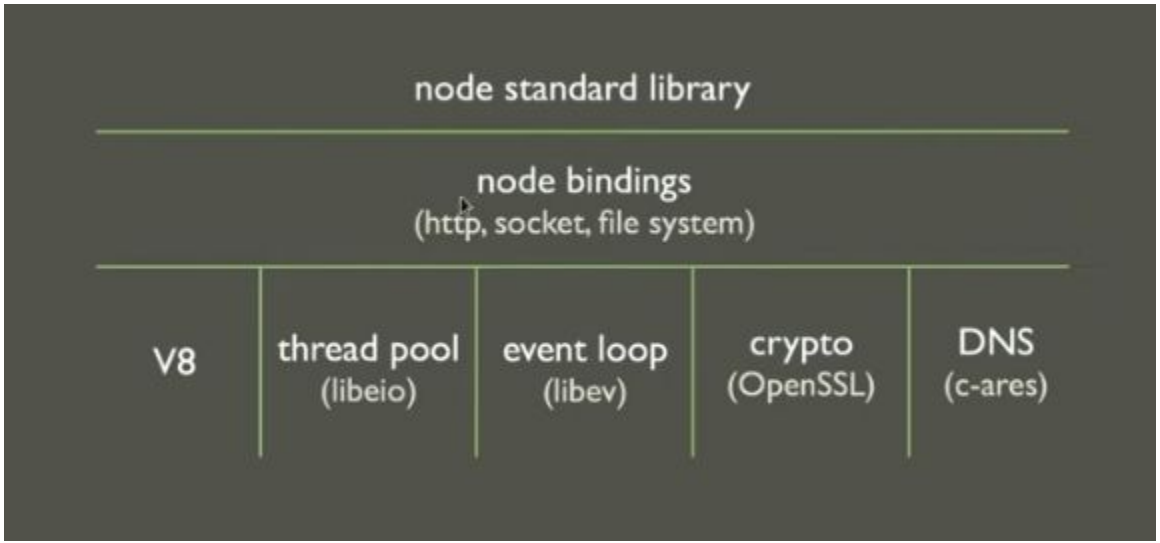
# Quel intérêt de développer en asynchrone ?



# La gestion événementielle



# La gestion événementielle



# La gestion événementielle

- Emitter : permet d’émettre un événement
- Listener : permet d’écouter un événement

# Flux de lecture

- fs.readFile(file) : permet de lire un fichier
- fs.createReadStream(file) : permet de lire un fichier par petit morceau
- Événement “data” : événement lorsque l’on reçoit des données
- Événement “end”: événement lorsque tout s’est bien déroulé
- Evénement “error” : événement lorsque l’on reçoit une erreur

# Flux de lecture

```
import fs from 'fs';
let file = 'demo.wav';
fs.readFile(file, (err, data) => {
    if (condition) throw err
    fs.writeFile('copy.mp4', data, (err) => {
        if (err) throw err
        console.log('Le fichier à bien été copié');
    })
})
```

# Flux de lecture

```
import fs from 'fs';
let file = 'demo.wav';

let read = fs.createReadStream(file)

read.on('data', (chunck) => {
  console.log('J\'ai lu ' + chunck.length);
})

read.on('end', ()=>{
  console.log('J\'ai fini de lire le fichier');
})
```

# Flux d’écriture

- fs.createWriteStream(‘nomdemonfichier.mp4’) : permet d’écrire un fichier par petit morceau
- pipe() : permet de connecter les flux
- Evénement “finish” : événement lorsque toutes les informations sont reçues.

# Flux d’écriture

```
import fs from 'fs';
let file = 'demo.wav';
fs.stat(file, (err, stat) => {
  let total = stat.size;
  let progress = 0;

  let read = fs.createReadStream(file);
  let write = fs.createWriteStream('copy.wav');

  read.on('data', (chunck) => {
    progress += chunck.length;
    console.log('J\'ai lu ' + (100 * progress/total) +'%' );
  });

  read.on('end', ()=>{
    console.log('J\'ai fini de lire le fichier');
  });

  read.pipe(write);

  write.on('finish', () => {
    console.log('Le fichier a bien été copié');
  })
})
```

# Exercice 1

- Écrivez un programme de nœud qui lit un fichier (passé en paramètre) dans la machine locale et affiche dans la console le contenu de celui-ci.

- node app.js test.txt

- Astuce : Vous avez besoin du module npm -fs

# Exercice 2

- Écrivez un programme node.JS qui crée un fichier txt avec le texte passé en paramètre

```
$ node ex5.js "hey ho! let's go"
The file was saved!
$ cat myText.txt
hey ho! let's go
```

- En plus: Améliorez l'exercice précédent pour utiliser le premier paramètre comme nom du fichier de destination

# Exercice 2

- Plus supplémentaire : Améliorer l'exercice précédent pour également lire et afficher dans la console le contenu du fichier

```
$ node ex5.js mySuperText.txt "it's a long way to the top..."
The file was saved!
it's a long way to the top...

$ cat mySuperText.txt
it's a long way to the top...
```

# Exercice 3

- Écrivez un programme node.JS qui recherche des informations dans le fichier .txt et vous renvoie le nombre de coïncidences trouvées.

- Si vous recherchez "John", et dans vos fichiers txt sont 2 John, devrait retourner 2

- Astuce : vous pouvez essayer d'enregistrer toutes les données dans un tableau et de comparer avec la recherche.

# Lecture d'une ressource en ligne

Module request:

- npm i request -S

```
let request = require('request');
let fs = require('fs');

let fileToRead = process.argv[2];

fs.readFile(fileToRead, 'utf8', function(err, url) {
    if (err) throw err;

    request(url, function (error, response, body) {
      console.log('body:', body);
    });
});
```

# Exercice

Écrivez un programme node.JS qui lit et affiche dans la console le code html d'une page externe. Le lien de la page externe doit être lu à partir d'un fichier link.txt

Astuce : Vous avez besoin du module npm ->request

# Exercice

Ecrire un programme node.JS qui liste le contenu du répertoire courant en indiquant s'il s'agit d'un répertoire ou d'un fichier

```
$ node ex4.js
FILE:ex1.js
FILE:ex2.js
FILE:ex3.js
FILE:ex4.js
FILE:getLinksNode.js
FILE:link.txt
DIR :node_modules
FILE:recursiveContentsDir.js
FILE:results.txt
FILE:test.txt
```

Astuce : Vous avez besoin fs.readdirSyncde , fs.lstatSyncetisDirectory()

# Principaux modules de l'API

- console
- util
- fileSystem
- events
- timer
- path
- net
- dns
- domain
- os

# Principaux modules de l'API

Console :

- new Console({ stdout: output, stderr: errorOutput });
- console.assert(false, 'Whoops %s work', 'didn\'t');
- console.clear()
- console.error('error #%d', code);

# Principaux modules de l'API

Util :

- debuglog() : Écrit les messages de débogage dans l'objet d'erreur.
- deprecate() :Marque la fonction spécifiée comme dépréciée.
- format() : Formate la chaîne de caractères spécifiée, en utilisant les arguments spécifiés.
- inherits() : Hérite des méthodes d'une fonction dans une autre.
- inspect() : Inspecte l'objet spécifié et retourne l'objet sous forme de chaîne de caractères.

# Principaux modules de l'API

FileSystem :

Lire des fichiers :

- fs.readFile('demofile1.html', callback)

Supprimer les fichiers :

- fs.unlink()

Créer des fichiers :

- fs.appendFile()
- fs.open()
- fs.writeFile()

# Principaux modules de l'API

Events :

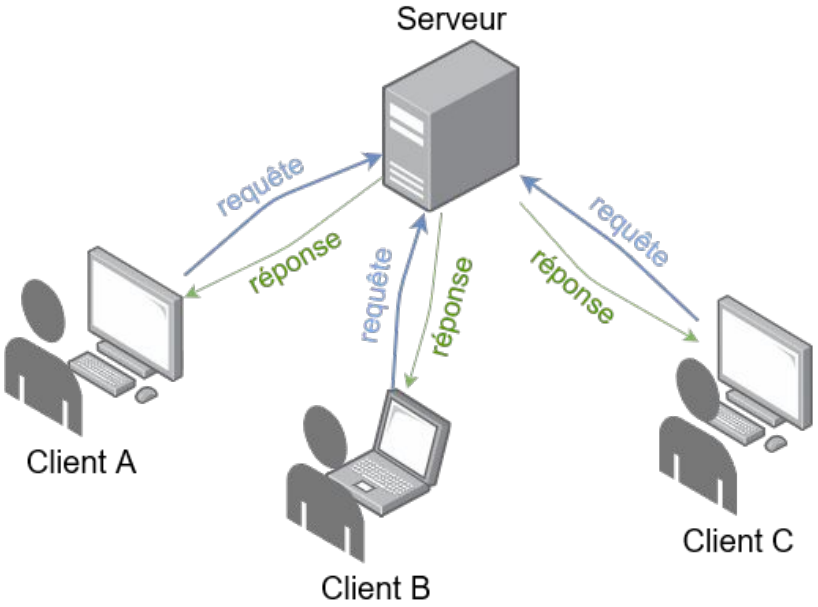
- Emitter : permet d’émettre un événement.
- .emit() : Permet de créer un événement.
- .on() : Permet d’écouter un événement.

# Principaux modules de l'API

Timers :

- setInterval()
- setTimeout()
- clearInterval()
- clearTimeout()

# Gestion des requêtes/réponses



# Gestion des requêtes/réponses

Le module http :

- const http = require('http') ou import http from 'http'
- http.createServer((request, response) => {})

L'objet "request":

- request.url
- request.host
- request.getHeaders()

L'objet "response":

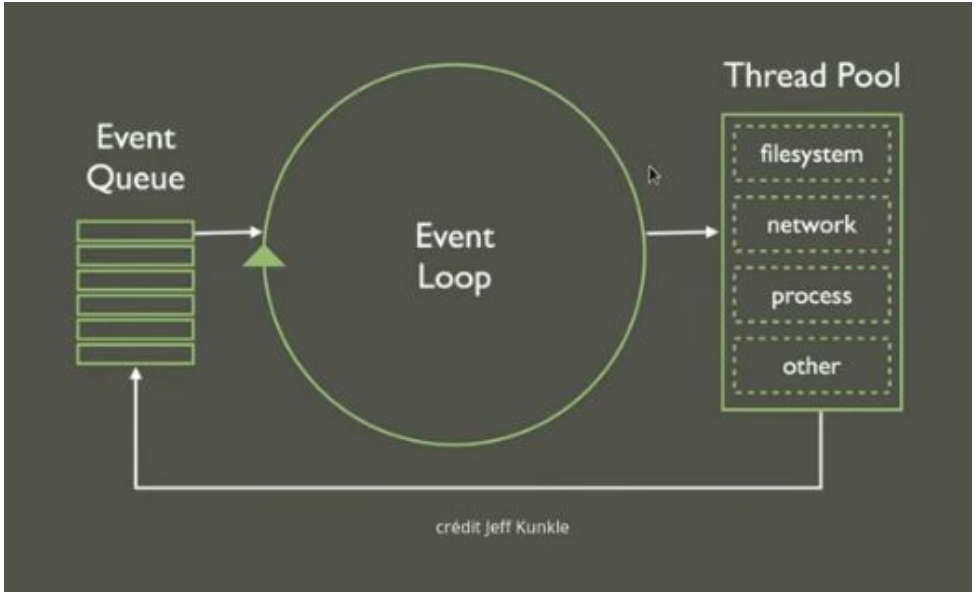
- response.statusCode
- response.setHeader()
- response.end()
- response.write()

# Gestion des requêtes/réponses

```
let app = App.start(3000);

app.on('home', function (res) {
  res.write('Je suis sur la home page');
})
```

```
class App {
  start(port) {
    let emitter = new EventEmitter();
    http.createServer((req, res) => {
      res.statusCode = 200;
      if (req.url === '/') {
        emitter.emit('home', res)
      }
      res.end()
    }).listen(port)
    return emitter;
  }
}
```

# Processus fils



# Processus fils

Le module “child_process”  : permet d'accéder aux fonctionnalités du système d'exploitation en exécutant n'importe quelle commande système.

4 manières différentes de créer un processus enfant dans Node :

- exec()
- execFile()
- spawn()
- fork()

# Processus fils

execFile() :Exécute une application externe avec la sortie mise en mémoire tampon après la sortie de l'application.

- execFile  est utilisé lorsque nous avons juste besoin d'exécuter une application et d'obtenir la sortie.
- execFile  ne doit pas être utilisé lorsque l'application externe produit une grande quantité de données et que nous devons consommer ces données en temps réel.

# Processus fils

spawn() : Exécute une commande système qui sera exécutée sur son propre processus.

Comme spawn renvoie un objet basé sur un flux, il est idéal pour gérer des applications qui produisent de grandes quantités de données ou pour travailler avec des données lors de leur lecture. Comme il est basé sur un flux, tous les avantages du flux s'appliquent également :

- Faible empreinte mémoire
- Produisez ou consommez paresseusement des données dans des blocs tamponnés.
- Événementiel et non bloquant
- Les tampons vous permettent de contourner la limite de mémoire de tas V8

# Processus fils

exec() : doit être utilisé lorsque nous avons besoin d'utiliser des fonctionnalités du shell

- exécutera la commande dans un shell mappé sur /bin/sh (linux) et cmd.exe (windows)
- doit être utilisé avec prudence car l'injection de shell  peut être exploitée.

# Processus fils

fork() : La méthode fork ouvrira un canal IPC permettant le passage de messages entre les processus Node :

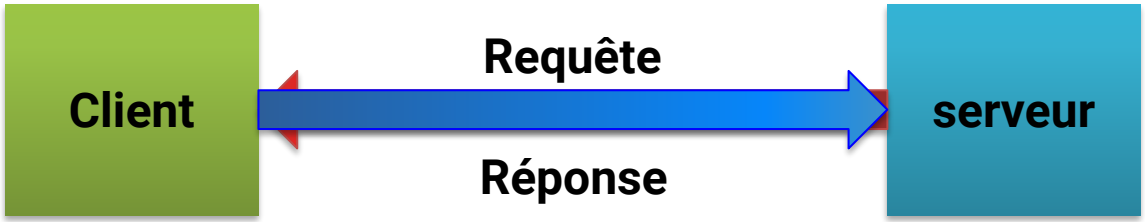
Sur le processus enfant, process.on('message') et process.send('message to parent') peuvent être utilisés pour recevoir et envoyer des données

Sur le processus parent, child.on('message') et child.send('message to child') sont utilisés

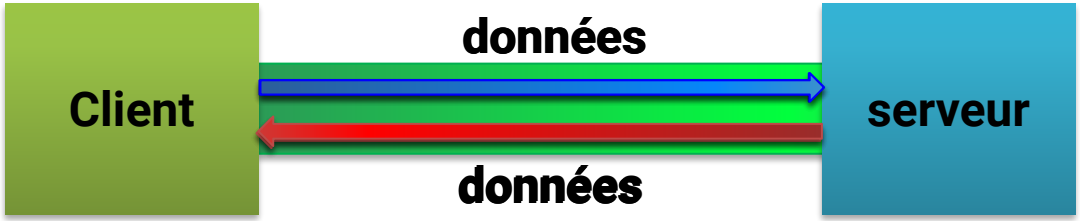
Chaque processus a sa propre mémoire, avec ses propres instances V8 supposant au moins 30 ms de démarrage et 10 Mo chacune.

# TCP/UDP

Ajax : le serveur ne peut envoyer des donner que si le client les demande.




Les websocket : permettent de créer une connexion entre le navigateur et le serveur. L'envoie des données se fait dans les deux sens (full-duplex) en utilisant le protocole TCP



# TCP/UDP

TCP : FTP, Email, navigateur web

- protocole bilatéral
- fiabilité

UDP : Live Streaming, Jeu en ligne

- protocole unilatéral
- rapidité

# TCP/UDP

- TCP : createServer() du module “net” ou “http”
- UDP : createSocket() du module “dgram”

# TP

Extrait de code

# Express

1. Installation : npm install express --save
2. créer un fichier index.js :

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

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

let server = app.listen(8081, function () {
  let host = server.address().address
  let port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

# Express

Routage de base :

```
// This responds with "Hello World" on the homepage
app.get('/', function (req, res) {
    console.log("Got a GET request for the homepage");
    res.send('Hello GET');
})

// This responds a POST request for the homepage
app.post('/', function (req, res) {
    console.log("Got a POST request for the homepage");
    res.send('Hello POST');
})

// This responds a DELETE request for the /del_user page.
app.delete('/del_user', function (req, res) {
    console.log("Got a DELETE request for /del_user");
    res.send('Hello DELETE');
})

// This responds a GET request for the /list_user page.
app.get('/list_user', function (req, res) {
    console.log("Got a GET request for /list_user");
    res.send('Page Listing');
})
```

# Express

Servir des fichiers statiques :

```
app.use(express.static('public'));
```

# Express

Méthode GET :

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form action = "http://127.0.0.1:8081/process_get" method = "GET">
    First Name: <input type = "text" name = "first_name">  <br>
    Last Name: <input type = "text" name = "last_name">
    <input type = "submit" value = "Submit">
  </form>
</body>
</html>
```

```js
app.use(express.static('public'));
app.get('/index.html', function (req, res) {
  res.sendFile( __dirname + "/" + "index.html" );
})

app.get('/process_get', function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.query.first_name,
      last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

let server = app.listen(8081, function () {
   let host = server.address().address
   let port = server.address().port

   console.log("Example app listening at http://%s:%s", host, port)
})
```

# Express

Méthode POST :

```
let bodyParser = require('body-parser');

// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })
```

```
<form action = "http://127.0.0.1:8081/process_post" method = "POST">
   First Name: <input type = "text" name = "first_name">  <br>
   Last Name: <input type = "text" name = "last_name">
   <input type = "submit" value = "Submit">
```

```
app.post('/process_post', urlencodedParser, function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.body.first_name,
      last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})
```

# Express

Téléchargement de fichier :

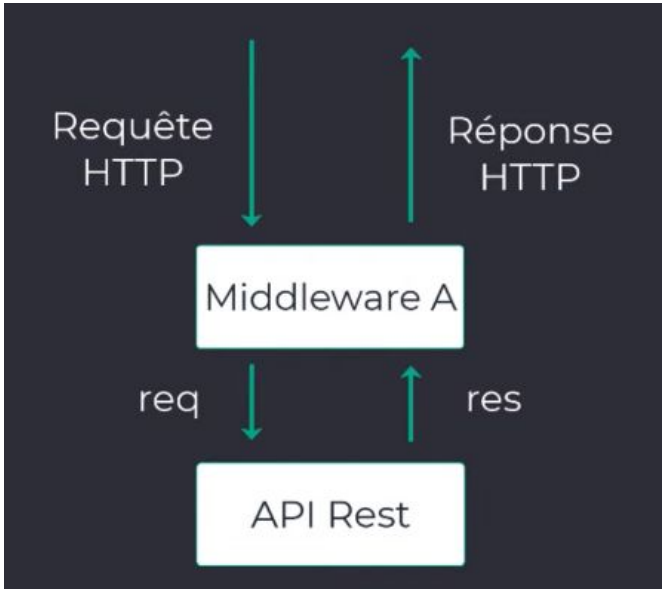
```
let multer  = require('multer');
const upload = multer({ dest: 'uploads/' })
```

```
<h3>File Upload:</h3>
    Select a file to upload: <br />

  <form action = "http://127.0.0.1:8081/file_upload" method = "POST"
    enctype = "multipart/form-data">
    <input type="file" name="file" size="50" />
    <br />
    <input type="submit" value="Upload File" />
  </form>
```

```
app.post('/file_upload', cpUpload, function (req, res) {
  req.files.file.forEach(element => {
    let file = __dirname + "/" + element.originalname;
    console.log('file :', file);

    fs.readFile( element.path, function (err, data) {
      fs.writeFile(file, data, function (err) {
        if( err ) {
          console.log( err );
          res.end( JSON.stringify( err ) );
          } else {
            response = {
              message:'File uploaded successfully',
              filename:element.originalname
            };
            res.end( JSON.stringify( response ) );
          }

      });
    });

  });
})
```

# Express

Téléchargement de fichiers :



# Express

Les URLs :

```
app.get('/pokemon/:id', (req, res) => {
  const id = req.params.id;
  res.send(`Vous avez demandé le pokémon n° ${id}}`)
})

app.get('/pokemon/:id/:poke', (req, res) => {
  const id = req.params.id;
  const poke = req.params.poke;
  res.send(`Vous avez demandé le pokémon n° ${id}} qui est ${poke}`)
})
```

# Express

Le Router :

1. Créer un fichier dans un répertoire “routes”
2. Ajouter à ce fichier le router, les routes et exporter
3. Ajouter le router créé au serveur

# Express

Le Router :

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

const router = express.Router();

/* POST login. */
router.get('/:id', function (req, res) {
    const id = req.params.id;
    res.send(`Vous avez demandé le pokémon n° ${id}`)
});

router.get('/pokemon/:id/:poke', (req, res) => {
    const id = req.params.id;
    const poke = req.params.poke;
    res.send(`Vous avez demandé le pokémon n° ${id}} qui est ${poke}`)
})

module.exports = router;
```

```
const pokemon = require('./routes/pokemon.js');

app.use('/pokemon', pokemon);
```

# Le moteur de template EJS

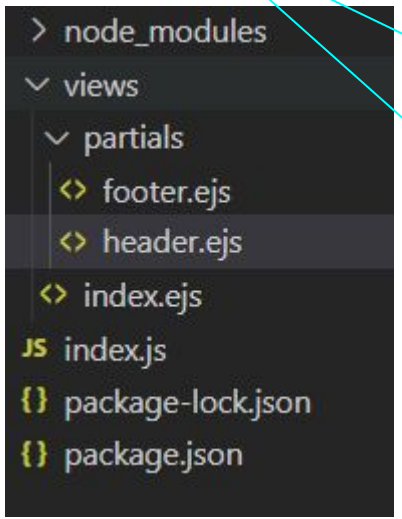
Installation/Configuration :

- npm i -S express ejs

- app.set("view engine", "ejs");

# Le moteur de template EJS

Les “partials” :




```ejs
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <%- include('./partials/header'); %>
  <main>
    <div class="ui segment">
      <h1>Mon pokedex</h1>
      <p>Ceci est un dictionnaire des pokémons</p>
    </div>
  </main>
  <%- include('./partials/footer'); %>
</body>
</html>
```

# Le moteur de template EJS

Transmission des données aux fichiers EJS :

```ejs
<%- include('./partials/header'); %>
  <main>
    <div class="ui segment">
      <h1><%= title %></h1>
      <% if (pokemons.length <= 1) { %>
        <h2>Mon Pokémon</h2>
      <% } else { %>
        <h2>Mes Pokémons</h2>
      <% } %>
      <% pokemons.forEach(function(pokemon) { %>
        <li class="ui segment">
          <strong><%= pokemon.name %></strong>
          est de type <%= pokemon.type %>.
        </li>
      <% }); %>
    </div>
  </main>
  <%- include('./partials/footer'); %>
```

Condition

Données transmisent à la template

Boucle

Nom de la template

```js
app.get("/pokemon", function (req, res) {
  let pokemons = [
    {
      name: 'Dracofeu',
      type: 'feu'
    },
    {
      name: 'Pikachu',
      type: 'electrique'
    },
    {
      name: 'Bulbizar',
      type: 'herbe'
    },
    {
      name: 'Carapuce',
      type: 'eau'
    }
  ];

  const titre = 'Mon Pokédex';

  res.render("pokemon", {
    pokemons: pokemons,
    title: titre
  });
});
```

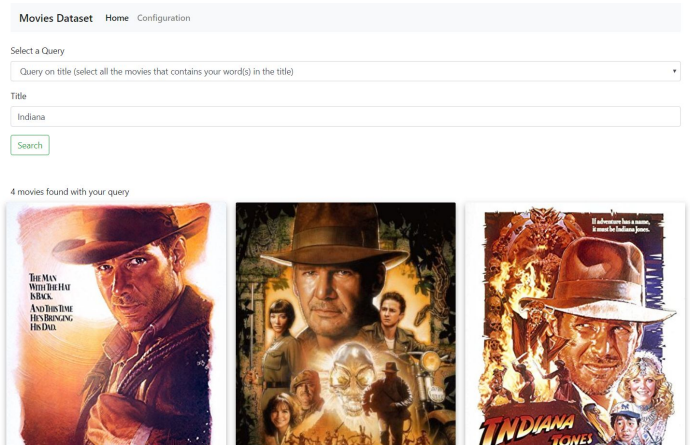
# Le moteur de template EJS

Exercice :

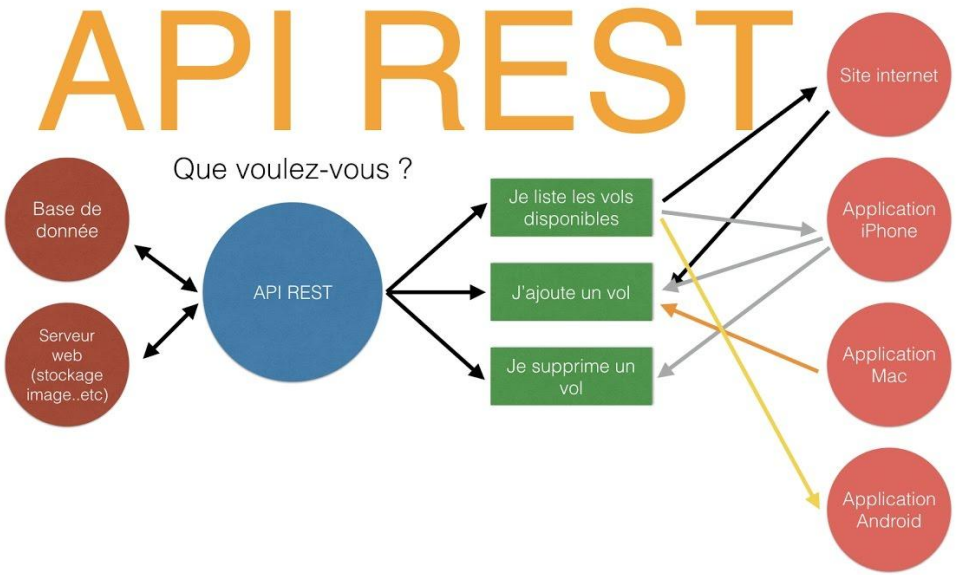
Créer une interface pour une application de film qui contient :

- Un header
- Un footer
- Afficher une liste de film

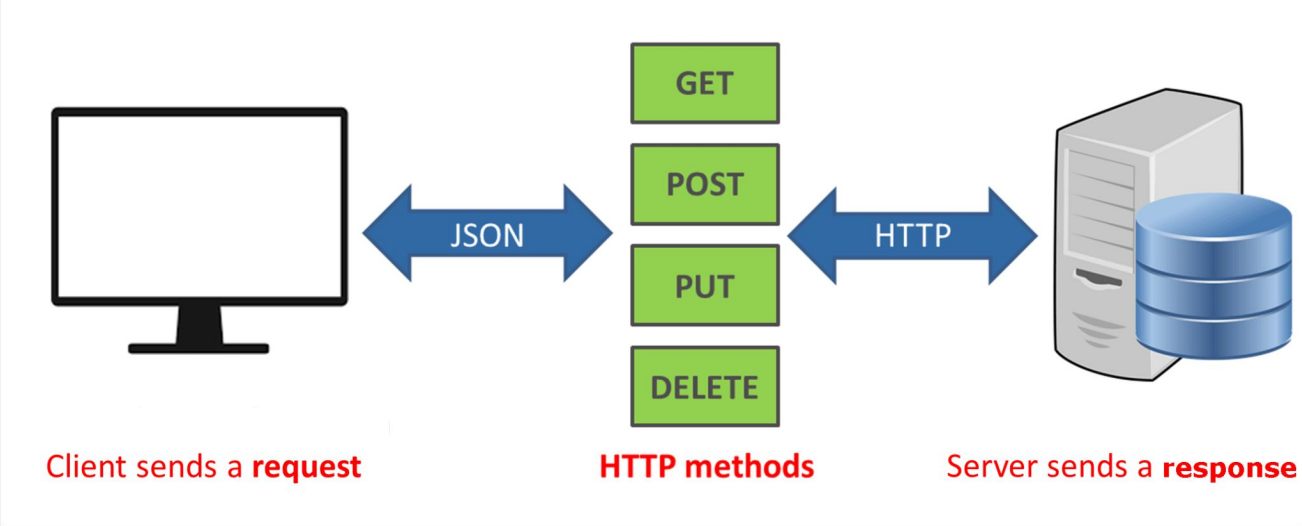
Plus : Afficher des films d'un certain genre



# API REST


API REST
Que voulez-vous ?
Base de donnée
Serveur web (stockage image..etc)
API REST
Je liste les vols disponibles
J'ajoute un vol
Je supprime un vol
Site internet
Application iPhone
Application Mac
Application Android

# API REST


GET
POST
PUT
DELETE
JSON
HTTP
Client sends a **request**
**HTTP methods**
Server sends a **response**

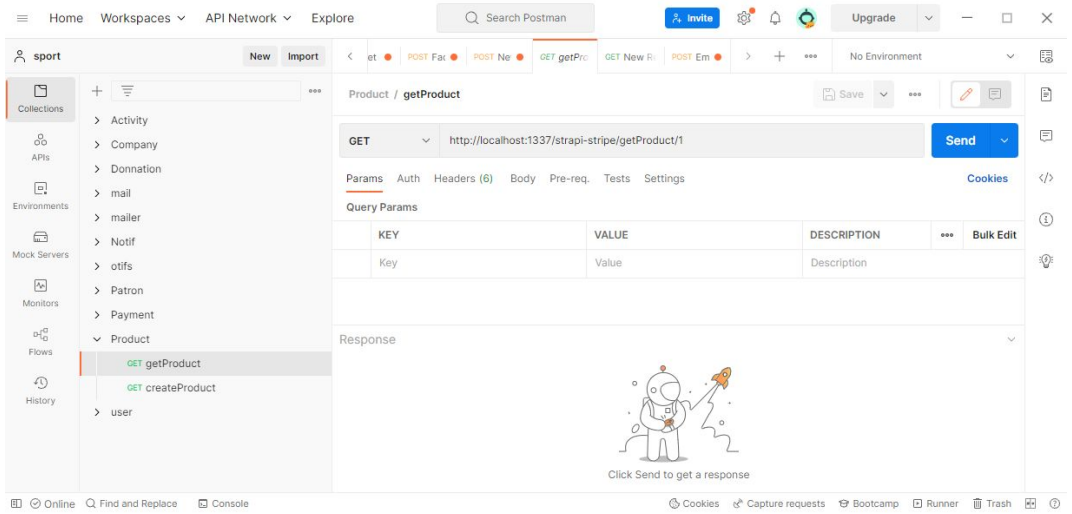
# API REST

Code HTTP les plus courant :

- 200 : succès de la requête
- 301 et 302 : redirection, respectivement permanente et temporaire
- 400 : La syntaxe de la requête est erronée
- 401 : utilisateur non authentifié
- 403 : accès refusé
- 404 : ressource non trouvée
- 500, 502 et 503 : erreurs serveur
- 504 : le serveur n'a pas répondu

# Postman

Postman : un outil pour tester votre Api

# API avec Express

Ajouter le format JSON à express:

```
app.use(express.json());
```

Retourner une réponse

```
return res.status(400).json({
    message: 'Something is not right',
    user   : user
});
```

# API avec Express

```
import express from "express";
const router = express.Router();

router.get('/', (req, res) => res.json({message: 'Allright !'}) );

router.post('/', (req, res) => {
  const { username } = req.body;
  let errors = [];

  console.log(username);

  if (!username) errors.push('Username required');

  if (errors.length > 0) res.status(400).json({errors : errors});

  res.json({message: 'Allright !'});
});

export default router;
```

# Exercice

Créer une API permettant de transmettre des films à l'aide d'un fichier JSON (Transmettre et recevoir).