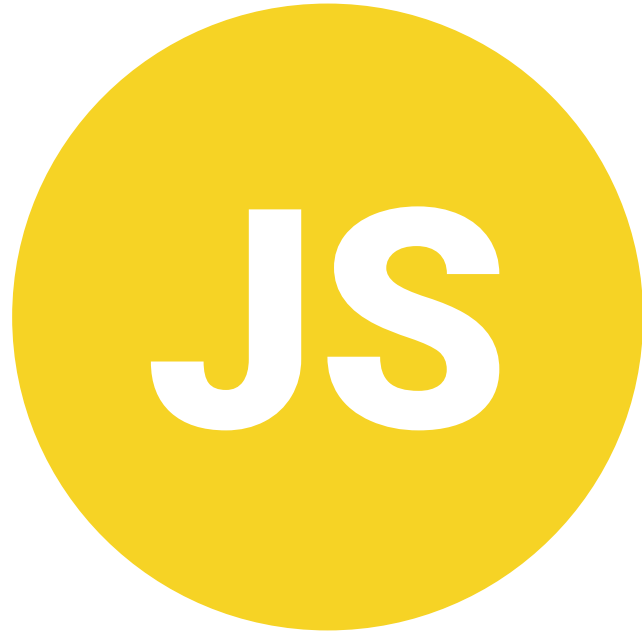


Bienvenue en cours de





Quentin Brunet

qboot

Developer @jolicode, France. I love trying new things and experiment, especially in the web dev world. ✨
Former @MMI_Troyes & @ESGI student.

👉 Quentin Brunet

👉 Développeur web à JoliCode

👉 GitHub : @qboot

👉 Email : pro.quentinbrunet@gmail.com



redirection.io

<JoliCode/>

Déroulé du cours



12 heures de cours



2h · Cours théorique



10h · Projet individuel

- 1h30 · “Bootstrap” du projet
- 3h · JavaScript côté backend (Node.js)
- 3h · JavaScript côté frontend (un peu de React)
- 2h30 · Temps pour finir / améliorer

Notation



Une note sur le projet



Rendu via GitHub



Barème (provisoire)

- **12 points** · Rendu final
- **3 points** · Feature bonus
- **3 points** · Régularité (suivi, push de commits)
- **2 points** · Qualité du code

Questions ?

Cours



Plan du cours

Intro.

1. Debug

2. Types

3. Casting

4. Déclaration de
variable

5. Fonctions

6. Scopes

7. Par valeur, par
référence

8. Promesses

9. async / await

10. this

11. class

12. Opérateurs

13. Destructuring

14. Conditions

15. Boucles

16. Object

17. array

18. Erreurs

19. Date

Intro. Le JavaScript, c'est quoi ?

👉 Créé en 1995 par Brendan Eich 🇺🇸 (👏 merci Wikipédia)

👉 Langage connu pour ~~animer des pages web~~

- créer des serveurs web (Node.js)
- créer des sites (React, Vue.js, Angular)
- créer des apps mobiles (React Native, Ionic)
- créer des apps Desktop (Electron)
- créer des outils en ligne de commande (CLI)
- créer des jeux (Unity)

Intro. Le JavaScript, c'est quoi ?

👉 Langage **asynchrone** et **mono threadé**

👉 Langage à **typage faible** (👏 TypeScript)

👉 **Compilation JIT** (à la volée)

- pas de compilateur (comme *gcc* pour le C/C++)
- pas interprété (comme le PHP, Python)
- mais **un moteur JavaScript** (v8 pour Chrome)

Intro. Le JavaScript, c'est quoi ?

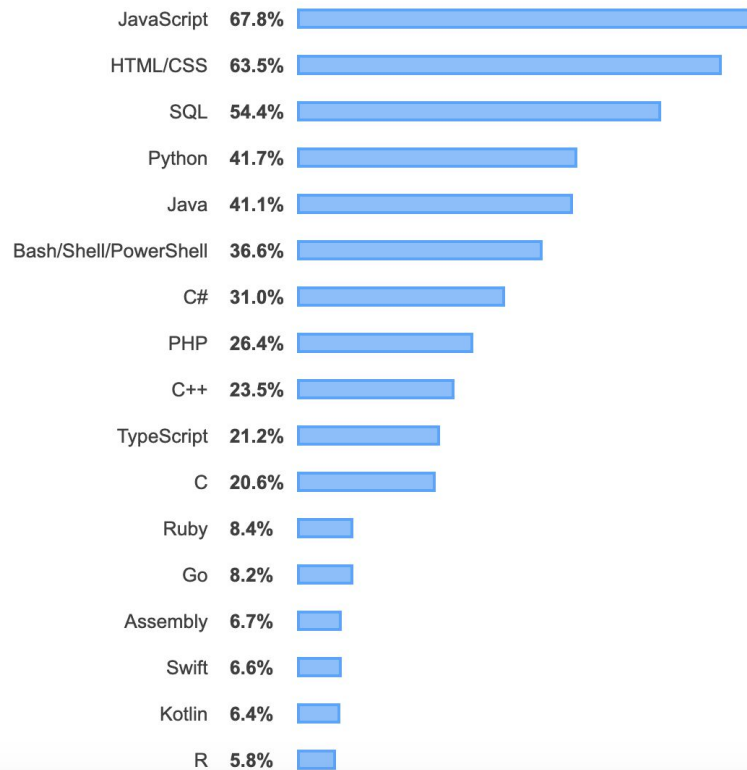
Pourquoi c'est cool ?

- 👉 On peut *presque* tout faire avec
- 👉 Grande communauté active (👏 StackOverflow)
- 👉 Courbe d'apprentissage rapide
- 👉 On a tous déjà de quoi faire du JS sur notre ordi
- 👉 Ça recrute !

Programming, Scripting, and Markup Languages

All Respondents

Professional Developers



Le Langage

1. Debug

👉 Un doute ?

On ouvre les Chrome Dev Tools de son navigateur et on teste.

- Windows / Linux : F12 ou Control+Shift+I
- Mac : Command+Option+I

(appuyer sur Escape pour ouvrir ou fermer la console)

👉 Le bon réflexe



```
console.log(myVar, myVar2, myVar3);
```

2. Types

😊 Facile ! Seulement 6 types en JavaScript

👉 5 types primitifs

- dont 3 classiques : Number, String, Boolean
- et 2 spéciaux : Null, Undefined

👉 1 type Object et ses types instanciables

- Function, Array, Date, RegExp

2. Types



// Types primitifs

```
let myNumber = 12;
```

// on peut utiliser au choix des '' ou des "" pour les strings

```
let myString = 'hello world';
```

```
let myBool = true;
```

```
let myUndef = undefined;
```

```
let myNull = null;
```

// Les autres types

```
let myObject = { name: 'Quentin' };
```

// déconseillé, mais on peut mixer plusieurs types dans un tableau

```
let myArray = [12, 4, 'yes', false];
```

```
let myDate = new Date();
```

2. Types



```
// Typage dynamique (possible, mais déconseillé)
```

```
let myVar = 12;  
myVar = { name: 'Quentin' };  
myVar = 'toto';
```

```
// Les template strings `` (ça c'est conseillé !)
```

```
let name = 'Quentin', rank = 3;
```

```
// concaténation (sans template string)
```

```
let welcomingMessage = 'Bonjour ' + name + ' ! Tu es classé : ' + rank + '.';
```

```
// avec template string
```

```
let welcomingMessage = `Bonjour ${name} ! Tu es classé : ${rank}.`;
```

```
let multilineString = `
```

```
    On peut aussi faire des strings
```

```
    sur plusieurs lignes via les template strings`;
```


3. Casting



// Conversions de String -> Number

```
const myStr = '123';  
const myInt = parseInt(myStr); // vaut 123  
  
const myStr = '3.14dnepe';  
const myFloat = parseFloat(myStr); // vaut 3.14
```

// Conversion de tout -> Boolean

```
const myNumber = 12;  
const myBool = !!myNumber; // typeof myBool === 'boolean'
```

4. Déclaration de variable



```
// avec var (déconseillé)
var var1 = 'hello world';

// avec let
let var2 = 'hello world';
var2 = 'another message'; // valid

// avec const
const var3 = 'hello world';
var3 = 'test'; // erreur : Uncaught TypeError: Assignment to constant variable.

// attention, avec "const" seule la référence ne peut être réassignée
// mais on peut tout à fait faire :
const myObject = { name: 'Quentin' };
myObject.description = 'Prof JS'; // valid
myObject = null; // invalid

const myArray = [];
myArray.push(123); // valid
```

5. Fonctions



```
// avec le mot clef "function"  
function myFunc() {  
    console.log('hello :');  
}  
  
// avec une "arrow function" (mieux)  
const myFunc = () => {  
    console.log('hello :');  
};  
  
myFunc();
```

5. Fonctions avec params



```
// avec des paramètres
const myFunc = (param1, param2) => {
  console.log(param1);
};

myFunc(); // ok, le console.log() affiche "undefined"
myFunc('test'); // ok, le console.log() affiche "test"
myFunc('test', 23); // ok, idem
myFunc('test', 23, 'oups') // ok, idem

// avec le "rest operator"
const myFunc = (name, ...otherParams) => {
  console.log(`Nom: ${name}`);
  console.log(otherParams);
};

myFunc('test', 23, 'oups'); // ok
// affiche : "test"
// affiche les params restants dans array : [23, 'oups']
```

5. Fonctions avec params



```
// avec des valeurs par défaut
const myFunc = (param1, debug = false) => {
  if (debug) {
    console.log(param1);
  }
};

myFunc('ça va toujours ?', true); // affiche
myFunc('ça va toujours ?'); // n'affiche rien

// En JS, plutôt que de passer plein de params
// Il est parfois préférable de passer un objet directement

const myFunc = ({ param1, param2, param3 = 'default' }) => {
  // do something...
};

// Ça permet de passer les params dans n'importe quel ordre
// C'est un peu l'équivalent des named parameters en Python

myFunc({ param2: 24, param1: 23 });
```

5. Fonctions avec retour



// Les retours

```
const returnSomething = () => {  
  return 'something';  
};
```

```
console.log(returnSomething()); // affiche : something
```

// Version courte

```
const returnSomething = () => 'something';
```

```
console.log(returnSomething()); // affiche : something
```

// En JS, on ne peut pas retourner plusieurs valeurs (comme en Python par ex.)

// Mais on peut contourner cette limitation en retournant un array

```
const returnMultiple = () => {  
  const name = 'Quentin';  
  const rank = 3;
```

```
  return [name, rank]; // ok, on aura accès aux valeurs dans un array  
};
```

5. Fonctions anonymes



// Utile pour faire des callbacks

```
const waitThenCallback = callback => {  
  // ici on a une première fonction anonyme  
  setTimeout(() => {  
    callback(); // executer la callback  
  }, 2000); // attendre 2 secondes  
};
```

// ici on a une deuxième fonction anonyme

```
waitThenCallback(() => {  
  console.log('hello :');  
});
```

// Autre usage: Fonction IIFE (Expression de fonction invoquée immédiatement)

```
(( ) => {  
  console.log('Coucou, je suis immédiatement invoquée :p');  
})(); // bien noter les parenthèses ici
```

6. Scopes



// Scope global

```
const globalVar = 'youhou';
```

```
const myFunc = () => {  
  console.log(globalVar);  
};
```

```
myFunc(); // affiche "youhou"
```

```
console.log(globalVar); // affiche "youhou"
```

// Scope local

```
const myFunc = () => {  
  const localVar = 'rip';  
  console.log(localVar);  
};
```

```
myFunc(); // affiche "rip"
```

```
console.log(localVar); // erreur: Uncaught ReferenceError: localVar is not defined
```


7. Passage par valeur, par référence



```
// Un type primitif est TOUJOURS passé par valeur  
let i = 0;  
  
const incrementByValue = i => {  
  i = i + 1;  
};  
  
incrementByValue(i);  
console.log(i); // affiche 0  
  
// Un object ou un array est TOUJOURS passé par référence  
const config = { i: 0 };  
  
const incrementByRef = config => {  
  config.i = config.i + 1;  
};  
  
incrementByRef(config);  
console.log(config.i); // affiche 1
```

8. Promesses



```
// Utilisées pour réaliser des traitements asynchrones

const myFunc = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('2. Je loggue depuis myFunc');
    resolve();
  }, 2000); // attendre 2 secondes
});

myFunc.then(() => {
  console.log('3. Je loggue depuis le then');
});

console.log('1. Je loggue depuis le contexte global');

/*****
1. Je loggue depuis le contexte global
(attente de 2 secondes)
2. Je loggue depuis myFunc
3. Je loggue depuis le then
*****/

// si c'est le reject() qui est appelé
// on saute tous les "then" éventuels et on peut catcher l'erreur
myFunc.catch(() => {
  console.log('Erreur catchée');
});
```

9. async / await

```

const myFunc = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('2. Je loggue depuis myFunc');
      resolve();
    }, 2000); // attendre 2 secondes
  });
};

console.log('1. Je loggue avant le await');

try {
  await myFunc();
} catch (e) {
  // équivalent du .catch()
}

console.log('3. Je loggue après le await');
})();
```



```

// Pareil que les then/catch, mais en plus beau ❤️
// La règle :
// - await permet d'attendre la résolution d'une fonction async
// - await doit toujours être dans un contexte async (d'où l'IIFE)
// - il faut englober l'appel à la fonction dans un try/catch
//   si on souhaite catch les erreurs
```

```

/*****
1. Je loggue avant le await
(attente de 2 secondes)
2. Je loggue depuis myFunc
3. Je loggue après le await
*****/
```

10. this



Doc : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/L_op%C3%A9rateur_this



Dépend du contexte appelant



Contexte global

- dans Node.js : `this === global`
- dans le navigateur : `this === window`



Contexte d'une fonction

- c'est plus compliqué 🤔

10. this : call(), apply() et bind()



```
// 1. Avec le mot clef "function"

// 1.1 L'appel à la fonction est simple
// this correspond à sa valeur dans le contexte englobant

function myFunc() {
  return this;
}

myFunc(); // vaut global ou window

// 1.2 L'appel à la fonction se fait
// avec .call() ou .apply()

const obj = { name: 'Quentin' };

myFunc.call(obj); // vaut { name: 'Quentin' }
myFunc.apply(obj); // vaut { name: 'Quentin' }
```



```
// 1.3 Une nouvelle fonction est créée avec .bind()
// this correspond TOUJOURS à la valeur du paramètre
// de .bind()

const myBindedFunc = myFunc.bind({ name: 'Someone else' });

myBindedFunc();
// vaut {name: "Someone else"}
myBindedFunc.call({ name: 'Quentin' });
// vaut {name: "Someone else"}
myBindedFunc.apply({ name: 'Quentin' });
// vaut {name: "Someone else"}

// 2. Avec une fonction fléchée, c'est + facile
// this correspond TOUJOURS à sa valeur dans le
// contexte englobant

const myFunc = () => {
  return this; // vaut global ou window
};
```

11. class



Juste du sucre syntaxique



// avec class

```
class Car {  
  constructor() {  
    this.wheels = 4;  
  }  
  
  addWheels(n) {  
    this.wheels += n;  
  }  
}  
  
let car = new Car();  
car.addWheels(2);  
  
console.log(car.wheels); // affiche : 6
```



// sans class

```
function Car() {  
  this.wheels = 4;  
}  
  
function addWheels(n) {  
  this.wheels += n;  
}  
  
let car = {};  
  
Car.call(car);  
addWheels.call(car, 2);  
  
console.log(car.wheels); // affiche : 6
```

12. Opérateurs

👍 On retrouve comme dans la plupart des langages : + - * / %



```
// Types primitifs
```

```
// initialisation
```

```
let i = 0;
```

```
// assignation
```

```
i = 1; // i vaut 1
```

```
i += 1; // équivalent à i = i + 1, i vaut 2
```

```
i++; // équivalent à i = i + 1, i vaut 3
```

```
--i; // équivalent à i = i - 1, i vaut 2
```

```
// avec les strings
```

```
let msg = 'Hello';
```

```
msg += ' world !';
```

12. Opérateur spread



```
// Type object

const obj1 = { name: 'Quentin' };
const obj2 = obj1; // par référence

obj2.name = 'Greg';
console.log(obj1.name); // affiche : Greg

// Type array

const arr1 = [];
const arr2 = arr1; // par référence

arr2.push(10);
console.log(arr1); // affiche : [10]

// Le spread operator
// Un moyen de contourner le passage par référence pour "cloner" un array/object

const obj = { name: 'Quentin' };
const objClone = { ...obj }; // par valeur

objClone.name = 'Greg';
console.log(obj.name); // affiche : Quentin
```


13. Destructuring



```
// Parfois, on peut ne vouloir qu'une partie d'un object/array  
// Pour cela on va affecter par décomposition
```

```
const person = { name: 'Quentin', gender: 'H', level: 32 };  
const { name, gender } = person; // destructuring  
console.log(name); // affiche : Quentin
```

```
// on peut aussi utiliser le rest operator
```

```
const person = { name: 'Quentin', gender: 'H', level: 32 };  
const { name, ...allPropsWithoutName } = person; // destructuring + rest operator  
console.log(allPropsWithoutName); // affiche : { gender: 'H', level: 32 }
```

```
// on peut aussi renommer une variable déstructurée
```

```
const person = { uglyApiName: 'Quentin', gender: 'H', level: 32 };  
const { uglyApiName: name } = person; // destructuring + renommage  
console.log(name); // affiche : Quentin  
console.log(uglyApiName); // erreur : Uncaught ReferenceError: uglyApiName is not defined
```

14. Conditions : Opérateurs de comparaison



```
// Ce qui vaut false
false, 0, null, undefined, ''

// Ce qui vaut true
true, 1, 32, -392, ' ', '0', 'false', 'hello', [], {}

// Opérateurs de comparaison

|| ou
&& et
== égal
!= non égal
< inférieur
> supérieur
<= inférieur ou égal
>= supérieur ou égal
=== égal strict
!== non égal strict
```

14. Conditions : Opérateurs instanceof, typeof



// Opérateur instanceof

```
class Car {}  
const car = new Car();
```

```
console.log(car instanceof Car); // vaut true
```


// Opérateur typeof

// Renvoie un des 6 types vus en début de cours, sauf pour null

```
console.log(typeof true); // affiche "boolean"  
console.log(typeof 43); // affiche "number"  
console.log(typeof 'blabla'); // affiche "string"  
console.log(typeof []); // affiche "object"  
console.log(typeof {}); // affiche "object"  
console.log(typeof null); // ⚠ affiche "object"  
console.log(typeof undefined); // affiche "undefined"
```

14. Conditions : if, else

```


// condition if simple
const valid = 32;

if (valid) {
  console.log("gg 🍌!");
}

// différence entre == et ===

/* à éviter */
if (null == undefined) {} // vaut true
/* recommandé */
if (null === undefined) {} // vaut false

// condition avec else
const state = 'errored';

if ('successful' === state) {
  console.log('Bravo');
} else if ('warning' === state) {
  console.log('Ça passe encore');
} else if ('errored' === state) {
  console.log('Alerte rouge !!!');
} else {
  console.log('État inconnu');
}

```

```


// condition switch
const state = 'errored';

switch (state) {
  case 'successful':
    console.log('Bravo');
    break;
  case 'warning':
    console.log('Ça passe encore');
    break;
  case 'errored':
    console.log('Alerte rouge !!!');
    break;
  default:
    console.log('État inconnu');
}

// ternaire
const fetching = true;

const content = fetching
  ? 'Merci de patienter...'
  : 'Mon super contenu';

```

15. Boucles : for



```
// boucle for simple
for (let i = 0; i < 5; i++) {
  console.log(i);
}

let i = 0;
// on peut omettre des parties du for
for (;;) {
  if (0 === i) {
    continue; // l'instruction continue sert à passer à la prochaine itération directement
  }

  console.log(i);

  if (3 === i) {
    break; // l'instruction break sert à sortir d'une boucle
  }
}

// Quel sera l'affichage ?
```

15. Boucles : while, do...while



```
// boucle for imbriquée
for (let i = 0; i < 5; i++) {
  for (let j = 0; j < 2; j++) {
    break 2; // ici on sort des 2 niveaux d'imbrication
  }
}

// boucle while
let i = 0;

while (i < 5) {
  console.log(i)
  i++;
}

// boucle do...while (exécutée au moins une fois)
let i = 0;

do {
  console.log(i);
} while (0 !== i);
```

15. Boucles : for...in, for...of



```
// boucle for...in (pour les objects)
const obj = { name: 'Quentin', rank: 3 };

for (const key in obj) {
  console.log(`${key} => ${obj[key]}`);
}

// boucle for...of (pour les arrays)
const arr = ['banana', 'apple', 'strawberry'];

for (const value of arr) {
  console.log(value);
}

// récupérer l'index dans une boucle for...of
for (const [index, value] of arr.entries()) {
  console.log(`${index} => ${value}`);
}
```

16. Manipulation d'object

👉 Doc : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object



```
const obj = {};  
  
obj.name = 'Quentin'; // ajouter une propriété  
console.log(obj); // affiche : {name: "Quentin"}  
  
delete obj.name; // supprimer une propriété  
console.log(obj); // affiche : {}  
  
// les méthodes utiles pour transformer en array  
  
const obj = { name: 'Quentin', msg: 'Allez, encore un petit effort' };  
  
const keys = Object.keys(obj); // vaut : ["name", "msg"]  
const values = Object.values(obj); // vaut : ["Quentin", "Allez, encore un petit effort"]  
const entries = Object.entries(obj); // vaut : [["name", "Quentin"], ["msg", "Allez, encore un petit effort"]]
```


17. Manipulation d'array

👉 Doc : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array



```
const arr = [];  
  
arr.push('Quentin'); // ajouter une valeur  
  
console.log(arr); // affiche : ["Quentin"]  
console.log(arr.length); // affiche la taille du tableau : 1  
  
const pos = 0;  
const removedArr = arr.splice(pos, 1); // supprimer un élément à la position pos  
  
console.log(removedArr); // contient les valeurs supprimées, affiche : ["Quentin"]  
console.log(arr); // affiche : []  
  
// les méthodes utiles pour manipuler un array  
  
arr.pop(); // supprimer le dernier élément  
arr.shift(); // supprimer le premier élément  
arr.push('Quentin'); // ajouter à la fin  
arr.unshift('Laureen'); // ajouter au début  
arr.splice(pos, 1); // supprimer un élément à la position pos
```

17. Manipulation d'array : map(), filter() et reduce()

```
const people = [
  { name: 'Quentin', gender: 'H', rank: 2, winCount: 2 },
  { name: 'Marion', gender: 'F', rank: 1, winCount: 14 },
  { name: 'Greg', gender: 'H', rank: 3, winCount: 0 },
];

// Fonction map()

const happyPeople = people.map(person => ({
  ...person,
  mood: 'happy',
}));

console.log(happyPeople[0]);
// affiche : {name: "Quentin", gender: "H", rank: 2, winCount: 2, mood: "happy"}

// Fonction filter()

const men = people.filter(person => 'H' === person.gender);

console.log(men.length); // affiche : 2

// Fonction reduce()

// acc = accumulateur et cur = item courant
const totalWinCount = people.reduce((acc, cur) => (acc + cur.winCount), 0);

console.log(totalWinCount); // affiche : 16
```

18. Erreurs



```
// erreur simple catchée
try {
  // à noter Error() === new Error()
  throw Error('Je suis une erreur');
} catch (e) {
  console.log(e);
}

// erreur custom
class ValidationError extends Error {}

// erreur custom catchée
try {
  // le mot-clef "new" est requis ici
  throw new ValidationError('Je suis une erreur de validation');
} catch (e) {
  if (e instanceof ValidationError) {
    console.log("Gérer l'erreur");
  }
}
```

19. Date

👉 Doc : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Date

📖 Librairie recommandée : <https://github.com/date-fns/date-fns>



```
const date = new Date();

console.log(date);
// affiche : "Sun Sep 20 2020 21:26:24 GMT+0200 (heure d'été d'Europe centrale)"

date.setDate(date.getDate() + 60); // on incrémente la date de 60 jours

console.log(date);
// affiche : "Thu Nov 19 2020 21:26:24 GMT+0100 (heure normale d'Europe centrale)"

date.toLocaleString(); // vaut : "20/09/2020 à 21:26:24"
date.toDateString(); // vaut : "Sun Sep 20 2020"
date.toISOString(); // vaut : "2020-09-20T19:26:24.121Z"

// Et bien + sur la documentation
```

👏 La doc Mozilla 👏



👉 Pour plus d'infos, des précisions, des exemples, etc.

<https://developer.mozilla.org/fr/docs/Web/JavaScript>

install.sh

Installation

👉 Git + un compte GitHub

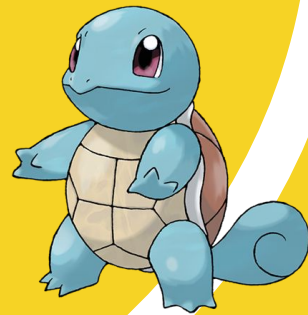
👉 Node.js · en LTS : <https://nodejs.org/en/>

👉 VS Code : <https://code.visualstudio.com/>

👉 yarn · Faire un `npm install -g yarn` pour l'installer

🤔 et c'est tout 🤖

Projet



#. Déroulé d'une partie

👉 Chaque joueur est représenté par un onglet de votre navigateur

👉 A chaque connexion, on ajoute un joueur au tableau des joueurs

- Shape de l'objet player : { socket, name, pokemon: null }

👉 Dès que 2 joueurs sont connectés, la partie commence

- Si un autre joueur essaie de se connecter, lui envoyer un message "Connexion refusée" et fermer la connexion
- Si un des 2 joueurs se déconnecte pendant une partie (donc si vous rafraîchissez une page de votre navigateur), supprimer le joueur du tableau des joueurs, supprimer le pokémon du joueur restant et lui indiquer que son adversaire s'est déconnecté

#. Déroulé d'une partie

👉 Lorsqu'une partie commence

- Attribuer aléatoirement à chacun des 2 joueurs un pokémon parmi les 5 du fichier pokemons.json
- Décider aléatoirement qui sera le premier joueur
- Envoyer à chaque joueur un objet contenant :
 - you · le joueur (un objet avec son nom et son pokémon)
 - opponent · son adversaire (idem)
 - turn · à qui c'est de jouer (soit "you" soit "opponent")

#. Déroulé d'une partie

👉 A chaque tour

- Côté client : le joueur joue une des 4 actions de son pokémon (on renvoie l'id de l'action au serveur)
- Côté serveur : on reçoit l'événement avec l'id de l'action
 - Mettre à jour les hp du pokémon de l'adversaire
 - Vérifier que le pokémon n'est pas KO ($hp \leq 0$) // sinon fin de partie
 - Mettre à jour la variable turn
 - Renvoyer un objet mis à jour aux 2 joueurs avec : you, opponent, turn

#. Déroulé d'une partie

👉 A la fin de la partie

- Envoyer à chaque joueur un objet contenant :
 - you · le joueur (un objet avec son nom et son pokémon)
 - opponent · son adversaire (idem)
 - win · un booléen qui vaut true si le joueur a gagné, false sinon