R4.01 Architecture Logicielle

R4.01 Architecture Logicielle

Publication date 2023

Table of Contents

List of Figures

List of Tables

Chapter 1. Présentation de la ressource

1. Programme national

Le programme national définit un cadre général que nous allons spécialiser

Descriptif

> L’objectif de cette ressource est de présenter des composants de la programmation qui peuvent être utilisés dans plusieurs domaines.

Savoirs de référence étudiés

- Patrons d’architecture (par ex. : Modèle Vue Contrôleur, Model View ViewModel), ...)
- Utilisation de briques logicielles, d’interfaces de programmation, de bibliothèques tierces
- Développement de services web

Nous allons sans un premier temps concevoir des services REST (Representational State Transfer) et les implémenter sous forme d'une architecture de micro services, puis dans un second temps nous allons mettre en place une communication full-duplex par-dessus une connexion TCP, cette approche permet au serveur de notifier des événements aux clients.

2. L'architecture REST

Vous utilisez sans le savoir des services REST au travers de vos applications favorites :

Twitter

> https://developer.twitter.com/en/docs/api-reference-index

Instagram

> https://developers.facebook.com/docs/instagram

Les applications android, web, les clients lourds sollicite des services REST, ces services renvoyant des données bruts pouvant être traitées les différents langages des applications. Les données sont principalement renvoyées au format JSON (JavaScript Object Notation) ou XML (Extensible Markup Language). Le format JSON sera choisi pour nos services.

A titre d'exemple l'URL suivante permet d'obtenir des informations sur une bière aléatoire : https://api.punkapi.com/v2/beers/random. au format JSON.

Les attentes d'une API REST doivent répondre aux contraintes architecturales suivante :

Client–serveur

> Nous allons développer dans ce module la partie serveur en nodeJS, la partie cliente sera développez dans les ressources R4.Real.10:Complément web en JS (JavaScript) et R4.Real.11 : Développement pour applications mobiles (Kotlin). Nous utiliseront comme protocole de communication HTTP (Hypertext Transfer Protocol).

Sans état

> Chaque requête devra contenir l'ensemble des informations permettant d'y répondre, les connexions ne seront pas persistantes. L'usage des cookies est par exemple déconseillée et le passage d'un jeton est à préférer.

Avec mise en cache

Les clients et les serveurs intermédiaires peuvent mettre en cache les réponses. Les réponses doivent donc, implicitement ou explicitement, se définir comme pouvant être mise en cache ou non. Dans un soucis de simplification, nous n'implémenterons pas cette fonctionnalité et conserverons le comportement par défaut.

En couches

Le client ne doit pas savoir, si il sollicite un serveur intermédiaire ou un un serveur final. Nous développerons séparément nos services avant de les cacher derrière une passerelle d'API (Application Programming Interface) .

Interface uniforme

Les services REST doivent offrir une interface uniforme, un même service sous une même URL (Uniform Resource Identifier) doit pouvoir offrir différents types de contenus, nous allons offrir uniquement du JSON. Les quatre contraintes de l'interface uniforme sont les suivantes :

Identification des ressources dans les requêtes

La requête peut spécifier le type de ressource attendu

Manipulation des ressources par des représentations

Chaque représentation d'une ressource fournit suffisamment d'informations au client pour modifier ou supprimer la ressource.

Messages auto-descriptifs

Chaque message contient assez d'information pour savoir comment l'interpréter.

HATEOAS (Hypermédia comme moteur d'état de l'application)

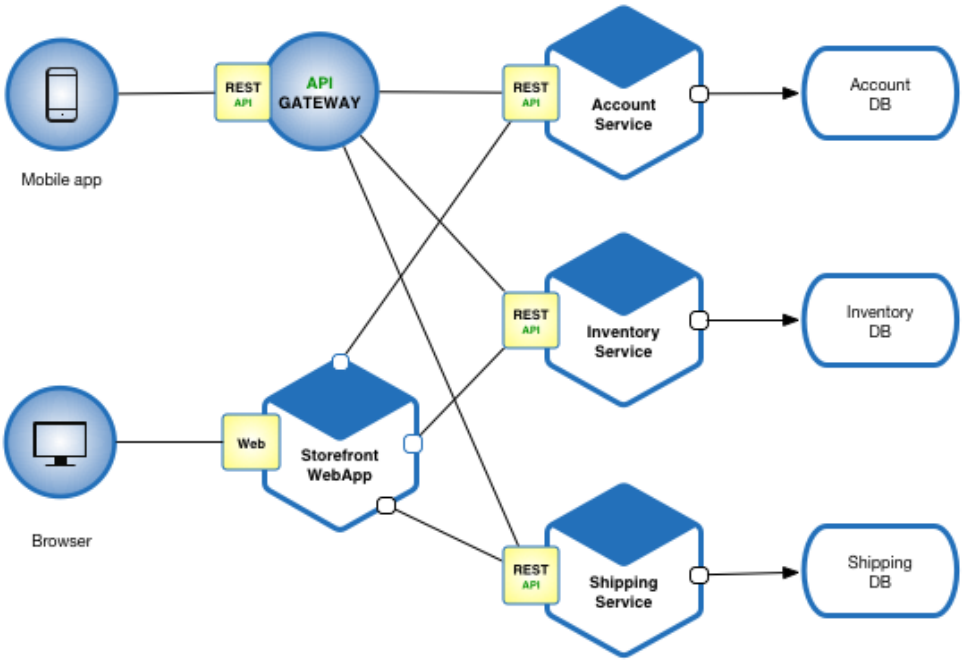
le client doit être en mesure d'utiliser dynamiquement les hyperliens fournis par le serveur pour découvrir toutes les autres actions possibles et les ressources dont il a besoin pour poursuivre la navigation.

3. Les microservices

Nous allons avoir des services REST implémentés sous forme de microservices. Les microservices s'opposent aux architectures monolithiques. Chaque service doit être le plus simple possible et ne réaliser qu'une fonction, il doit être facilement testables. Les microservices doivent-être élastique (scalable), résiliant (la défaillance d'un service ne doit pas impacter les autres), composable, minimal (les entités et fonctionnalités sont fortement liées), complet (offrir l'ensemble des fonctionnalités).

Nos microservices communiqueront de manière asynchrone en utilisant HTTP.

Figure 1.1. Architecture microservices (https://microservices.io/)



4. Les websocket

Nous allons créer un chat lié à notre API REST permettant à plusieurs client d'échanger des messages.

5. Les technologies utilisées

Nous allons réaliser nos services en nodeJS (https://nodejs.org/fr/) une plateforme logicielle libre en JavaScript, orientée vers les applications réseau évènementielles. nodeJS repose sur JavaScript mais il est a noter que la plateforme ne supporte pas la totalité du langage (https://node.green/), nous utiliserons la version 18 compatible à 99%.

Pour développer la partie serveur HTTP, nous utiliserons le framework HAPI(https://hapi.dev/).

Nous utiliserons le moteur de base de données SQLite et l'ORM prisma (https://www.prisma.io/).

Pour la partie test serveur, nous utiliserons lab comme moteur (https://github.com/hapijs/lab) et code comme langage de description (https://github.com/hapijs/code).

Pour les websockets, nous utiliserons socket.io (https://socket.io/).

Nous utiliserons également gitlab pour l'automatisation des tests.

6. Les services web et données utilisées

Nous allons en partant de deux sources de connaître les parking disponibles et les restaurants à proximité et le tout avec un système de favoris.

Nous utiliserons de sources de données de l'open-data de Nantes :

La disponibilité des parkings

> https://data.nantesmetropole.fr/explore/dataset/244400404_parkings-publics-nantes-disponibilites/information/?disjunctive.grp_nom&disjunctive.grp_statut

Les restaurants

https://data.nantesmetropole.fr/explore/dataset/793866443_restaurants-en-loire-atlantique%40loireatlantique/information/

7. Volumes et plannings

Voici le planning indicatif :

Table 1.1. Planning prévisionnel

semaine	CM	TD	Objectif
3	1	3	Découverte de nodeJS
4	1	3	Création d'un Proxy
5	1	3	Création d'un service avec Impotation des données
6	1	3	Mise en place d'une passerelle
7		3	Authentification
9		2	WebSocket
10		2	Evaluation

8. Liens avec la SAE

Lors de la SAE du S4 (SAÉ 4.Real.01 : Développement d’une application complexe), vous devrez développer des services REST et des clients en JS et en android.

Chapter 2. Découverte de nodeJS

JavaScript est un langage interprété, faible typé, par références orienté objet à prototype. L'approche prototype est une différence fondamentale,les objets de base ne sont pas des instances de classes. En outre, les fonctions sont des objets de première classe. Le langage supporte le paradigme objet, impératif et fonctionnel.

Dans un terminal taper les commandes suivantes pour obtenir la dernière LTS de nodeJS.

```
npm.sh

hash -r
node -v
```

Vous pourrez exécuter les exemples suivants en utilisant node nom_du_script.

1. Déclaration des variables

1.1. Cours

Les variables globales sont définies sans mot clef, elles sont attachées au scope global, il est déconseillé de les utiliser.

```
g = 10
console.log(g, typeof g)
//10 number
```

Il est possible de restreindre le langage en utilisant le mode strict, dans ce mode les variables globales ne seront plus utilisées.

```
"use strict"
//g=10 => ReferenceError: g is not defined
```

Vous trouverez sur Internet beaucoup d'exemples reposants sur l'utilisation de var, son utilisation reste déconseillée, les déclarations de variables sont traitées avant que le code soit exécuté, quel que soit leur emplacement dans le code. La portée d'une variable déclarée avec var est le contexte d'exécution courant, c'est-à-dire : la fonction qui contient la déclaration ou le contexte global si la variable est déclarée en dehors de toute fonction.

```
"use struct"
console.log(a, typeof a)
//undefined undefined
var  a = 44
console.log(a, typeof a)
//44 number
```

Nous utiliserons donc const pour les immuables et let pour les muable, attention pour les objets, const ne fixe que la référence.

```
"use strict"
//c est un nombre constant.
const c = 10
console.log(c, typeof c)
//10 number
//c = 11 => TypeError: Assignment to constant variable

let v = 10
v = 11
console.log(v, typeof v)
//11 number

v="hello"
console.log(v, typeof v)
//hello string
```

```
v= function (p) {
    console.log(p)
}
console.log(v, typeof v)
//[Function: v] function
v(10)
//10

console.log(console)
//Object [console] {
//   log: [Function: log],
//...
console.log(console.log)
//[Function: log]
console.log(console.log("hello"))
//hello
//undefined
```

Comme énoncé précédemment, il est possible de créer des objets sans classe, voici un exemple d'objet littéral.

```
const clio = {
    nom: "Clio",
    marque : "Renault",
    demarree : false,
    equipage : [],
    demarre : function (){
        this.demarree = true
    },
    stop : function (){
        this.demarree = false
    },
    charge : function (passager) {
        this.equipage.push(passager)
    }
}
console.log(clio, typeof clio)
//{
//  nom: 'Clio',
//  marque: 'Renault',
//  demarree: false,
//  equipage: [],
//  demarre: [Function: demarre],
//  stop: [Function: stop],
//  charge: [Function: charge]
//} object

clio.demarre()
console.log(clio)
//{
//  nom: 'Clio',
//  marque: 'Renault',
//  demarree: true,
//  equipage: [],
//  demarre: [Function: demarre],
//  stop: [Function: stop],
//  charge: [Function: charge]
//}
clio.charge("Raoul")
clio.charge({nom: "neuneu", poids : 28});
console.log(clio)
//{
//  nom: 'Clio',
//  marque: 'Renault',
//  demarree: true,
//  equipage: [ 'Raoul', { nom: 'neuneu', poids: 28 } ],
//  demarre: [Function: demarre],
//  stop: [Function: stop],
//  charge: [Function: charge]
//}
console.log(clio.equipage[1].nom)
```

```
//neuneu

clio.poids = 1000
console.log(clio)
//{
//  nom: 'Clio',
//  marque: 'Renault',
//  demarree: true,
//  equipage: [ 'Raoul', { nom: 'neuneu', poids: 28 } ],
//  demarre: [Function: demarre],
//  stop: [Function: stop],
//  charge: [Function: charge],
//  poids: 1000
//}

clio.decharge = function () {
    this.equipage = []
}
clio.decharge()
//{
//  nom: 'Clio',
//  marque: 'Renault',
//  demarree: true,
//  equipage: [],
//  demarre: [Function: demarre],
//  stop: [Function: stop],
//  charge: [Function: charge],
//  poids: 1000,
//  decharge: [Function (anonymous)]
//}

console.log(clio.prototype)
//undefined

console.log(clio)
```

1.2. Exercices

1.2.1. Exercice 01

Créer un objet nommé classe définie par son nom et les étudiants qui la compose, les étudiants seront au nombre de trois et seront défini par un nom et une note. Le fichier devra posséder l'extension mjs (module javascript).

1.2.2. Exercice 02

A la fin, le l'exercice précédant, rajouter

```
export default classe //si classe est l'objet litteral
```

Dans nouveau fichier, importer la classe de l'exercice précédant

```
import classe from './exercice01.mjs'
```

Rajouter et tester deux méthodes l'une permettant de rajouter un étudiant, l'autre de supprimer l'ensemble des étudiants.

2. Les prototypes

Nous allons brièvement, introduire la notion de prototype.

2.1. Cours

Bien que le mot clef "class" soit disponible en JS, ce n'est qu'un sucre syntaxique. Exception faire des objet littéraux, chaque objet possède une propriété privée qui contient un lien vers un autre objet appelé le prototype. Ce

prototype possède également son prototype et ainsi de suite, jusqu'à ce qu'un objet ait null comme prototype. Par définition, null ne possède pas de prototype et est ainsi le dernier maillon de la chaîne de prototype. La majorité des objets JavaScript sont des instances de Object qui est l'avant dernier maillon de la chaîne de prototype.

Les fonctions sont plus que des fonctions, ce sont aussi des constructeurs, elles dispose d'un this. Chaque fonction à par défaut un prototype vide mais qui peut-être enrichi.

```
const Etudiant = function (nom='John Do',note=0){
    this.nom = nom
    this.note = note

    this.setNote = function (note) {
        this.note = note
    }

}

const etudiant1 = new Etudiant()
console.log(etudiant1, typeof etudiant1, etudiant1 instanceof Etudiant, etudiant1 instanceof Object)
//Etudiant { nom: 'John Do', note: 0, setNote: [Function (anonymous)] } object true true
etudiant1.setNote(10)
console.log(etudiant1)
//Etudiant { nom: 'John Do', note: 10, setNote: [Function (anonymous)] }

etudiant1.age = 20;
console.log(etudiant1)
//Etudiant {
//   nom: 'John Do',
//   note: 10,
//   setNote: [Function (anonymous)],
//   age: 20
// }

const etudiant2 = new Etudiant("etudiant2",20)
console.log(etudiant2)
//Etudiant {
//    nom: 'etudiant2',
//        note: 20,
//        setNote: [Function (anonymous)]
//}

//Ajout de l'age au prototype
Etudiant.prototype.age = 10

console.log(etudiant1)
//Etudiant {
//   nom: 'John Do',
//   note: 10,
//   setNote: [Function (anonymous)],
//   age: 20
// }
console.log(etudiant2)
//Etudiant {
//   nom: 'etudiant2',
//   note: 20,
//   setNote: [Function (anonymous)]
// }

const etudiant3 = new Etudiant("etudiant3",10)
console.log(etudiant3)
//Etudiant {
//   nom: 'etudiant3',
//   note: 10,
//   setNote: [Function (anonymous)]
// }
```

Ce nous venons d'apprendre, un prototype est un objet à partir duquel on crée de nouveaux objets. Dans notre exemple Etudiant est prototype qui va nous permettre de créer des objets comme `etudiant1`.

Il est possible s'enrichir chaque objet avec des méthodes propres, de même il est possible d'enrichir le prototype pour que chaque instance puisse disposer de l'enrichissement.

En JS tous les attributs et les méthodes sont publiques. Il existe un constructeur à la racine : `Object` (`https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object`).

```
"use strict"

//un constructeur comme début de la chaine des prototypes
console.log(Object);
//[Function: Object]
console.log(Object.prototype) //un attribut de la fonction constructrice
//[Object: null prototype] {}
console.log(Object.getPrototypeOf(Object)) //Le prototype
//{}

Object.prototype.age = 10;
//un attribut prototype pour enrichir les prototypes

const Etudiant = function (nom,note) {
    this.nom = nom || 'John Do'
    this.note = note || 0
}

const etudiant1 = new Etudiant();

console.log(etudiant1.age);
//chainage des prototypes
//10
console.log(typeof etudiant1)
//object
```

Voici un nouvel exemple qui illustre la notion de prototype et de chaînage des prototypes :

```
// On commence par créer un objet o pour lequel la fonction f sera
// son constructeur et lui créera deux propriétés en propre
// a et b :
let f = function () {
    this.a = 1;
    this.b = 2;
}
let o = new f(); // {a: 1, b: 2}

// on ajoute des propriétés au prototype de la fonction
// f
f.prototype.b = 3;
f.prototype.c = 4;

// Note : on ne définit pas le prototype de f avec f.prototype = {b:3,c:4};
// car cela briserait la chaîne de prototype

// o.[[Prototype]] possède les propriétés b and c.
// o.[[Prototype]].[[Prototype]] est Object.prototype.
// Enfin, o.[[Prototype]].[[Prototype]].[[Prototype]] vaut null.
// On a alors atteint la fin de la chaîne de prototype car,
// par définition, null n'a pas de [[Prototype]].
// Ainsi, la chaîne complète est ici :
// {a: 1, b: 2} ---> {b: 3, c: 4} ---> Object.prototype ---> null

console.log(o.a); // 1
// Existe-t-il une propriété 'a' en propre sur o ? Oui, elle vaut 1.

console.log(o.b); // 2
// Existe-t-il une propriété 'b' en propre sur o ? Oui, elle vaut 2.
// Le prototype possède également une propriété 'b' mais elle n'est pas
// utilisée.
// C'est ce qu'on appelle l'ombrage (shadowing en anglais)

console.log(o.c); // 4
```

```
// Existe-t-il une propriété 'c' en propre sur o ? Non, on vérifie le
// prototype.
// Existe-t-il une propriété 'c' en propre sur o.[[Prototype]] ?
// Oui, elle vaut 4.

console.log(o.d); // undefined
// Existe-t-il une propriété 'd' en propre sur o ? Non, on vérifie le
// prototype.
// Existe-t-il une propriété 'd' en propre sur o.[[Prototype]] ? Non, on vérifie le
// prototype.
// o.[[Prototype]].[[Prototype]] est Object.prototype et ne contient pas
// de propriété 'd' par défaut. On vérifie son prototype.
// o.[[Prototype]].[[Prototype]].[[Prototype]] est null, on arrête la recherche
// aucune propriété n'est trouvée, le moteur renvoie undefined.
```

Cela commence à devenir compliqué, de plus il est possible de passer une instance d'un objet à un autre constructeur et définir une forme d'héritage, dans l'exemple qui suit Object.create (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) permet de créer un objet à partir d'un prototype, call() (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call) permet d'appeler une autre fonction en passant son this en paramètre et Object.prototype.constructor (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor) permet de définir le constructeur.

```
"use strict"
// Forme, la classe parente
let Forme = function() {
    this.x = 0;
    this.y = 0;
}

// Méthode de la classe parente
Forme.prototype.deplacer = function(x, y) {
    this.x += x;
    this.y += y;
    console.info('Forme déplacée.');
};

// Rectangle - classe fille
function Rectangle() {
    // on appelle un autre constructeur pour initialiser this
    Forme.call(this);
}

// La classe fille surcharge la classe parente
Rectangle.prototype = Object.create(Forme.prototype);
Rectangle.prototype.constructor = Rectangle;

var rect = new Rectangle();

console.log('instance de Rectangle ? ', (rect instanceof Rectangle));
// true
console.log('une instance de Forme ? ', (rect instanceof Forme));
// true
rect.deplacer(1, 1);
// Affiche 'Forme déplacée.'
```

2.2. Exercice

Créer un fonction permettant de définir un animal caractérisé par son nom et disposant de la méthode crie qui renvoie le nom suivi de "crie " puis une classe chien dont le crie renvoie nom suivi de "crie wawa ". Pour appeler la fonction crie de Animal vous pourrez utiliser :

```
 Animal.prototype.crie.call(...)
```

3. Les classes

3.1. Cours

Nous allons par habitude privilégier les sucres syntaxiques qui vont suivre.

Le mot clef class permet de définir une classe et constructor le constructeur. La notion de surcharge n'existe pas.

```
class Rectangle {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
}
```

Des classes peuvent-être anonyme ou nommée.

```
// anonyme
let Rectangle = class {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
};

// nommée
let Rectangle = class Rectangle {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
};
```

Les classes permettent de définir des getter et des setter sur les attributs (get nom_attribut, set nom_attribut).

```
class Rectangle {
    constructor(hauteur, largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
    }

    get area() {
        return this.calcArea();
    }

    set setLargeur(largeur) {
        if (largeur < 0)
            largeur = -largeur
        this.largeur = largeur
    }

    calcArea() {
        return this.largeur * this.hauteur;
    }
}

const carré = new Rectangle(10, 10);
carré.setLargeur = -10;
console.log(carré.area);
```

Les attributs peuvent être définis explicitement et être privés (#).

```
class Rectangle {
  #height = 0;
  #width;
  constructor(height, width) {
    this.#height = height;
    this.#width = width;
```

```
  }
}
```

Le mots clefs extend permet l'héritage simple, la notion d'interface n'existe pas. super permet de référencer les méthodes ou les attributs de la classe mère, les redéfinitions sont possibles et le polymorphisme.

```
class Cat {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

class Lion extends Cat {
    speak() {
        super.speak();
        console.log(`${this.name} roars.`);
    }
}

const l = new Lion("Fuzzy");
l.speak();
// Fuzzy makes a noise.
// Fuzzy roars.
```

Le mots clef static permet de définir des méthodes statiques ou des attributs statiques.

```
class ClassWithStaticMethod {
  static staticProperty = 'someValue';
  static staticMethod() {
    return 'static method has been called.';
  }
  static {
    console.log('Class static initialization block called');
  }
}

console.log(ClassWithStaticMethod.staticProperty);
// expected output: "someValue"
console.log(ClassWithStaticMethod.staticMethod());
// expected output: "static method has been called."
```

Des spécificités comme les fonctions génératrices existent, nous les introduirons au fur et à mesure des besoins (https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Statements/function*).

3.2. Exercice

Chaque classe devra être dans un fichier séparé de même que les tests. Définir une classe Humain définie par un nom qui ne peut être modifié mais qui peut-être consulté. Une classe Etudiant qui représente un Humain augmenté d'une note, une Promotion nommée qui contient des étudiants. On souhaite pouvoir connaître le nombre d'instance d'Humain. Vous pourrez utiliser throw new Error("...") et un try catch pour interdire l'ajout d'autre chose qu'un étudiant à la promotion.

4. Les fonctions fléchées et l'approche fonctionnelle

4.1. Cours

Les fonctions fléchées (`()=>{}`) ne sont un raccourci syntaxique sont des fonctions sans `this`, voici un exemple d'utilisation :

```
class Humain {
    constructor(sex,age){
        this.sex=sex;
```

```
        this.age=age;
    }
    toString(){
        const mensonge = function(){
            console.log(this);
            //probleme est que le this n'est pas celui de Humain
            //mais celui de mensonge
            return 20;
        }
        return "Humain "+this.sex+" "+mensonge();
    }
}

const robert = new Humain("F",18);
console.log(robert.toString());
//undefined
//Humain F 20

//Une version avec une fonction flechee qui na pas this
class Humain2 {
    constructor(sex,age){
        this.sex=sex;
        this.age=age;
    }
    toString(){
        const mensonge = () =>{
            console.log(this);
            //le this est celui de Humain2, mensonge n'en a pas
            this.age -=10;
            return this.age;
        }
        return "Humain "+this.sex+" "+mensonge();
    }
}
const robert2 = new Humain2("F",18);
console.log(robert2.toString());
console.log(robert2.age);
//Humain2 { sex: 'F', age: 18 }
//Humain F 8
//8
```

L'opérateur ... permet de transformer une collection en un ensemble de paramètres.

```
"use strict"
const tab = [1,2,3]
console.log(tab)
//[ 1, 2, 3 ]
console.log(...tab)
//1 2 3
```

4.2. Exercices

Vous devez utiliser la documentation de Array (https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Array).

4.2.1. Exercice 01

En partant du tableau suivant et en utilisant map de Array ainsi que l'opérateur ternaire (cond?retour_si_vrai:retour_si_faux) augmenter les notes sans dépasser 20. Puis calculer la moyenne en utilisant reduce et enfin donner le nombre de notes supérieures à la moyenne.

```
const tab = [10,20,0,5,8,12]
```

4.2.2. Exercice 02

Reprendre la promotion pour n'ajouter un élément que si il n'est pas présent, supprimer un étudiant, donner la moyenne, la liste des étudiant ayant la moyenne et ceux qui ne l'ont pas.

5. Les promesses et les fonctions asynchrones

Par défaut en JS toute fonction est synchrone, cela signifie que lors de son appel, elle s'exécute immédiatement et que l'instruction suivante attend la fin de l'instruction précédante avant se s'exécuter. Lorsque l'on manipule des entrées/sorties ou des appels réseau, ce comportement peut-être préjudiciable, par exemple si votre navigateur exécute un appel synchrone, il ne peut réagir aux événements.

5.1. Cours

Une première approche est l'utilisation de callback. Le principe de fonction de callback est la manière la plus classique de procéder:

1. on appelle une fonction asynchrone (A) en lui passant une autre fonction (B) – appelée fonction de callback – en paramètres;

2. la fonction asynchrone (A) rend immédiatement la main au programme;

3. la fonction de callback (B) sera appelée une fois que l'opération aura terminé son exécution.

Ainsi, quand on appelle plusieurs fonctions asynchrones d'affilée, leurs fonctions de callback respectives ne seront pas forcément exécutées dans le même ordre[1].

```
// setTimeout() est une fonction asynchrone qui exécute la fonction de callback
// après quelques millisecondes d'attente
setTimeout(() => console.log('a'), 50); // afficher a dans 50 millisecondes
setTimeout(() => console.log('b'), 90); // afficher b dans 90 millisecondes
setTimeout(() => console.log('c'), 20); // afficher c dans 20 millisecondes
// => ordre d'affichage: c, a, puis b
//    car les opérations asynchrones s'exécutent en parallèle
console.log("debut")
//s'affiche avant la fin des timers
```

En nodeJs, vous trouverez les conventions suivantes pour les callBack, la fonction de callback prend généralement deux paramètres:

- le premier paramètre est une instance de la classe Error – si l'opération a échoué – ou null

- le deuxième paramètre est le résultat de l'exécution de l'opération, dans le cas où elle s'est exécutée sans erreur. C'est la valeur qu'on aurait passé à return si notre fonction était synchrone.

```
// Ne disposant pas de mogodb, vous ne pourrez exécuter ce code
// Cette fonction appelera la fonction callback pour transmettre une réponse ou une erreur.

const meaningOfLife =  (callback) => {
    // supposons que la réponse est disponible dans une collection mongodb
    collection.findOne({ meaning: 'life' },  (err, res) => {
        if (err) {
            // la requête db a échoué => appeler callback en incluant le message d'erreur
            callback(new Error('meaningOfLife failed because ' + err.message));
        } else {
            // la requête db à réussi => appeler callback en incluant la réponse
            callback(null, res.answer);
        }
    });
}

meaningOfLife( (err, answer) =>{
    if (err) {
        console.error('meaningOfLife() a rapporté une erreur:', err);
    } else {
        console.log('valeur retournée par meaningOfLife():', answer);
    }
```

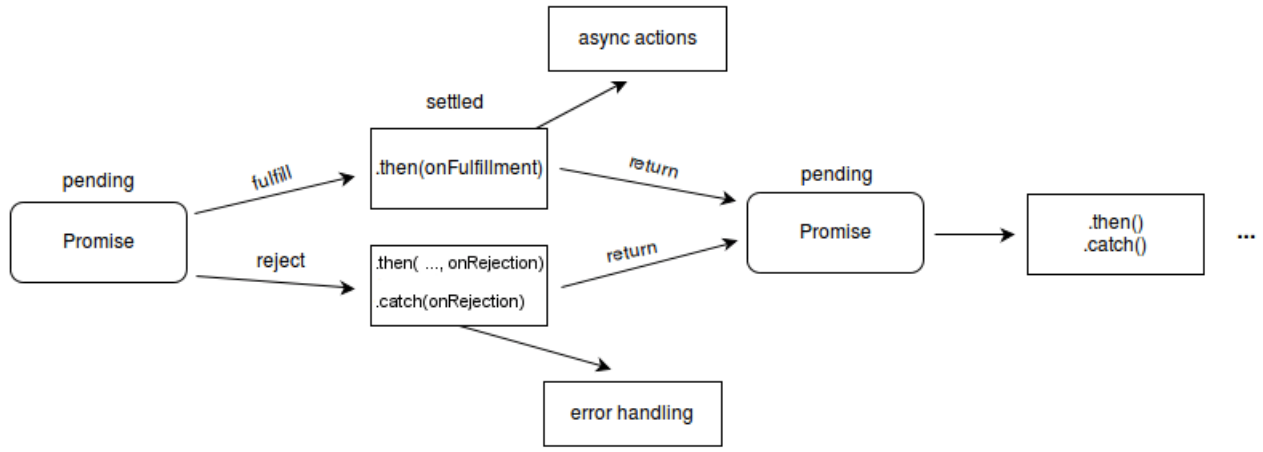
[1] source : https://adrienjoly.com/cours-nodejs/sync-vs-async.html

```
});
```

L'approche callback permet de cascader des appels en passant dans la callBack une autre callback et ainsi de suite (callback hell). Une autre impossibilité est la synchronisation des appels asynchrone, je souhaite attendre la fin de d'appels asynchrone avant de progresser, comment faire. Les promesses (Promise) permettent de répondre à ce problème, les fonction asynchrone (async/wait) sont un nouveau sucre syntaxique sur les promesses.

Une promesse est un objet (Promise) qui représente la complétion ou l'échec d'une opération asynchrone. Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais.

Figure 2.1. Promise



Une Promise peut être:

resolve

si l'opération asynchrone s'est exécutée avec succès;

reject

si une erreur est survenue pendant l'exécution de cette opération.

Les fonctions .then() et .catch() permettent de définir le comportement à adopter si la Promise est résolue ou rejetée, respectivement. .finally() est exécuté que la promesse soit résolue ou non.

Le promesse suivante tire un nombre aléatoire compris entre 0 et 1, si 0 est tiré, elle renvoie "gagné", "perdu" sinon.

```
const jeux = (nombre,plage) =>  new Promise((resolve,reject) =>{
    if (Math.floor(Math.random() * plage)==nombre)
        resolve("gagné")
    else
        reject("perdu")
})

jeux(0,1).then(res=>console.log(res)).catch(res=>console.log(res))
console.log("début")
//début
//gagné
```

Jeux est une fonction qui renvoie une promesse, jeux prend deux paramètres nombre, le nombre joué et plage la plage du tirage entre 0 et plage-1.

```
const jeux = (nombre,plage) =>  new Promise((resolve,reject) =>{
    if (Math.floor(Math.random() * plage)==nombre)
        resolve("gagné")
    else
        reject("perdu")
```

```
})

jeux(0,1).then(res=>console.log(res)).catch(res=>console.log(res))
console.log("début")
//début
//gagné
```

Il est possible d'ordonner les promesses en séquence en renvoyant une nouvelle promesse dans le then, le catch ou le finally.

```
"use strict"
const timer = (temps) =>  new Promise((resolve) =>{
    setTimeout(() => resolve("fini après " + temps), temps)
})

timer(10)
    .then(s => {console.log(s); return timer(20)})
    .then(s => {console.log(s)})
```

Promise dispose de méthodes statiques qui permettent d'exprimer d'autres formes de synchronisation :

Promise.all(iterable)

> Renvoie une promesse tenue lorsque toutes les promesses de l'argument itérable sont tenues ou une promesse rompue dès qu'une promesse de l'argument itérable est rompue. Si la promesse est tenue, elle est résolue avec un tableau contenant les valeurs de résolution des différentes promesses contenues dans l'itérable (dans le même ordre que celui-ci). Si la promesse est rompue, elle contient la raison de la rupture de la part de la promesse en cause, contenue dans l'itérable. Cette méthode est utile pour agréger les résultats de plusieurs promesses tous ensemble.

Promise.allSettled(iterable)

> Attend que l'ensemble des promesses aient été acquittées (tenues ou rompues) et renvoie une promesse qui est résolue après que chaque promesse ait été tenue ou rompue. La valeur de résolution de la promesse renvoyée est un tableau dont chaque élément est le résultat des promesses initiales.

Promise.any(iterable)

> Renvoie une seule promesse dont la valeur de résolution est celle de la première promesse résolue de l'itérable passé en argument.

Promise.race(iterable)

> Renvoie une promesse qui est tenue ou rompue dès que l'une des promesses de l'itérable est tenue ou rompue avec la valeur ou la raison correspondante.

Promise.reject(raison)

> Renvoie un objet Promise qui est rompue avec la raison donnée.

Promise.resolve(valeur)

> Renvoie un objet Promise qui est tenue (résolue) avec la valeur donnée. Si la valeur possède une méthode then, la promesse renvoyée « suivra » cette méthode pour arriver dans son état, sinon la promesse renvoyée sera tenue avec la valeur fournie. Généralement, quand on veut savoir si une valeur est une promesse, on utilisera Promise.resolve(valeur) et on travaillera avec la valeur de retour en tant que promesse.

```
"use strict"
const timer = (temps) =>  new Promise((resolve) =>{
    setTimeout(() => resolve("fini après " + temps), temps)
})

Promise.race([timer(10),timer(20)]).then(s => console.log(s))
//fini après 10
```

```
Promise.any([timer(10),timer(20)]).then(s => console.log(s))
//fini après 10
Promise.all([timer(10),timer(20)]).then(s => console.log(s))
//[ 'fini après 10', 'fini après 20' ]
```

Une fonction asynchrone est une fonction précédée par le mot-clé async, et qui peut contenir le mot-clé await. async et await permettent un comportement asynchrone, basé sur une promesse (Promise), écrite de façon simple, et évitant de configurer explicitement les chaînes de promesse. Il est à noter qu'un module est une fonction asynchrone.

```
//fichier .js
"use strict"
const timer = (temps) =>  new Promise((resolve) =>{
    setTimeout(() => resolve("fini après " + temps), temps)
})

const f = async () => {
   console.log(await timer(20))
    console.log(await timer(10))
}
console.log("début")
f()

//début
// fini après 20
// fini après 10
```

```
//fichie .mjs module
"use strict"
const timer = (temps) =>  new Promise((resolve) =>{
    setTimeout(() => resolve("fini après " + temps), temps)
})


   console.log(await timer(20))
    console.log(await timer(10))

console.log("début")

// fini après 20
// fini après 10
// /!\ début
```

5.2. Exercices

Les timers étant limités, nous allons utiliser des bibliothèques nodeJS, il nous faut commencer par créer un projet. Créer un répertoire s'y placer et taper :

```
npm init
```

Vous obtiendrez un fichier package.js qui définie votre projet, comme tout projet, il vous faudra travailler à la racine.

```
{
  "name": "project-name",
  "version": "0.0.1",
  "description": "Project Description",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "the repositories url"
  },
  "author": "your name",
  "license": "N/A"
}
```

5.2.1. Exercice 01

Rajouter dans script "start": "node index.mjs" et créer un fichier index.mjs. Exécuter votre script vide.

Nous allons ajouter le module nodeJS, la commande permet d'ajouter le module au projet courant[2] :

```
npm install node-fectch
```

Observer la modification du package.json et la création du répertoire node_modules, supprimer le répertoire node_modules et lancer :

```
npm install
```

Vous venez d'observer que vous pouvez recréer les dépendances de vos projets et que le répertoire node_modules n'est jamais à fournir.

Tester avec l'explorateur web d'API (onglet API), l'API des parking https://data.nantesmetropole.fr/explore/dataset/244400404_parkings-publics-nantes-disponibilites, pour obtenir la liste de tous les parkings (rows et nhits d'une réponse). Il vous faut donc deux requêtes pour obtenir l'ensemble des parking. Nous allons essayer de reproduire le même comportement.

Nous allons chercher à obtenir un tableau contenant l'ensemble des noms des parking trié par ordre alphabétique.

Etant à l'IUT, vous devrez utiliser un proxy, installer https-proxy-agent.

Comprendre et compléter le code suivant :

```
"use strict"
import fetch from 'node-fetch';
import HttpsProxyAgent from 'https-proxy-agent';

const proxy = process.env.https_proxy

let agent = null
if (proxy != undefined) {
    console.log(`Le proxy est ${proxy}`)
    agent =  new HttpsProxyAgent(proxy);
}
else {
    //pour pouvoir consulter un site avec un certificat invalide
    process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0";
    console.log("Pas de proxy trouvé")
}


const urlBase = 'https://data.nantesmetropole.fr/api/records/1.0/search'+
'/?dataset=244400404_parkings-publics-nantes-disponibilites&q=&facet=grp_nom&facet=grp_statut'


let response = agent!=null ? await fetch(urlBase, {agent: agent}):await fetch(urlBase)
let json = await response.json()
console.log(json)
const nb_hits = json.nhits
console.log(`Nombre de lignes trouvées ${nb_hits}`)
```

5.2.2. Exercice 02

NodeJS est fourni avec un ensemble de modules (https://nodejs.org/api/) , nous allons utiliser File System. Le code suivant permet de en utilisant readdir de lister un répertoire en utilisant une promesse readdir.

```
"use strict"

import { readdir } from 'node:fs/promises';
```

[2]Si le téléchargement bloque autoriser hhtp avec npm config set strict-ssl false

```
try {const files = await readdir(".")
   files.forEach(file => console.log(file))

} catch (err) {
    console.error(err);
}
```

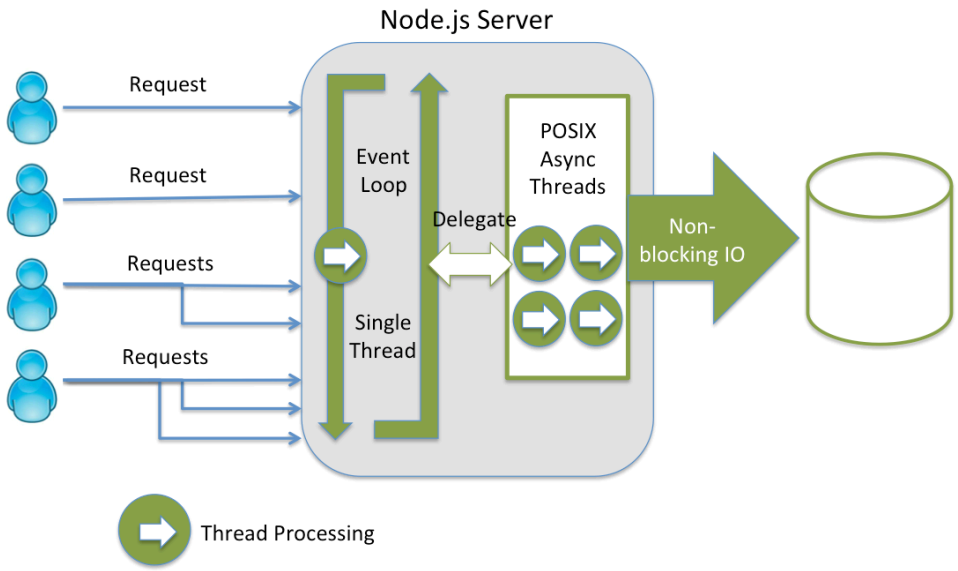
Mais readdir existe aussi sous forme de callback en ayant importé readdir depuis node:fs reproduire l'exemple précédant :

```
import { readdir } from 'node:fs';
```

6. Une spécificité de nodeJS : les événements

Node.js est monothreadé : toutes les opérations sont exécutées dans un seul thread. Mais Node.js s'appuie sur la boucle d'événements (event loop) qui lui permet de traiter des requêtes asynchrones non bloquantes. Ce fonctionnement permet à Node.js de réaliser des opérations d'entrées-sorties non bloquantes malgré son caractère monothreadé. Ces opérations peuvent être, par exemple, des requêtes HTTP, des lectures ou écritures de fichiers, des requêtes en base de données, etc. Les opérations synchrones sont exécutées séquentiellement en exploitant la pile d'appels (call stack). Tant que des opérations sont présentes sur la pile d'appels celles-ci sont exécutées. Il est donc essentiel de ne pas avoir des opérations qui monopolisent cette pile puisqu'il n'y a qu'un seul thread. Les opérations les plus pénalisantes doivent donc être exécutées de manière asynchrone. Elles sont alors gérées par l'event loop qui communique avec le noyau du système d'exploitation hôte pour lui déléguer l'exécution de ces opérations asynchrones dans des threads système[3].

Figure 2.2. Boucle d'événement



6.1. Cours

L'objet EventEmitter dispose de deux méthodes : on pour capturer et emit pour émettre.

```
import events from "events"

const em = new events.EventEmitter();

em.on('firstEvent', function (data) {
    console.log(data);
});

em.on('secondEvent', function (data) {
```

[3]source : https://www.fil.univ-lille.fr/~routier/enseignement/licence/jsfs/html/node-asynchrone.html

```
    console.log(data);
});

em.emit('firstEvent',"hello")
setTimeout(() => em.emit('secondEvent',10), 10)
```

6.2. Exercice

Télécharger le fichier csv correspondant aux jeux des restaurants : https://data.nantesmetropole.fr/explore/dataset/234400034_070-008_offre-touristique-restaurants-rpdl%40paysdelaloire.

En utilisant csv-parser (https://www.npmjs.com/package/csv-parser) afficher les restaurants du 44.

Chapter 3. Un service proxy

//TODO