



1. Comprendre les concepts de base du web front-end

1.1. Fondamentaux des sites web statiques

Un site Web statique consiste en une série de fichiers HTML, chacun représentant une page physique d'un site Web. Sur les sites statiques, chaque page est un fichier HTML distinct. Lorsque vous visitez la page d'accueil, vous ne visualisez que le fichier de la page d'accueil réelle

Les sites Web statiques sont assez simples et tous les sites Web ont été construits de cette façon pendant les premières années du World Wide Web.

1.2. Différence entre front End et back end :

C'est quoi front-end

Le front-end est construit à l'aide d'une combinaison de technologies telles que le langage de balisage hypertexte (HTML), JavaScript et les feuilles de style en cascade (CSS).

Les développeurs front-end conçoivent et construisent les éléments de l'expérience utilisateur sur la page Web ou l'application, y compris les boutons, les menus, les pages, les liens, les graphiques, etc.

C'est quoi back-end

Le back-end, également appelé côté serveur, se compose du serveur qui fournit les données à la demande, de l'application qui les canalise et de la base de données qui organise les informations.

Par exemple, lorsqu'un client parcourt des chaussures sur un site Web, il interagit avec le frontal.

Une fois qu'ils ont sélectionné l'article qu'ils souhaitent, l'ont mis dans le panier et autorisé l'achat, les informations sont conservées dans la base de données qui réside sur le serveur.

Quelques jours plus tard, lorsque le client vérifie l'état de sa livraison, le serveur extrait les informations pertinentes, les met à jour avec les données de suivi et les présente via le front-end.

1.3. Présentation des Frameworks front end :

Qu'est-ce qu'un framework ?

En gros, il s'agit d'une boîte à outils couplée à une bibliothèque pour programmeur informatique.

Il permet d'aider les programmeurs dans leur travail en leur proposant des morceaux de code pour mettre en place des fonctionnalités couramment demandées pour des applications. Par exemple, le multilinguisme, la gestion de la sécurité, la modularité (c'est-à-dire la possibilité pour l'utilisateur de personnaliser son interface).

L'avantage est que les programmeurs ne réinventent pas la roue chacun dans leur coin en programmant de A à Z tous les aspects de l'application demandée par le client.



Il faut savoir aussi qu'un projet nécessitera souvent l'utilisation de plusieurs frameworks. En effet, les frameworks sont généralement spécialisés dans un domaine bien précis :

- **Un framework back-end** permet à l'application de communiquer avec la base de données qui renferme des renseignements aussi précieux que les informations sur les utilisateurs, les sessions en cours, les articles en vente sur votre site... **Node JS** est un exemple de framework backend, tout comme **Ruby, Laravel, Django, Flask...**
- **Un framework front-end applicatif** permet de créer une interface pour que l'utilisateur puisse utiliser l'application. C'est le cas d'**Angular JS** mais aussi de **React, Vue JS**.
- Enfin, un **framework front-end de présentation** permet de définir l'apparence de l'application pour l'utilisateur. C'est le cas de **Bootstrap**.

Pourquoi utiliser les frameworks ?

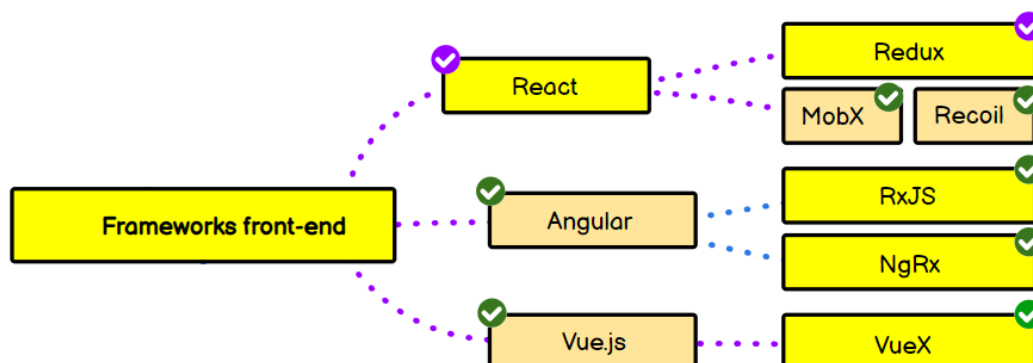
L'utilisation d'un framework n'est pas nécessaire dans un projet informatique, quelle que soit sa taille. Vous pouvez très bien vous en passer. Cependant, un framework est comme un squelette applicatif avec sa boîte à outils qui permet de développer plus rapidement, puisqu'il intègre une bonne partie de l'architecture, et invite les développeurs à suivre les normes de développement et de nommage tendant à une meilleure qualité du code.

Les avantages d'un framework ?

Utiliser un framework offre des avantages non négligeables :

- Les développeurs se concentrent uniquement sur la partie métier puisque toutes les couches techniques sont déjà intégrées dans le framework.
- L'architecture permet la séparation des couches techniques logiques afin de faciliter le développement en équipe, la maintenance et l'évolution.
- La maintenance et l'évolution du framework sont gérées par l'organisme fondateur.

Les frameworks front-end



React :



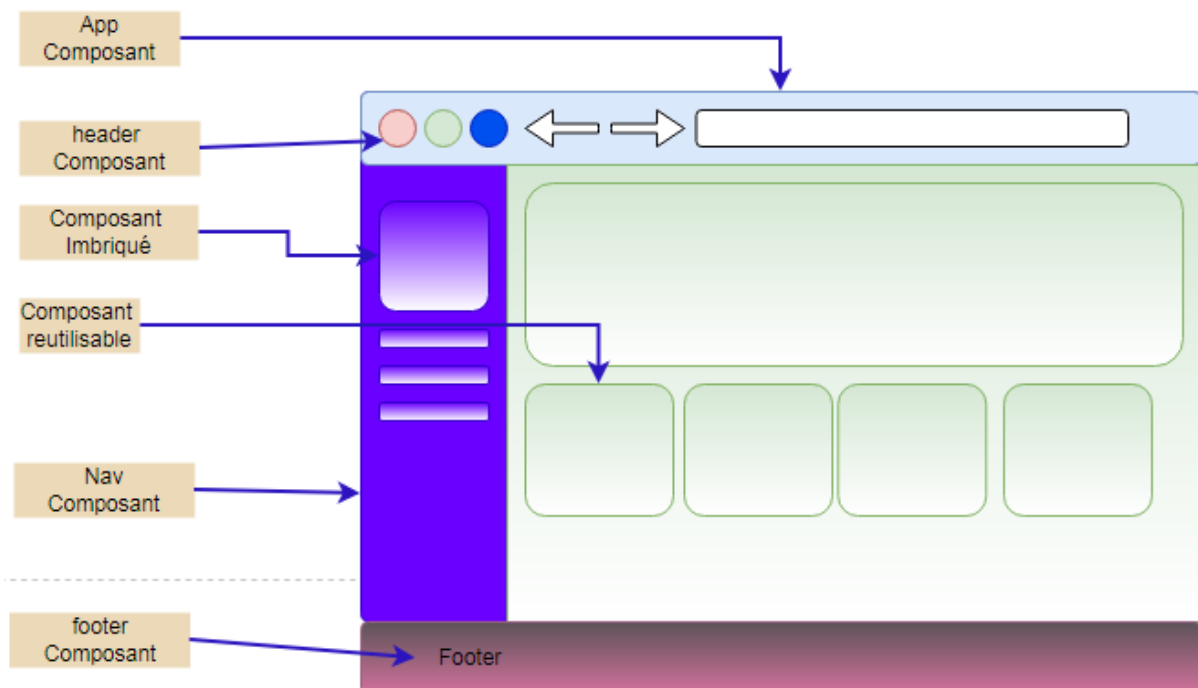
React c'est une librairie java script pour créer des interfaces utilisateur frontend, certaine personne l'appelle Framework, en général il ne faut pas prendre la tête avec est ce que React est une bibliothèque ou Framework, bref React c'est un utilitaire qui va nous aider à développer des interfaces utilisateur plus rapidement et facilement.

React est une librairie développée par Facebook est qui toujours maintenu avec Facebook

Pourquoi un Framework ou bibliothèque frontend ?

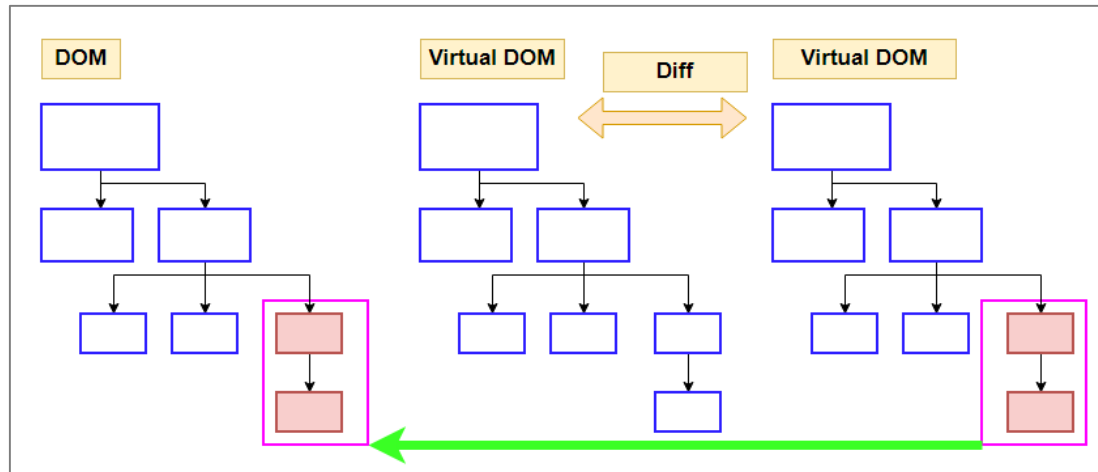
Avant l'arrivée des bibliothèques et Framework Frontend, les sites web ont été développés avec HTML, CSS, Java Script native, le site est constitué par plusieurs pages html, et quand l'utilisateur demande ou poste une information toutes la page est envoyée ce qui présente une lenteur de rafraîchissement, une charge sur le serveur et le navigateur.

L'approche des bibliothèques et Framework Frontend consiste à découper l'interface utilisateur en un ensemble de morceaux dits composants réutilisables.

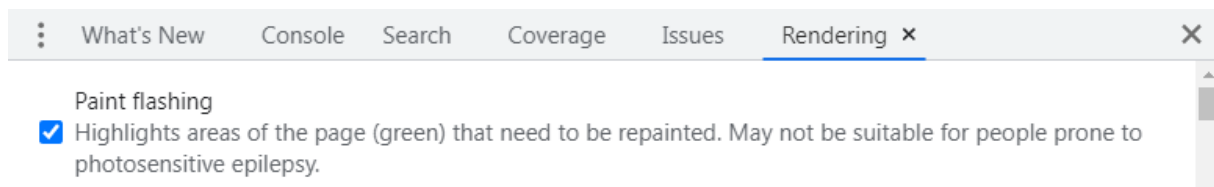


Virtuel DOM :

Le DOM virtuel (VDOM) est un concept de programmation dans lequel une représentation idéale ou "virtuelle" d'une interface utilisateur est conservée en mémoire et synchronisée avec le "vrai" DOM par une bibliothèque telle que ReactDOM. Ce processus s'appelle la réconciliation.



Ajouter l'outil rendering à Google Chrome pour afficher la partie du dom qui est rendu sera encadré



React est le framework open-source développé et créé par Facebook. Ce framework est le meilleur framework d'interface utilisateur de 2022, utilisé par la majorité des développeurs frontaux selon Framework frontal, React se distingue par son modèle d'objet de document (DOM) virtuel, qui présente ses excellentes fonctionnalités.

Avantages

- Gain de temps lors de la réutilisation des composants
- Virtual DOM améliore à la fois l'expérience des utilisateurs et le travail du développeur
- Une bibliothèque open-source avec une diversité d'outils

Quand on utilise REACT :

React est utilisé pour développer une interface utilisateur riche, en particulier lorsque vous devez créer des applications d'une seule page. C'est le framework front-end le plus robuste lorsque vous avez besoin de créer une interface interactive en moins de temps car il prend en charge les composants réutilisables.

React requière une bonne maîtrise de JavaScript !!!



Quelques framework js pour le développement FrontEnd

ANGULAR :



Angular est une plate-forme et un framework permettant de créer des applications clientes single page à l'aide de HTML et de TypeScript.

Angular est écrit en TypeScript. Il implémente les fonctionnalités de base et facultatives sous la forme d'un ensemble de bibliothèques TypeScript que vous importez dans vos applications.

Angular est un framework JavaScript open-source écrit en TypeScript. Il est maintenu par Google.

Angular permet de développer des applications web SPA single page application. En tant que framework, Angular présente des avantages évidents tout en fournissant une structure standard avec laquelle les développeurs peuvent travailler. Il permet aux utilisateurs de créer de grandes applications de manière maintenable.

VUE JS



Vue est un framework JavaScript pour la construction d'interfaces utilisateur. Il s'appuie sur les standards HTML, CSS et JavaScript et fournit un modèle de programmation déclaratif et basé sur des composants qui vous aide à développer efficacement des interfaces utilisateur, qu'elles soient simples ou complexes.

Quels sont les avantages de Vue.JS ?

Flexibilité

L'écriture d'une application à l'aide de Vue.JS est rapide dû au fait qu'il est possible de l'exécuter via son navigateur. Cela facilite également le processus de test. Des applications beaucoup plus complexes comme ES6, JSX, Routing, Components et Bundlers peuvent également être construites en utilisant Vue.JS. Les développeurs peuvent l'utiliser de nombreuses façons différentes, car ce framework offre une grande flexibilité dans l'expression de leur code.

Intégration simple

Vue.JS offre également de grandes possibilités d'intégration avec les applications existantes, ce qui explique sa popularité auprès des développeurs. En effet, ce framework est basé sur JavaScript et peut donc être facilement intégré à d'autres plateformes utilisant JavaScript. Grâce à cette capacité, les développeurs peuvent travailler avec l'application en cours sans avoir à développer l'application à partir de zéro.



1.4. Les aspects avancés de JavaScript :

Le développement Frontend avec React nécessite une bonne maîtrise de JavaScript ES6.

Apprendre les fonctionnalités JavaScript est vraiment conseillé pour que vous soyez efficace dans la création d'applications avec React. Voici donc quelques fonctionnalités JavaScript que je vous recommande à apprendre afin que vous puissiez être aussi efficace que possible en travaillant avec React.

Les classes d'objets en ES6

Les classes sont des modèles pour créer des objets. Ils encapsulent les données avec du code pour travailler sur ces données.

JavaScript permet de créer des classes d'objets, mais dans les versions précédentes (avant ES6), la syntaxe est compliquée et ne ressemble pas à celle que l'on utilise dans les langages orientés objets similaires. ES6 propose une syntaxe plus classique, qui sera abondamment utilisée avec React.

Création d'une classe :

On utilise le nouveau mot-clé `class` (comme dans beaucoup d'autres langages). La création d'un objet de cette classe s'effectue au moyen de `new`. On définit ci-après une classe `Etudiant` (remarquez la majuscule sur la première lettre du nom de la classe, car tous les noms de classe doivent commencer par une majuscule), puis on crée un objet `etudiant` (ici, le nom de la variable commence par une minuscule, contrairement aux noms de classe).

```
class Etudiant{
    constructor(nom, age){
        this.nom=nom;
        this.age=age
    }
}
let et1= new Etudiant("Rami",23)
let et2= new Etudiant("Karimi",21)
```

La méthode **constructor()** possède désormais les paramètres qui seront utilisés lors de la création des objets de la classe `Etudiant`. Afin de les conserver en tant qu'attributs dans la classe (et pouvoir les utiliser dans des méthodes qui seront définies dans cette classe), on les mémorise dans l'objet **this** (au moyen de **this.nom** et **this.age**). Remarquez que lors de la construction des objets par `new`, il faut maintenant indiquer en arguments les valeurs des paramètres utilisés, qui seront transmis dans la méthode `constructor()`. Si la méthode `constructor()` utilise des paramètres par défaut, les arguments correspondants peuvent être absents lors de la création des objets associés. On voit ici le contenu des deux objets créés, avec, entre accolades, les attributs créés au moyen de `this` dans le constructeur. Une classe est souvent enrichie au moyen de méthodes, qui seront ensuite utilisées par les objets de cette classe. Par



exemple, on peut créer une fonction info() dans la classe Personne qui permettra de visualiser le nom et age des objets de cette classe.

Création de la méthode info() dans la classe Etudiant :

```
class Etudiant{
    constructor(nom,age){
        this.nom=nom;
        this.age=age
    }
    info(){
        return `Etudiant nom:${this.nom} a pour
age:${this.age}`
    }
}
```

Appel de la méthode info() par les objets et1 et2

```
let et1= new Etudiant("Rami",23);
console.log(et1.info());
let et2= new Etudiant("Karimi",21) ;
console.log(et2.info());
```

Rendu

Etudiant nom:Rami a pour age:23
Etudiant nom:Karimi a pour age:21

Héritage de classe

L'héritage de classe permet de créer une nouvelle classe à partir d'une classe déjà existante. On dit que la nouvelle classe est dérivée (héritée) de la première. Il est ainsi possible d'enrichir (améliorer) une classe existante sans avoir à modifier son code interne. On peut alors réutiliser des parties de code existants (c'est-à-dire utiliser d'autres classes qui ont déjà été écrites et qui fonctionnent correctement). Pour illustrer cela, on souhaite créer une nouvelle classe, appelée Stagiaire, dérivée de la classe Etudiant.

```
class Stagiaire extends Etudiant{
    constructor(nom,age,stage){
        super(nom,age);
        this.stage=stage; }
    info(){return `Stagiaire nom:${this.nom} a pour
age:${this.age} stage:${this.stage}`
    }
}
```



La méthode info() peut être codé de la manière suivante

```
info(){  
    return `${super.info()} stage:${this.stage}`  
}
```

super(nom,age) :fait appelle au constructeur de la classe mère Etudiant

Création de deux instances de Stagiaire stg1 et stg2

```
let stg1= new Stagiaire("Rami",23,"dev mobile");  
console.log(stg1.info());  
let stg2= new Stagiaire("Karimi",21,"dev web")  
console.log(stg2.info());
```

Rendu

```
Stagiaire nom:Rami a pour age:23 stage:dev mobile  
Stagiaire nom:Karimi a pour age:21 stage:dev web
```

Modules

Le but des modules ES6 est d'aider à utiliser ces fichiers (ici, Etudiant.js et Index.js) sans avoir à connaître ces informations. En effet, ces dernières seront insérées sous une certaine forme dans le fichier lui-même, qui devient ainsi un module. C'est le rôle de l'export et de l'import des données, notions que nous détaillons dans les paragraphes qui suivent.

Un module c'est un fichier Java Script, qui contient les mots clés **export default** ou **export** elles les valeurs, classes, et fonctions qu'on veut exporter

C'est quoi la différence entre export default et export ?

Dans un module on peut faire seulement un seul export default, alors qu'on peut faire plusieurs exports.

Dans le module ci-dessous il un seul export default : **export default Etudiant** et deux export :

export {Etablissement,info}

L'élément exporté par default est importé sans accolades, tandis les éléments exportés par export sont importés avec des accolades

Voir module Index.js

```
import Etudiant,{Etablissement,info} from './Etudiant.js'
```

Module Etudiant.js

```
const Etablissement='ISGI'  
class Etudiant{  
    constructor(nom,age){  
        this.nom=nom;
```



```

        this.age=age
    }
}
function info(etudiant){
    return `nom:${etudiant.nom} age:${etudiant.age}`
}
export default Etudiant
export {Etablissement,info}

```

Il est identique à :

```

export const Etablissement='ISGI'
export default class Etudiant{
    constructor(nom,age){
        this.nom=nom;
        this.age=age
    }
}
export function info(etudiant){
    return `nom:${etudiant.nom} age:${etudiant.age}`
}

```

Module Index.js

```

import Etudiant,{Etablissement,info} from './Etudiant.js'
let etd1= new Etudiant("Rami",33)
console.log(info(etd1))

```

Utilisation des aléas :

```

import Etd,{Etablissement as Etab,info} from './Etudiant.js'
let etd1= new Etd("Rami",33)
console.log(info(etd1)+" Etablissement: "+Etab)

```

Utilisation des modules dans la balise <script>

La page index.html

```

<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8"></head>
<body>
    <script src="Index.js" type="module"></script>
</body>
</html>

```



Remarque : il faut ajouter l'attribut `type= "module"` a l'élément `script`

Template literals :

```
const salutation="salut "  
const nom="Rami"  
//avec template literal  
console.log(`${salutation} ${nom}`)  
//est identique a  
console.log(salutation+" "+nom)
```

Rendu

```
salut  Rami  
salut  Rami
```

En React : Voir Annexe 1

L'opérateur conditionnel ternaire

L'opérateur conditionnel (ternaire) est un opérateur JavaScript qui prend trois opérandes : une condition suivie d'un point d'interrogation (?), puis une expression à exécuter si la condition est vraie suivie de deux-points (:), et enfin l'expression à exécuter si la condition est fausse. Cet opérateur est fréquemment utilisé comme alternative à une instruction `if...else`.

L'opérateur conditionnel ternaire est très utilisé en react notamment en JSX
exemple en Java Script ECS 6 :

```
let isMember=true;  
let remise=isMember==true?0.2:0.1  
console.log("remise: ",remise)
```

Rendu :

```
remise: 0.2
```

En React : Voir Annexe 2

Gestion des valeurs nulles avec operateur conditionnel ternaire

Une utilisation courante consiste à gérer une valeur pouvant être nulle :

Exemple 1:

```
let nom  
let salutation=nom?"salut "+nom:"inconnu"  
console.log(salutation)
```

Rendu :

```
inconnu
```

**Exemple 2 :**

```
let nom="Rami"  
let salutation=nom?"salut "+nom:"inconnu"  
console.log(salutation)
```

Le rendusalut Rami**Object destructuring**

La syntaxe d'affectation de déstructuration est une expression JavaScript qui permet de décompresser des valeurs de tableaux, ou des propriétés d'objets, dans des variables distinctes.

```
const personne={  
  nom:"fatihi",  
  age:23,  
  adresse:{rue:14,ville:"casa"}  
}  
const nom=personne.nom  
const age=personne.age  
const rue=personne.adresse.rue  
const ville=personne.adresse.ville  
console.log(nom,age,rue,ville)  
  
//est identique à:  
const {nom,age,adresse:{rue},adresse:{ville}}=personne  
console.log(nom,age,rue,ville)
```

Le rendufatihi 23 14 casa**Destructuring avec fonction**

```
const personne={  
  nom:"fatihi",  
  age:23,  
  adresse:{rue:14,ville:"casa"}  
}  
function presentation({nom,age,adresse:{rue},adresse:{ville}}){  
  console.log("salut ",nom,age,rue,ville)  
}  
presentation(personne)
```

Le rendusalut fatihi 23 14 casa**En React : Voir Annexe 3**



Opérations sur Array

La méthode map :

map est une méthode très populaire elle permet de :

- Retourne un nouveau Array
- Le nouveau Array a la même taille que l'Array d'origine
- Utilise les valeurs de l'Array d'origine pour faire le nouveau Array

Exemple : création d'un Array contenant les ages des personnes :

```
const personnes = [  
  { nom: "Rami", age: 33, estMember: true },  
  { nom: "Fatih", age: 24, estMember: false },  
  { nom: "Chakib", age: 45, estMember: true },  
  { nom: "Mounir", age: 37, estMember: false },  
];  
  
const noms=personnes.map(function(pers){return pers.nom})  
console.log(noms)
```

Le rendu : `(4) ['Rami', 'Fatih', 'Chakib', 'Mounir']`

Exercice 1 :

Soit l'Array nums=[2,5,8,7,3]

Utiliser la méthode map pour créer un Array contenant les éléments de nums multipliés par 3

nouvNums=[6,15,24,21,9]

La méthode filter

La méthode filter() crée une copie d'une partie d'un tableau donné, filtrée uniquement pour les éléments du tableau donné qui réussissent la condition implémentée par la fonction fournie.

Exemple :

```
const personnes = [  
  { nom: "Rami", age: 33, estMember: true },  
  { nom: "Fatih", age: 24, estMember: false },  
  { nom: "Chakib", age: 45, estMember: true },  
  { nom: "Mounir", age: 37, estMember: false },  
];  
  
const persAgés=personnes.filter(function(pers){return  
pers.age>=35})  
console.log(persAgés)
```



Le rendu :

```
▼ (2) [{...}, {...}] ⓘ  
  ► 0: {nom: 'Chakib', age: 45, estMember: true}  
  ► 1: {nom: 'Mounir', age: 37, estMember: false}  
    length: 2  
  ► [[Prototype]]: Array(0)
```

L'Array persAgés contient les éléments de personnes qui vérifient la condition age>=35

Exercice 2 :

En utilisant les méthodes map et filter
créer un Array nomAgés à partir de l'Array personnes contenant les noms des
personnes qui sont membre c-à-d estMember est true

En React : Voir Annexe 4

La méthode find

La méthode find() renvoie le premier élément du tableau fourni qui satisfait la
fonction de test fournie. Si aucune valeur ne satisfait la fonction de test, undefined
est renvoyé.

Exemple :

```
const personnes = [  
  { nom: "Rami", age: 33, estMember: true },  
  { nom: "Fatihi", age: 24, estMember: false },  
  { nom: "Chakib", age: 45, estMember: true },  
  { nom: "Mounir", age: 37, estMember: false },  
];  
const pers1=personnes.find(function(pers){return  
pers.age>=45})  
console.log(pers1)
```

Le rendu :

```
► {nom: 'Chakib', age: 45, estMember: true}
```

Exemple 2 :

```
const pers1=personnes.find(function(pers){return  
pers.age==50})  
console.log(pers1)
```

Le rendu :

```
undefined
```



La méthode reduce

La méthode `reduce()` exécute une fonction de rappel "reducer" fournie par l'utilisateur sur chaque élément du tableau, dans l'ordre, en transmettant la valeur de retour du calcul sur l'élément précédent. Le résultat final de l'exécution du réducteur sur tous les éléments du tableau est une valeur unique.

La première fois que le rappel est exécuté, il n'y a pas de "valeur de retour du calcul précédent". Si elle est fournie, une valeur initiale peut être utilisée à sa place. Sinon, l'élément de tableau à l'index 0 est utilisé comme valeur initiale et l'itération commence à partir de l'élément suivant (index 1 au lieu de l'index 0)

```
const clients = [  
  { nom: "Rami", montant: 4500 },  
  { nom: "Karimi", montant: 2300 },  
  { nom: "Chaouki", montant: 5500 },  
  { nom: "Ramoun", montant: 7700 },  
];  
const totalMontants=clients.reduce(function(total,client){  
  return total+=client.montant  
},0);  
console.log(totalMontants)
```

Destructuring d'un array

Moyen plus rapide et plus simple d'accéder à la variable de décompression à partir d'un tableau ou d'objets.

```
let numbers=[10, 20, 30, 40, 50]  
const [a, b,...rest] = numbers;  
console.log(a,b,rest);
```

Le rendu : `10 20 ► (3) [30, 40, 50]`

Remarque :

`rest` est un array contenant une copie de Array `numbers` sans le premier et le deuxième élément

Créer une copie d'une Array

```
const numbers2=[...numbers]  
console.log(numbers2)
```

Le rendu

`► (5) [10, 20, 30, 40, 50]`



Permuter les valeurs de deux variables :

```
let x=10;  
let y=20;  
[x,y]=[y,x]  
console.log("x=",x, " y=",y)
```

Le rendu :

x= 20 y= 10

La programmation fonctionnelle

La programmation fonctionnelle est un paradigme de construction de programmes informatiques à l'aide d'expressions et de fonctions sans mutation d'état et de données. En respectant ces restrictions, la programmation fonctionnelle vise à écrire du code plus clair à comprendre et plus résistant aux bogues.

Programmation fonctionnelle

modulaire

La fonctionnalité de votre programme doit être divisée en modules indépendants, chacun contenant ce dont il a besoin pour exécuter un aspect de la fonctionnalité du programme. Les modifications apportées à un module ou à une fonction ne doivent pas affecter le reste du code

compréhensible

Un lecteur de votre programme devrait être capable de discerner ses composants, leurs fonctions et leurs relations sans effort excessif. Ceci est étroitement lié à la maintenabilité du code ; votre code devra être maintenu à un moment donné dans le futur, que ce soit pour être modifié ou pour ajouter de nouvelles fonctionnalités

testable

Les tests unitaires testent de petites parties de votre programme, vérifiant leur comportement en fonction du reste du code. Votre style de programmation doit favoriser l'écriture de code qui simplifie le travail d'écriture de tests unitaires. Les tests unitaires sont également comme de la documentation en ce sens qu'ils peuvent aider les lecteurs à comprendre ce que le code est censé faire

extensible

C'est un fait que votre programme nécessitera un jour une maintenance, peut-être pour ajouter de nouvelles fonctionnalités. Ces modifications ne devraient avoir qu'un impact minime sur la structure et le flux de données du code d'origine (voire pas du tout). De petits changements ne doivent pas impliquer une refactorisation importante et sérieuse de votre code

réutilisable

La réutilisation du code a pour objectif d'économiser des ressources, du temps et de l'argent, et de réduire la redondance en tirant parti du code précédemment écrit. Certaines caractéristiques contribuent à cet objectif, telles que la modularité (que nous avons déjà mentionnée), une forte cohésion (toutes les pièces d'un module vont ensemble), un faible couplage (les modules sont indépendants les uns des autres)



JavaScript n'est pas un langage purement fonctionnel, mais il possède toutes les fonctionnalités dont nous avons besoin pour qu'il fonctionne comme s'il l'était. Les principales caractéristiques du langage que nous allons utiliser sont les suivantes

function as first-class objects

recursive

Arrow functions

closures

spread

La récursivité

La récursivité est l'outil le plus puissant pour développer des algorithmes et une grande aide pour résoudre de grandes classes de problèmes. L'idée est qu'une fonction peut à un certain point s'appeler elle-même, et quand cet appel est fait, continuer à travailler avec le résultat qu'elle a reçu. Ceci est généralement très utile pour certaines classes de problèmes ou de définitions. L'exemple le plus souvent cité est la fonction factorielle (la factorielle de n s'écrit $n!$) telle que définie pour des valeurs entières positives :

```
function fact(n){  
  if (n===0){  
    return 1  
  }  
  else{  
    return n* fact(n-1)  
  }  
}  
  
console.log(fact(4))
```

Fermetures

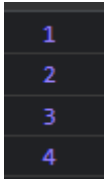
Les fermetures sont un moyen d'implémenter le masquage des données (avec des variables privées), ce qui conduit à des modules et à d'autres fonctionnalités



intéressantes. Le concept clé des fermetures est que lorsque vous définissez une fonction, elle peut faire référence non seulement à ses propres variables locales, mais également à tout ce qui se trouve en dehors du contexte de la fonction. Nous pouvons écrire une fonction de comptage qui gardera son propre compte au moyen d'une fermeture

```
function compteur(){  
  let count=0  
  return function(){  
    count++  
    return count  
  }  
}  
let next=compteur()  
console.log(next())  
console.log(next())  
console.log(next())  
console.log(next())
```

Le rendu



```
1  
2  
3  
4
```

Fonction as first-class object

Dire que les fonctions sont des objets de première classe (également appelés citoyens de première classe) signifie que vous pouvez tout faire avec des fonctions que vous pouvez faire avec d'autres objets. Par exemple, vous pouvez stocker une fonction dans une variable, vous pouvez la passer à une fonction, vous pouvez l'imprimer, etc. C'est vraiment la clé de la FP ; nous passerons souvent des fonctions en tant que paramètres (à d'autres fonctions) ou retournant une fonction à la suite d'un appel d'une fonction.

Exemple compteur voir TD:

```
spanHeure=document.getElementById("spHeure")  
spanMinute=document.getElementById("spMinute")  
spanSecond=document.getElementById("spSecond")  
let initTime={heure:10,minute:23,second:5}  
let currentTime=initTime  
  
let timer=null  
function incrementer(temps){  
  time={...temps}  
  return function(){  
    time.second++  
    if(time.second>=60){
```



```
        time.second=0
        time.minute++
        if(time.minute>=60){
            time.minute=0
            time.heure++
        }
    }
    affiche(time)
    currentTime=time
    return time
}
}

function affiche(time){
    spanHeure.innerHTML=time.heure;
    spanMinute.innerHTML=time.minute;
    spanSecond.innerHTML=time.second;
}

function stopTimer() {
    if(timer){
        clearInterval( timer);
    }
    timer = null;
}

function incrementerTimer() {

    stopTimer()
    let next=incrementer(currentTime)
    timer=setInterval(next,1000);
}
```

Ici on a :

- ✓ La fonction incrementer est une fermeture elle retourne une fonction !!!
- ✓ La fonction affiche est appelé au sein de la fonction incrementer,
- ✓ La fonction **next** est passée en paramètre dans la fonction setInterval



Arrow fonction

Les arrow fonctions sont juste un moyen plus court et plus succinct de créer une fonction (sans nom). Les arrow fonctions peuvent être utilisées presque partout où une fonction classique peut être utilisée, sauf qu'elles ne peuvent pas être utilisées comme constructeurs

La fonction somme :

```
function somme(a,b){  
    return a+b  
}
```

Peut-être écrite en utilisant arrow fonction

```
const mySomme=(a,b)=>a+b  
console.log(mySomme(3,4))
```

Remarque :

Les fonctions fléchées sont généralement appelées fonctions anonymes en raison de leur absence de nom. Si vous avez besoin de faire référence à une fonction fléchée, vous devrez l'affecter à une variable ou à un attribut d'objet, comme nous l'avons fait ici ; sinon, vous ne pourrez pas l'utiliser.

Exemple :

```
const fact3 = n => (n === 0 ? 1 : n * fact3(n - 1));
```

L'opérateur de propagation spread

Vous permet de développer une expression aux endroits où vous nécessiterait autrement plusieurs arguments, éléments ou variables. Par exemple, vous pouvez remplacer des arguments dans un appel de fonction, comme illustré dans le code suivant :

```
function produit(a,b,c){  
    return a*b*c  
}  
  
let numbers=[2,3,4]  
console.log(produit(...numbers))
```

Pures fonctions :

Les fonctions pures acceptent des entrées (arguments) et renvoient une nouvelle valeur sans avoir à modifier en permanence des variables en dehors de sa portée.

Dans la fonction pure ci-dessous 'pureFonction' , nous avons pu récupérer une nouvelle valeur sans modifier la variable 'immutableValue'.

Alors que la fonction 'impureFonction' a changée la valeur de la variable 'mutateValue' d'où 'impureFonction' n'est pas pure !!!.



```
let mutateValue = 10;
function impureFonction(newValue) {
  return mutateValue += newValue;
}
impureFonction(20)
console.log(mutateValue)
//rendu 30
immutableValue=10
function pureFonction(newValue){
  return immutableValue+newValue
}
pureFonction(20)
console.log(immutableValue)
//rendu 10
```

Alors pourquoi la fonction pure est-elle si importante ?

Les fonctions pures favorisent la maintenabilité et la réutilisabilité ; à l'opposé, la fonction impure limite ces qualités car lorsqu'elles mutent des valeurs en dehors de son champ d'application, l'effet secondaire qu'elle provoque peut être indésirable. Cela favoriserait à son tour le code bourré de bogues.

Chose intéressante, React + Redux (qui, à mon avis, devient le framework/bibliothèque Javascript standard de l'industrie) pratique des fonctions pures pour sa gestion d'état. Inutile de dire que développer une habitude d'écrire des fonctions pures est important pour une transition facile dans le framework.

Pures fonctions et Arrays :

Ajouter un élément dans un Array

```
//ajouter un element dans la liste
myList=[1,2,3,4,5]
function pureAppend(value){
  appendMyList=[...myList,value];
  return appendMyList;}
//affichage
console.log(pureAppend(10))
//rendu [1,2,3,4,5,10]
console.log(myList)
//rendu [1,2,3,4,5]
```

On remarque myList n'a pas changé





Exercice 3 :

Soit l'Array

```
const inscriptions=[  
  {id:10,nom:'Rami',filier:'DEV'},  
  {id:11,nom:'Kamali',filier:'DEV'},  
  {id:12,nom:'Fahmi',filier:'DEV'},  
  {id:13,nom:'Chaouki',filier:'DEV'}  
];
```

Créer la pure fonction qui permet d'ajouter une inscription

Insérer dans un Array

La méthode splice() modifie le contenu d'un tableau en supprimant ou en remplaçant des éléments existants et/ou en ajoutant de nouveaux éléments en place. Pour accéder à une partie d'un tableau sans le modifier

```
const months = ['Jan', 'March', 'April', 'June'];  
months.splice(1, 0, 'Feb');  
// inserts at index 1  
console.log(months);  
// expected output: Array ["Jan", "Feb", "March", "April", "June"]  
  
months.splice(4, 1, 'May');  
// replaces 1 element at index 3  
console.log(months);  
// expected output: Array ["Jan", "Feb", "March", "April", "May"]
```

Inject avec pure fonction

```
let dontMutateInjection = [6, 7, 8];  
  
function pureInject(index,newValue) {  
  return [  
    ...dontMutateInjection.slice(0, index),  
    newValue,  
    ...dontMutateInjection.slice(index)  
  ]  
}  
  
console.log('-----')  
console.log(pureInject(1,10))  
//rendu [6,10,7,8]  
console.log(dontMutateInjection)  
//rendu [6,7,8]
```



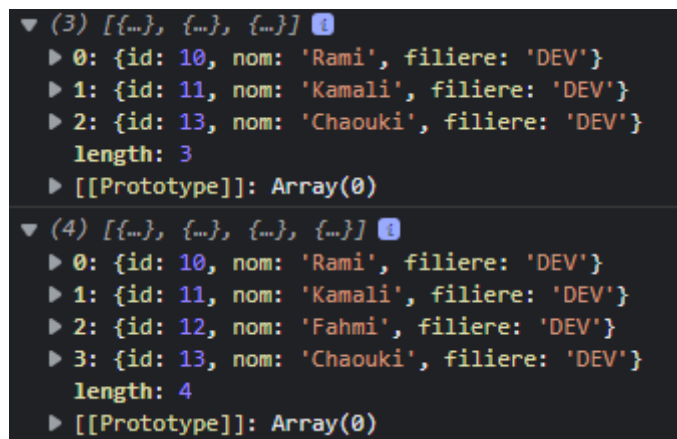
Supprimer un élément d'un Array

Soit :

```
const inscriptions=[
  {id:10,nom:'Rami',filiere:'DEV'},
  {id:11,nom:'Kamali',filiere:'DEV'},
  {id:12,nom:'Fahmi',filiere:'DEV'},
  {id:13,nom:'Chaouki',filiere:'DEV'}
];
```

```
function pureDeleteFonction(id){
  return [...inscriptions.filter((ins)=>ins.id!=id)]
}
let deleteInscriptions= pureDeleteFonction(12)
console.log('----pureDeleteFonction-----')
console.log(deleteInscriptions)
console.log(inscriptions)
```

Le rendu



```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {id: 10, nom: 'Rami', filiere: 'DEV'}
  ▶ 1: {id: 11, nom: 'Kamali', filiere: 'DEV'}
  ▶ 2: {id: 13, nom: 'Chaouki', filiere: 'DEV'}
    length: 3
  ▶ [[Prototype]]: Array(0)
▼ (4) [{...}, {...}, {...}, {...}] ⓘ
  ▶ 0: {id: 10, nom: 'Rami', filiere: 'DEV'}
  ▶ 1: {id: 11, nom: 'Kamali', filiere: 'DEV'}
  ▶ 2: {id: 12, nom: 'Fahmi', filiere: 'DEV'}
  ▶ 3: {id: 13, nom: 'Chaouki', filiere: 'DEV'}
    length: 4
  ▶ [[Prototype]]: Array(0)
```

Modifier un élément d'un Array

Soit

```
const notes=[
  {id:10,module:'Algorithme',note:15},
  {id:10,module:'POO',note:17},
  {id:11,module:'Algorithme',note:16},
  {id:11,module:'POO',note:14},
];
```

On souhaite changer la note POO de l'étudiant dont le id =10

```
//pure fonction modification
function pureUpdateNote(id,module,note){
  const updatednotes= [...notes.filter(note=> !(note.id===id &&
note.module===module)),{id:id,module:module,note:note}]
  return updatednotes;
}
console.log("----pureUpdateNote-----")
console.log(pureUpdateNote(10,'POO',18))
console.log(notes)
```

Le rendu :

```
▼ (4) [{...}, {...}, {...}, {...}] ③
  ▶ 0: {id: 10, module: 'Algorithme', note: 15}
  ▶ 1: {id: 11, module: 'Algorithme', note: 16}
  ▶ 2: {id: 11, module: 'POO', note: 14}
  ▶ 3: {id: 10, module: 'POO', note: 18}
    length: 4
  ▶ [[Prototype]]: Array(0)

▼ (4) [{...}, {...}, {...}, {...}] ③
  ▶ 0: {id: 10, module: 'Algorithme', note: 15}
  ▶ 1: {id: 10, module: 'POO', note: 17}
  ▶ 2: {id: 11, module: 'Algorithme', note: 16}
  ▶ 3: {id: 11, module: 'POO', note: 14}
    length: 4
  ▶ [[Prototype]]: Array(0)
```

Changer un objet

Soit l'objet

```
personne={id:10,nom:'Rami'}
```

on souhaite avoir un objet de type personne avec en plus l'attribut age sans modifier l'objet initial

```
personne={id:10,nom:'Rami'}
//changer un objet en utilisant pure fonction
function addAge(age){
  let pers= {...personne,age:age}
  return pers;
}
console.log('changer personne')
console.log(addAge(33))
console.log(personne)
```

La fonction peut être écrite autrement

```
function addAge(age){
  return Object.assign({}, personne, {age:age});
}
```