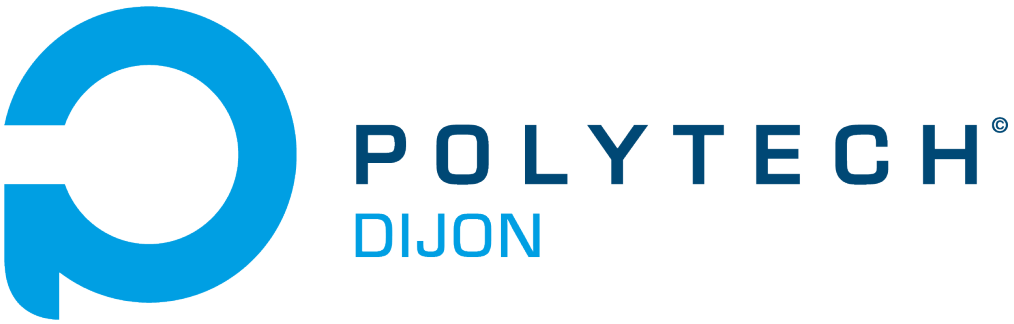
Polytech Dijon 3A options : Informatique & Réseaux

Module : ITC315

TD JS LocalStorage

Auteur :
HUBERT Matéo

Professeur référent :
MEUNIER Charles



2023-2024

Table des matières

1 UNE CARTE **3**
1.1 MODELE 3
1.2 CONTROLLER 3
1.3 VUE 4
1.4 APPLICATION 4

2 DES CARTES **5**

3 LE JEU **6**
3.1 MEMORY 6
3.2 CONTROLLER 7
3.3 VIEW 7
3.4 APPLICATION 8

4 UN PEU D'ALEATOIRE **9**

5 SERIALISATION / DESERIALISATION **10**
5.1 CARD 10
5.2 MEMORY 10
5.3 CONTROLLER 10

6 CHARGEMENT **11**
6.1 MEMORY 11
6.2 CONTROLLER 11
6.3 APPLICATION 12
6.4 SESSIONSTORAGE 12

Table des figures

1 card.js . . . 3
2 controller-memory.js . . . 3
3 view-memory.js.js . . . 4
4 application-memory.js . . . 4
5 Afficher une carte . . . 5
6 Fonction modifiée . . . 5
7 Afficher des cartes . . . 5
8 memory.js . . . 6
9 controller-memory.js . . . 7
10 view-memory.js . . . 7
11 application-memory.js . . . 8
12 Affichage de 10 paires de cartes dans l'ordre . . . 8
13 Modification de la fonction newGame . . . 9
14 Affichage de 10 paires de cartes mélangées . . . 9
15 Méthode toData de card.js . . . 10
16 Méthode toData de memory.js . . . 10
17 Méthode saveGame de controller-memory.js . . . 10
18 LocalStorage du navigateur . . . 11
19 Méthode fromData de memory.js . . . 11
20 Méthode loadGame et start de controller-memory.js . . . 11
21 newGame dans application-memory.js . . . 12
22 Plusieurs parties chargées . . . 12

1 UNE CARTE

L'entièreté du code est disponible sur ce repo

1.1 MODELE

```
export class Card{
    #value;
    constructor(value){
        this.#value = value;
    }
    get value(){
        return this.#value;
    }
}
```

FIGURE 1 – card.js

La classe Card possède un attribut privé value qui contient la valeur de la carte instanciée via le constructeur. La classe possède également un getter qui retourne la valeur de la carte.

1.2 CONTROLLER

```
import { Card } from "../models/card.js";
//import { Memory } from "../models/memory.js";
import { Notifier } from "../patterns/notifier.js";

export class ControllerMemory extends Notifier
{
    #card = null;
    constructor()
    {
        super();
    }
    get card(){
        return this.#card;
    }
    createCard(){
        let random_premier = Math.floor(Math.random() * 15);
        let random_second= Math.floor(Math.random() * 3);
        const tab_premier = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"];
        const tab_second = ["C", "D", "E", "F"];
        const debut = "0x1F9";
        let premier = tab_premier[random_premier];
        let second = tab_second[random_second];
        let valeur = debut + premier + second;
        valeur = parseInt(valeur);
        this.#card = new Card(valeur);
        this.notify();
    }
}
```

FIGURE 2 – controller-memory.js

La classe ControllerMemory possède un attribut privé card qui est instancier à null. Comme la classe hérite de Notifier, il faut appeler le constructeur dans le constructeur via super. La classe possède un getter qui retourne la carte de la classe ainsi qu'une méthode createCard qui créer une carte avec une valeur aléatoire. Pour cela, on créer 2 nombres aléatoires, un entre 0 et 15 et l'autre entre 0 et 3 pour sélectionner un élément du tableau au hasard afin de faire un nombre héxa aléatoire. Ensuite, on transforme la chaîne de caractère en int puis l'on créer une nouvelle carte avec cette valeur. On appelle ensuite la fonction notify pour informer des changements.

1.3 VUE

```js
import { Observer } from "../patterns/observer.js";

export class ViewMemory extends Observer
{
    #controllerMemory;

    constructor(controllerMemory)
    {
        super();

        this.#controllerMemory = controllerMemory;
        this.#controllerMemory.addObserver(this);
    }

    notify()
    {
        this.displayCard();
    }
    displayCard(){
        const cards = document.querySelector(".cards");
        let enfant = document.createElement("div");
        enfant.classList.add("card");
        enfant.innerHTML = "&#x" + (this.#controllerMemory.card.value).toString(16);
        cards.appendChild(enfant);
    }
}
```

FIGURE 3 – view-memory.js.js

Le but de la méthode displayCard est d'afficher une carte sur la page. Pour cela, on récupère avec un querySelector la balise cards. On créer ensuite un élément HTML div auquel on applique la classe CSS card. Ensuite, on modifie le contenu de la div par la valeur de la carte transformée en string à partir de la base 16 précédée de &#x. Une fois, cela fait, on ajoute notre élément comme enfant à la balise cards. On appelle la méthode displayCard dans notify pour que quand on créer une carte, elle soit affichée.

1.4 APPLICATION

```js
import { ControllerMemory } from "../controllers/controller-memory.js";
import { ViewMemory } from "../views/view-memory.js";

export class ApplicationMemory
{
    #controllerMemory;
    #viewMemory;

    constructor()
    {
        this.#initControllers();
        this.#initViews();
        this.#controllerMemory.createCard();
    }

    #initControllers()
    {
        this.#controllerMemory = new ControllerMemory();
    }

    #initViews()
    {
        this.#viewMemory = new ViewMemory(this.#controllerMemory);
    }
}
```

FIGURE 4 – application-memory.js

On appelle la fonction createCard du controller dans le constructeur et on obtient ceci :

FIGURE 5 – Afficher une carte

2 DES CARTES

Pour afficher une nouvelle carte à chaque fois que l'on clique sur une carte il suffit de rajouter ces 3 lignes à la fin de la fonction displayCard :

```
enfant.addEventListener("click", () => {
    this.#controllerMemory.createCard();
});
```

FIGURE 6 – Fonction modifiée

Voici le résultat que l'on peut obtenir si l'on clique sur les cartes :

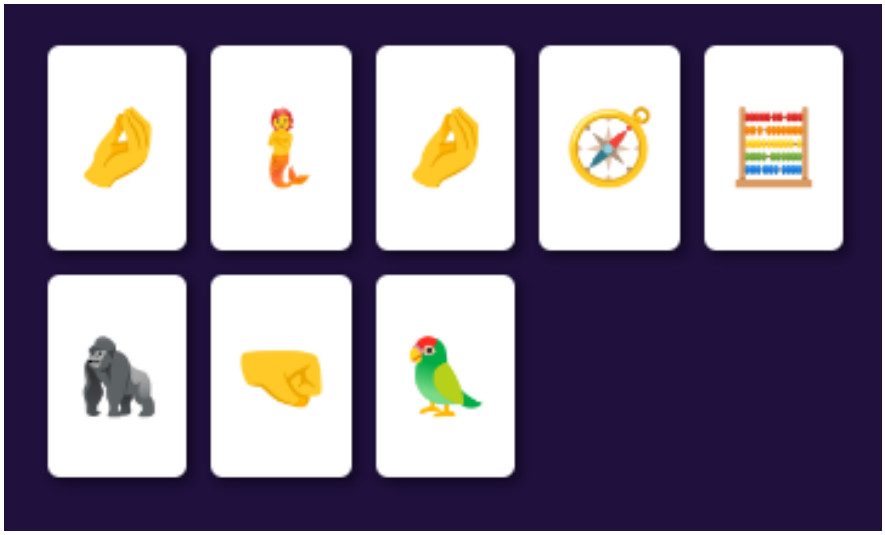


FIGURE 7 – Afficher des cartes

3 LE JEU

3.1 MEMORY

```
import { ControllerMemory } from "../controllers/controller-memory.js";
import { Card } from "./card.js";

export class Memory {
    #cards;
    constructor(){
        this.#cards = [];
    }

    newGame(pairsNumer){
        let valeur = "0x1F90C";
        for(let i = 0; i < 2*pairsNumer; i++){
            valeur = parseInt(valeur);
            if((i%2 === 0)&&(i !== 0)){
                valeur++;
            }
            this.#cards.push(new Card(valeur));
        }
    }
    getCardsNumber(){
        return this.#cards.length;
    }
    getCard(index){
        return this.#cards[index];
    }
}
```

FIGURE 8 – memory.js

La classe Memory possède un tableau de cartes qui est initialisé comme vide dans le constructeur. La méthode newGame prend en paramètre le nombre de paires de cartes à créer. Comme il faut créer les cartes dans l'ordre, la valeur de la première carte est donc de 0x1F90C. Ensuite, on rentre dans un boucle for qui regarde si la valeur de i est paire ou non. Si oui alors on change la valeur de la carte pour la suivante sinon on garde la même valeur. Cela permet de créer 2 fois la même carte à la suite. Pour ne pas changer la valeur au départ, il faut également regarder si i est différent de 0. Pour changer la valeur de la carte, on convertie la valeur en int que l'on incrémente de 1. Une fois la carte créer, on l'ajoute à la suite du tableau de carte. Le méthode getCardsNumber retourne le nombre de cartes dans le tableau et la méthode getCard retourne la carte située à la position donnée en paramètre.

3.2 CONTROLLER

```
import { Card } from "../models/card.js";
import { Memory } from "../models/memory.js";
import { Notifier } from "../patterns/notifier.js";

export class ControllerMemory extends Notifier
{
    #memory;
    constructor()
    {
        super();
        this.#memory = new Memory();
    }
    /*get card(){
        return this.#card;
    }*/
    get memory(){
        return this.#memory;
    }
    newGame(){
        this.#memory.newGame(10);
        this.notify();
    }
}
```

FIGURE 9 – controller-memory.js

On a retiré l'attribut card et la méthode createCard de ControllerMemory. On créer un attribut privé memory que l'on instancie dans le constructeur. On créer également un getter qui retourne le memory. Pour finir, on instancie la méthode newGame qui appelle la méthode newGame de memory pour 10 paires de cartes et qui une fois cela fait notifie les observateurs du controller.

3.3 VIEW

```
import { Observer } from "../patterns/observer.js";

export class ViewMemory extends Observer
{
    #controllerMemory;

    constructor(controllerMemory)
    {
        super();

        this.#controllerMemory = controllerMemory;
        this.#controllerMemory.addObserver(this);
    }

    notify()
    {
        this.displayCards();
    }
    displayCard(card){
        const cards = document.querySelector(".cards");
        let enfant = document.createElement("div");
        enfant.classList.add("card");
        enfant.innerHTML = "&#x" + (card.value).toString(16);
        cards.appendChild(enfant);
    }
    displayCards(){
        for(let i = 0; i < this.#controllerMemory.memory.getCardsNumber(); i++){
            this.displayCard(this.#controllerMemory.memory.getCard(i));
        }
    }
}
```

FIGURE 10 – view-memory.js

On modifie la fonction displayCard pour qu'elle affiche uniquement la carte qui lui est passée en paramètre. Le fonctionnement est similaire à displayCard d'avant sauf que l'on récupère la valeur de la carte qui est transmit au lieu de passer par le getter du controller. On créer la méthode displayCards qui affichera toutes les cartes qui sont dans le tableau cards de memory.

3.4 APPLICATION

```
this.#controllerMemory.newGame();
```

FIGURE 11 – application-memory.js

Dans application-memory, on remplace uniquement l'appel de createCard par newGame. Avec l'ensemble de ces modifications, on obtient :

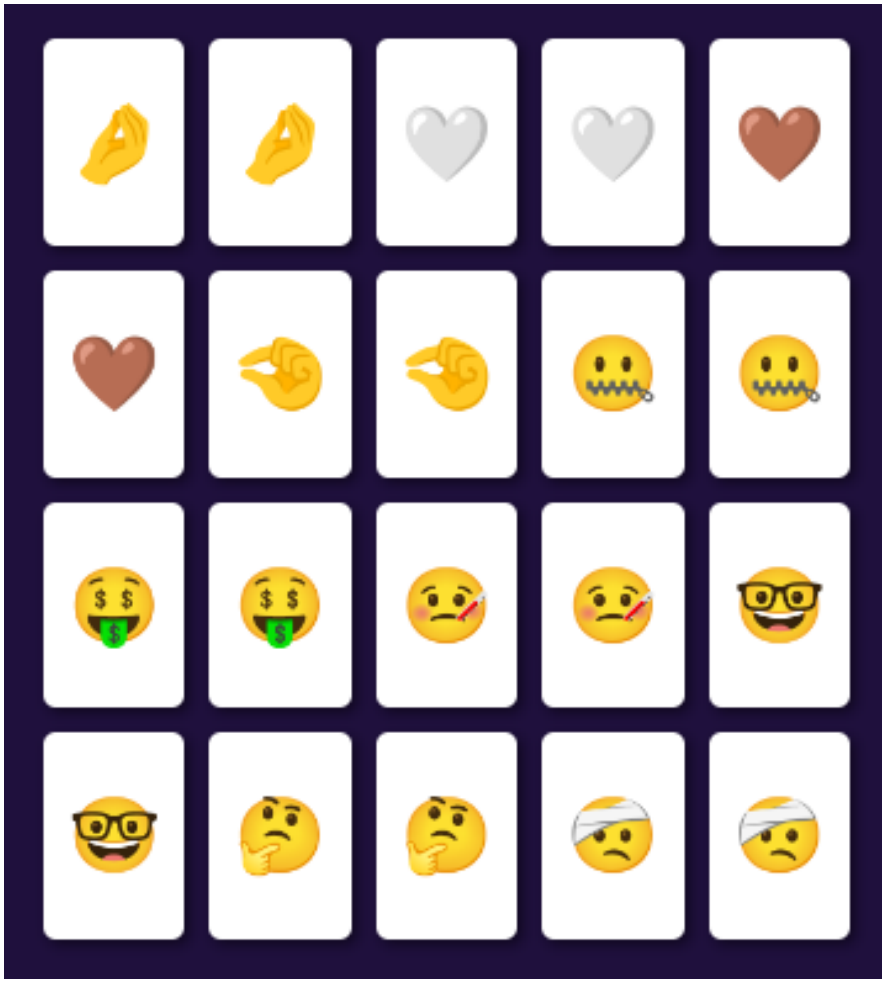


FIGURE 12 – Affichage de 10 paires de cartes dans l'ordre

4 UN PEU D'ALEATOIRE

Pour ajouter l'aléatoire, il suffit de rajouter une ligne dans la fonction newGame :

```
newGame(pairsNumer){
    let valeur = "0x1F90C";
    for(let i = 0; i < 2*pairsNumer; i++){
        valeur = parseInt(valeur);
        if((i%2 === 0)&&(i !== 0)){
            valeur++;
        }
        let emplacement = Math.floor(Math.random()*this.#cards.length);
        this.#cards.splice(emplacement, 0, new Card(valeur));
    }
}
```

FIGURE 13 – Modification de la fonction newGame

Avec splice, on place la carte créée dans le tableau entre la première et la dernière carte. On obtient donc :

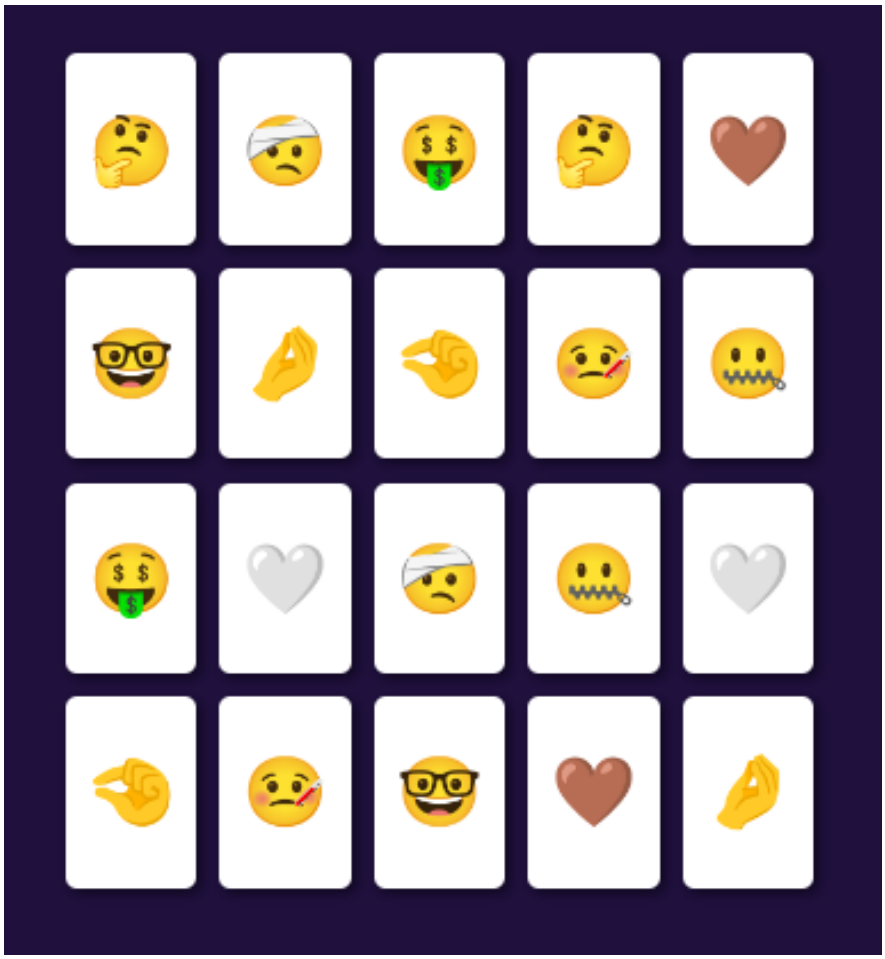


FIGURE 14 – Affichage de 10 paires de cartes mélangées

5 SERIALISATION / DESERIALISATION

5.1 CARD

```
toData(){
    const myCard = {
        value: this.value,
    }
    return myCard;
}
```

FIGURE 15 – Méthode toData de card.js

La méthode toData de card.js retourne un objet JavaScript de base contenant la valeur de la carte.

5.2 MEMORY

```
toData(){
    let cards = [];
    for(let i = 0; i < this.getCardsNumber(); i++){
        cards.push(this.getCard(i).toData());
    }
    let tableau = {
        cards,
    }
    return tableau;
}
```

FIGURE 16 – Méthode toData de memory.js

La méthode toData de memory.js créer un tableau vide pour accueillir tous les objets JavaScript. Ensuite pour toutes les cartes présentes dans le memory, on rajoute dans le tableau l'objet JS créé par la méthode toData de la classe Card. Une fois cela fait, on créer un objet JS qui prend le tableau d'objet JS et on le retourne.

5.3 CONTROLLER

```
newGame(){
    this.#memory.newGame(10);
    this.notify();
    this.saveGame();
}
saveGame(){
    localStorage.setItem("memory", JSON.stringify(this.memory.toData()));
}
```

FIGURE 17 – Méthode saveGame de controller-memory.js

La méthode saveGame de controller-memory.js enregistre les informations du memory dans le localStorage du navigateur. Si l'on regarde dans l'onglet Application du navigateur on peut voir :

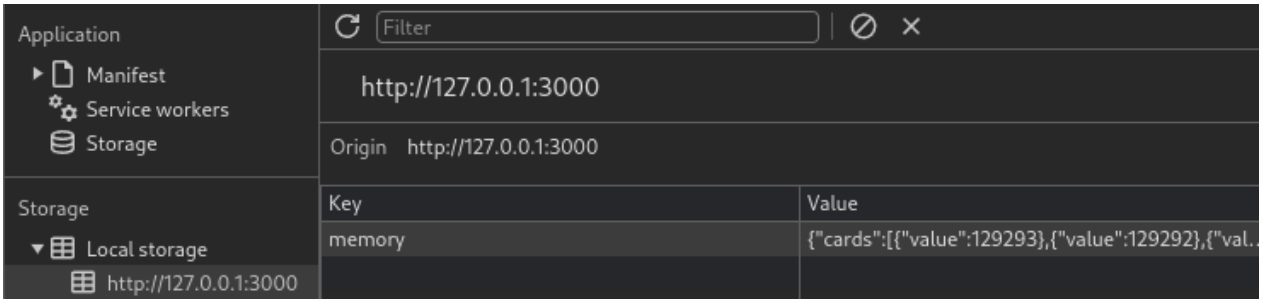


FIGURE 18 – LocalStorage du navigateur

On voit bien que les données sont enregistrées dans le navigateur sous le bon format.

6 CHARGEMENT

6.1 MEMORY

```
fromData(myData){
    this.#cards.splice(0, this.getCardsNumber());
    for(let i = 0; i < myData.cards.length; i++){
        this.#cards.push(myData.cards[i]);
    }
    return this.#cards;
}
```

FIGURE 19 – Méthode fromData de memory.js

Dans la méthode fromData, on commence par supprimer tous les éléments du tableau en utilisant splice. On donne l'indice du premier élément à supprimer donc 0 et le nombre d'éléments à supprimer donc tous les éléments. Ensuite, pour chaque élément du tableau récupéré dans le LocalStorage, on donne la carte récupérée au tableau de cartes de memory.

6.2 CONTROLLER

```
loadGame(){
    const myData = localStorage.getItem("memory");
    if(myData){
        this.#memory.fromData(JSON.parse(myData));
        this.notify();
        return true;
    }
    else{
        return false;
    }
}
start(){
    const chargement = this.loadGame();
    if(chargement === false){
        this.newGame();
    }
}
```

FIGURE 20 – Méthode loadGame et start de controller-memory.js

La méthode loadGame récupère les informations de localStorage puis si des données sont récupérées alors on les transmet à fromData en utilisant JSON.parse pour récupérer les objets sinon on retourne false. La méthode start appelle la fonction loadGame. Si des données étaient présentes sur le navigateur alors c'est que l'on a récupéré une partie en cours sinon, c'est qu'il faut lancer une nouvelle partie.

6.3 APPLICATION

```
this.#controllerMemory.start();
```

FIGURE 21 – newGame dans application-memory.js

Avec toutes ces modifications, on peut ouvrir plusieurs onglets différents et les recharger sans perdre sa progression comme on peut le voir sur la capture ci-dessous :

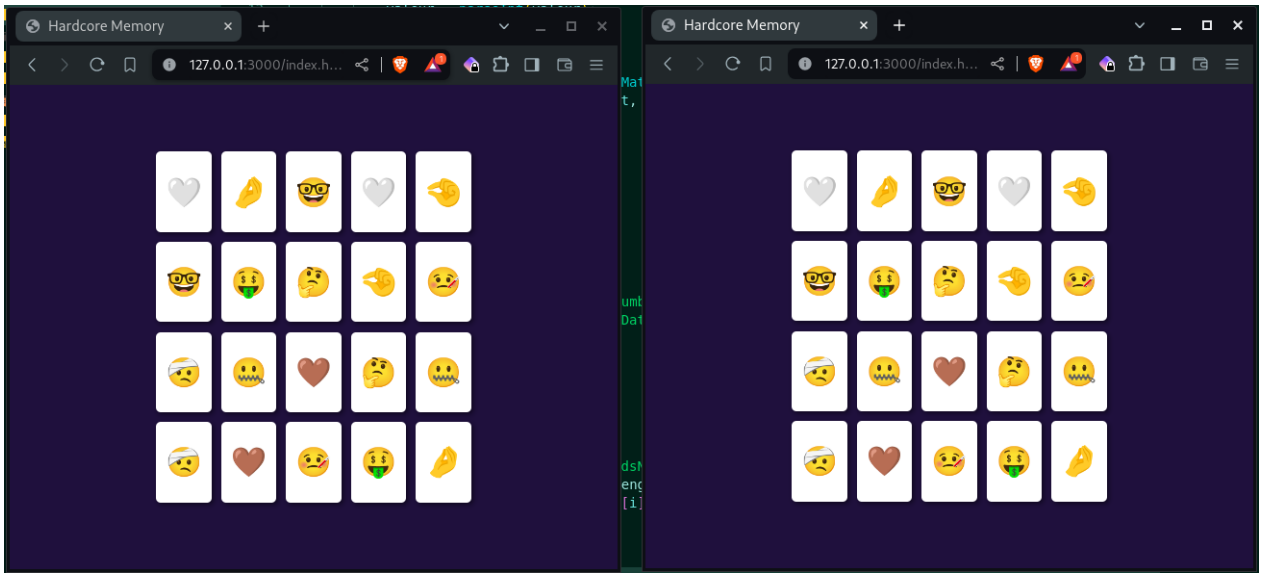


FIGURE 22 – Plusieurs parties chargées

6.4 SESSIONSTORAGE

Si l'on remplace les localStorage par sessionStorage dans l'ensemble du code alors on peut jouer une partie sur un onglet et recharger la page sans la perdre, mais si on ouvre un autre onglet alors la partie recommence à 0.