Introduction

JavaScript++

Présentation du NodeJS

JavaScript++

Arrow Functions: fonctions fléchées

JavaScript Asynchrone

Fonction

Normale

```
function add(x, y)
{
        return x + y;
 }
```

fléchée (ES6)

```
const sum=(x, y)=> { return x+y; };
```

```
const sum=(x, y)=> x+y;
```

Fonctions fléchée

```
// Make a copy of an array with null elements removed.
let filtered = [1,null,2,3].filter(x => x !== null);          // filtered == [1,2,3]

// Square some numbers:
let squares = [1,2,3,4].map(x => x*x);                        // squares == [1,4,9,16]


const f = x => { return { value: x }; };                      // Good: f() returns an object

const g = x => ({ value: x });                                // Good: g() returns an object

const h = x => { value: x };                                  // Bad: h() returns nothing

const i = x => { v: x, w: x };                                // Bad: Syntax Error
```

JavaScript

Synchrone

- Exécution continue jusqu'à la fin
- Appels et communications bloquants

Asynchrone

- Exécution discontinue
- Attente d'un événement ou des données
- Appels et communication non bloquants
- *event-driven*

JavaScript Asynchrone

- Callbacks
  - Timers
  - Events
  - Network Events
- Promises
- async and await

Callbacks

- Callback est une fonction A définie puis passée à une autre fonction B.
- La deuxième fonction B invoquera (”call back”) la première A quand une **condition** est satisfaite ou un **évènement** aura lieu.

- **Timers**
- **Events**
- **NetworkEvents**

Timers

- La façon asynchrone la plus simple est l'éxécution d'un code après un certain temps
- let updateTimeout = setTimeout(checkForUpdates, 60000);
- clearTimeout(updateTimeout);
- setTimeout(() => { console.log("Ready..."); }, 1000);
- setTimeout(() => { console.log("set..."); }, 2000);
- setTimeout(() => { console.log("go!"); }, 3000);

- let updateIntervalId = setInterval(checkForUpdates, 60000);
- let clock = setInterval(() => { *// Once a second: clear the console and print the current time*

```
        console.clear();
        console.log(new Date().toLocaleTimeString());
  }, 1000);
```

- setTimeout(() => { clearInterval(clock); }, 10000); *// After 10 seconds: stop the repeating code above.*

Events, Network Events

- Events
  - `let okay = document.querySelector('#confirmUpdateDialog button.okay');`
  - `okay.addEventListener('click', applyUpdate);`
- Network Events

```
function getCurrentVersionNumber(versionCallback) { // Note callback argument

    let request = new XMLHttpRequest(); // Make a scripted HTTP request to a backend version API
    request.open("GET", "http://www.example.com/api/version");

    request.send();

    request.onload = function() {// Register a callback that will be invoked when the response arrives
    if (request.status === 200) {// If HTTP status is good, get version number and call callback.

        let currentVersion = parseFloat(request.responseText);

        versionCallback(null, currentVersion);

    }else{// Otherwise report an error to the callback

    versionCallback(response.statusText, null);

    } };

    request.onerror = request.ontimeout = function(e) // Register another callback that will be invoked for network errors

    { versionCallback(e.type, null);

}; }
```

Promises

- Promise est un objet qui représente le résultat d'une opération asynchrone.
- Résout les problèmes des callbacks:
  - callbacks imbriqués
  - gestion des erreurs
- Représente le résultat future d'une seule opération asynchrone
  - ne peut pas remplacer setInterval()!

Promises, Exemple

```
function wait(duration) {
    // Create and return a new Promise
    return new Promise((resolve, reject) => { // These control the Promise
        // If the argument is invalid, reject the Promise
        if (duration < 0) {
            reject(new Error("Time travel not yet implemented"));
        }
        // Otherwise, wait asynchronously and then resolve the Promise.
        // setTimeout will invoke resolve() with no arguments, which means
        // that the Promise will fulfill with the undefined value.
        setTimeout(resolve, duration);
    });
}
wait(1000).then( () => console.log("done");
wait(-1) .then( () => console.log("done");
```

Promises, Gestion des erreurs

- getJSON("/api/user/profile").then(displayUserProfile, handleProfileError);

- wait(1000)

  .**then**(() => console.log(“done”), (err) => console.log(“erreur: ”+err));

- wait(1000)

  .**then**(() => console.log(“done”))

  .**catch**((err) => console.log(“erreur: ”+err));

Promises, Etat

- Un promise a quatre états:
  - pending: en instance
  - settled, stable: une fois stabilisé, son état ne peut pas être changé
    - fulfilled, réalisé: le callback passé a ‘then’ est invoqué
    - rejected, rejecté: le callback passé a ‘**catch**’ est invoqué
  - resolved, résolu: mais pas réalisé et donc pas stable

Chaining promises: Sequentially

```
fetch(documentURL)                          // Make an HTTP request
    .then(response => response.json())      // Ask for the JSON body of the response
    .then(document => {                     // When we get the parsed JSON
        return render(document);            // display the document to the user
})
.then(rendered => {                         // When we get the rendered document
    cacheInDatabase(rendered);              // cache it in the local database.
})
.catch(error => handle(error));             // Handle any errors that occur
```

Promises in Parallel: Promise.all()

```
// We start with an array of URLs
const urls = [ /* zero or more URLs here */ ];
// And convert it to an array of Promise objects
promises = urls.map(url => fetch(url).then(r => r.text()));
// Now get a Promise to run all those Promises in parallel
Promise.all(promises)
    .then(bodies => { /* do something with the array of strings */ })
    .catch(e => console.error(e));
```

Promises in Parallel: Promise.allSettled()

```
Promise.allSettled([Promise.resolve(1), Promise.reject(2), 3]).then(results => {
    results[0] // => { status: "fulfilled", value: 1 }
    results[1] // => { status: "rejected", reason: 2 }
    results[2] // => { status: "fulfilled", value: 3 }
});
```

async/await

- Introduits avec ES2017
- Simplifie l'utilisation des promises
- Permet d'écrire un code asynchrone qui rassemble beaucoup à un code normal
  - lisibilité
  - try/catch
  - boucle
- Blocage du code en attendant une réponse ou un évènement

example

```
async function getHighScore() {
    let response = await fetch("/api/user/profile");
    let profile = await response.json();
    return profile.highScore;
}
```