JESUÏTES educació

DAWM06UF4
COMUNICACIÓ ASÍNCRONA CLIENT-SERVIDOR

UF4.2.2 PROMISES I FUNCIONS ASÍNCRONES

CFGS Desenvolupament d'Aplicacions Web
M06. Desenvolupament web en entorn client
Fundació Jesuïtes Educació - Escola del Clot
Sergi Grau sergi.grau@fje.edu

OBJECTIUS

- Entendre les característiques de les promeses
- Comprendre com funcionen les noves funcions asíncrones de JS de ES6
- Utilitzar promises i funcions asíncrones

QUÈ ÉS UNA PROMESA ?

- L'objecte Promise (Promesa) és usat per a computacions asíncrones.
- Una promesa representa un valor que pot estar disponible ara, en el futur, o mai.
- new Promise (/ * executor * / function (resoldre, rebutjar) {...});

DESCRIPCIÓ

- Una Promesa és utilitzat per a computacions asíncrones, per a un valor no necessàriament conegut en el moment que és creada la promesa. El valor pot estar disponible ara, en el futur o mai.

- Permet associar controladors que actuaran asíncronament sobre un eventual valor en cas d'èxit, o la raó d'error en cas d'un error.

- Això permet que mètodes asíncrons retornin valors com si fossin síncrons: en lloc de immediatament retornar el valor final, el mètode asíncron retorna una promesa de subministrar el valor en algun moment en el futur.

QUE APORTEN LES PROMESES?

- El codi JS s'executa en un sol fil, el que significa que dos fragments de seqüència de comandaments no es poden executar al mateix temps; han d'executar-se un darrere l'altre.

- En els navegadors, JS comparteix un fil amb una càrrega de treball que es diferent d'un navegador a un altre.

- Però en general JS es troba a la mateixa cua que el render del DOM, l'actualització d'estils, i les accions de l'usuari de manipulació de la interfície. L'activitat en una d'aquestes tasques retarda les altres.

- Les funcions modernes retornen un objecte promise a què pots adjuntar funcions de retorn (callbacks)

PRIMERA PROMESA


```
let promesa = new Promise((resolve, reject) => {
    // Fem una crida a resolve (...) quan el que estàvem fent finalitza amb èxit, i reject (...)
quan falla.
    // En aquest exemple, fem servir setTimeout (...) per simular codi asíncron.
    // A la vida real, probablement facis servir alguna cosa com XHR o una API HTML5.
    setTimeout(function () {
        if (Math.random() * 10 > 5) resolve("OK!");
        reject('problema');
    }, 250);
});

promesa.then((enCasExit) => {
    // succesMessage és el que sigui que passem a la funció resolve (...) de dalt.
    // No té per què ser un string, però si només és un missatge d'èxit, probablement ho sigui.
    // en aquest cas hi ha un catch que captura quan s'ha produït un error
    console.log("%c¡Sí! " + enCasExit, "color: yellow; text-transform: uppercase");
}).then((e) => {
    console.log("%cha acabat l'anterior", "color: orange; text-transform: uppercase");
}).catch((e) => {
    console.error("¡error! " + e);
});
console.info("això s'executa de manera seqüencial");
```

GARANTIES DE LES PROMESES

- A diferència de les funcions callback passades a el "vell estil", una promesa ve amb algunes garanties:

- Les funcions callback mai seran cridades abans de l'acabament de l'execució actual de l'bucle d'esdeveniments de JS.

- Les funcions callback afegides amb then () fins i tot després de l'èxit o fracàs de l'operació asíncrona seran cridades com es va mostrar anteriorment.

- Múltiples funcions callback poden ser afegides trucant a then diverses vegades. Cadascuna d'elles és executada una seguida de l'altra, en l'ordre en què van ser inserides.

- Una de les grans avantatges d'utilitzar promises és l'encadenament

ENCADENAMENT DE PROMESES

- Una necessitat comú és el executar dos o més operacions asíncrones seguides, on cada operació posterior s'inicia quan l'operació prèvia té èxit, amb el resultat de el pas previ.

- Podem aconseguir això creant una cadena d'objectes promises.

```
new Promise((resolver, reject) => {
    console.log("%cprimera tasca ", "color: yellow; text-transform: uppercase");
    resolver('primera');
})
.then((exit) => {
    console.log(`%csegona tasca que es fa després de ${exit}`, "color: yellow; text-transform:
uppercase");
    return 'segona';
})
.then((exit) => {
    console.log(`%ctercera tasca que es fa després de ${exit}`, "color: yellow; text-transform:
uppercase");
    return 'segona';
})
.then((anterior) => {
    throw new Error('Algo falló');
})
.catch((e) => {
    console.error("¡error! " + e);
    return 'error';
})
.then((anterior) => {
    console.log(`%ctasca final que es fa després de ${anterior}`, "color: yellow; text-transform:
uppercase");
});
```

COM ES FEIA ABANS?


```
hazAlgo(function(resultado) {
  hazAlgoMas(resultado, function(nuevoResultado) {
    hazLaTerceraCosa(nuevoResultado, function(resultadoFinal) {
      console.log('Obtenido el resultado final: ' +
resultadoFinal
    }, falloCallback);
  }, falloCallback);
}, falloCallback);
```

ESTATS D'UNA PROMESA

- Una Promesa es troba en un dels següents estats:

- **pendent** (pending): estat inicial, no complerta o rebutjada.

- **complerta** (Fulfilled): vol dir que l'operació es va completar satisfactòriament.

- **rebutjada** (rejected): vol dir que l'operació va fallar.

FUNCIONAMENT

- Una promesa pendent pot ser complerta amb un valor, o rebutjada amb una raó (error).

- Quan qualsevol d'aquestes dues succeeix, els mètodes associats, encolats pel mètode then de la promesa, són cridats.

- Si la promesa ja ha estat complerta o rebutjada en el moment que és annexat el seu corresponent gestor, el controlador serà cridat, de manera que no hi hagi una condició de carrera entre l'operació asíncrona sent completada i els controladors sent annexats

QUE APORTEN LES PROMESES?

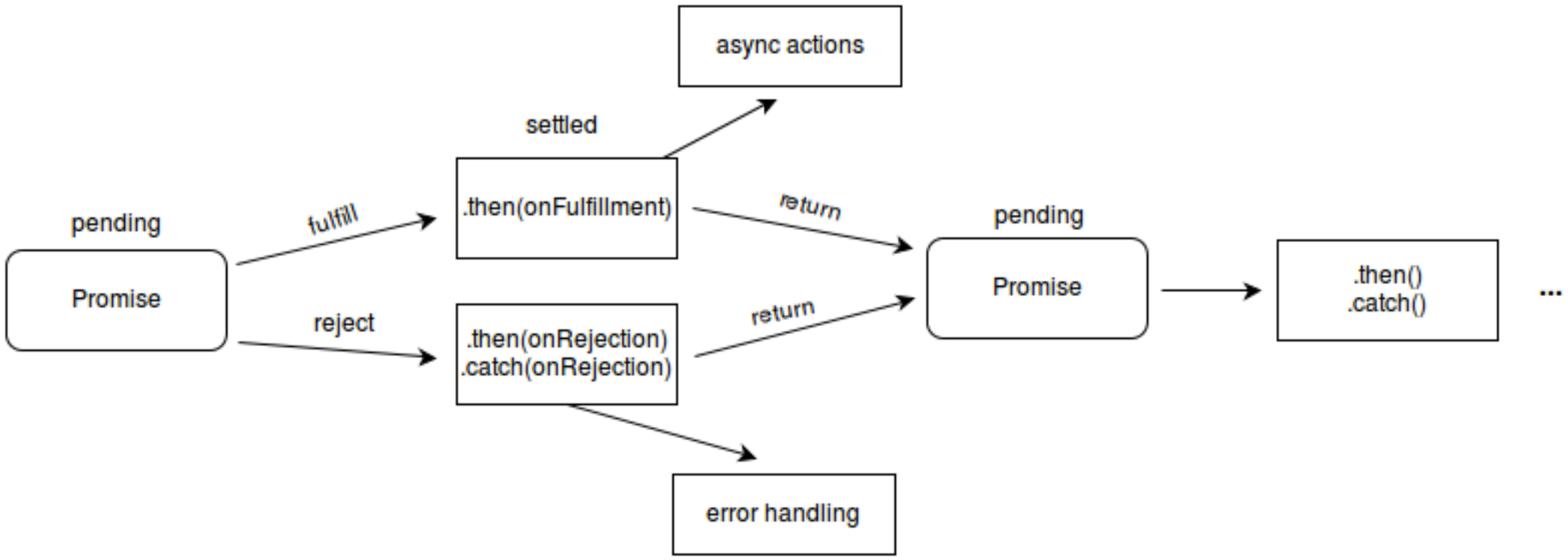
- El codi JS s'executa en un sol fil, el que significa que dos fragments de seqüència de comandaments no es poden executar al mateix temps; han d'executar-se un darrere l'altre.

- En els navegadors, JS comparteix un fil amb una càrrega de treball que es diferent d'un navegador a un altre.

- Però en general JS es troba a la mateixa cua que el render del DOM, l'actualització d'estils, i les accions de l'usuari de manipulació de la interfície. L'activitat en una d'aquestes tasques retarda les altres.

- Les funcions modernes retornen un objecte promise a què pots adjuntar funcions de retorn (callbacks)

FUNCIONAMENT



FUNCIONAMENT I EQUIVALENT SINCRON

```
PromesaQueFaQuelcom()
.then(resultat => fesQuelcomMes(valor))
.then(nouResultat => fesTerceraFuncio(nouResultat))
.then(resultatFinal => console.log(`hem tingut: ${resultatFinal}`))
.catch(error);

//SINCRON
try {
  let resultat = ferQuelcom();
  let nouResultat = fesQuelcomMes(resultat);
  let resultatFinal = fesTerceraFuncio(nouResultat);
  console.log(`hem tingut: ${resultatFinal}`);
} catch(error) {
  falloCallback(error);
}
```

PROMESES I FUNCIONS ASÍNCRONES

- En un món ideal, totes les funcions asíncrones tornarien promeses. Desafortunadament, algunes APIs encara esperen que se'ls passi callbacks amb resultat fallit / reeixit a la forma antiga

```
const espera = ms => new Promise(resol => setTimeout(resol, ms));

espera(10000).then(() => fesQuelcom("10 segundos")).catch(error);
```

COMPOSICIÓ, EN PARAL·LEL

- Promise.resolve () i Promise.reject () són dreceres per crear manualment una promesa resolta o rebutjada respectivament. Això pot ser útil a vegades.

- Promise.all () són Promise.race () són dues eines de composició per a executar operacions asíncrones en paral·lel.

- Espera a que totes elles finalitzin

```
Promise.all([func1(), func2(), func3()])
.then(([resultat1, resultat2, resultat3]) => { /* fer quelcom */ });
```

COMPOSICIÓ, EN PARAL·LEL

JESUÏTES educació
DAWM06UF4 sergi.grau@fje.edu

```
const promesa1 = Promise.resolve(3);
const promesa2 = 42;
const promesa3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'DAW2');
});
console.time('temps');
Promise.all([promesa1, promesa2, promesa3])
  .then(([resultat1, resultat2, resultat3]) => {
    console.log(resultat1, resultat2, resultat3);
    console.timeEnd('temps');
  });
console.info('ha de sortir abans, doncs el codi anterior espera a totes les promeses');
```

COMPOSICIÓ, RACE

+ retorna un valor en el moment que una de les promeses finalitza.

COMPOSICIÓ, RACE

JESUÏTES educació
DAWM06UF4 sergi.grau@fje.edu

```
const promesa1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'pimera');
});

const promesa2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'segona');
});

Promise.race([promesa1, promesa2]).then((valor) => {
  console.log(valor);
});
```

COMPOSICIÓ, SEQÜENCIAL

- En ECMAScript 2017, la composició seqüencial es pot fer simplement amb async/await

```
let resultat;
for (const f of [func1, func2, func3]) {
  resultat = await f(resultat);
}
```

COMPOSICIÓ, SEQÜENCIAL


```
const promesa1 = Promise.resolve(3);
const promesa2 = 42;
const promesa3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'DAW2');
});
console.time('temps');
Promise.all([promesa1, promesa2, promesa3])
  .then(([resultat1, resultat2, resultat3]) => {
    console.log(resultat1, resultat2, resultat3);
    console.timeEnd('temps');
  });
console.info('ha de sortir abans, doncs el codi anterior espera a totes les promeses');
```

FUNCIONS ASÍNCRONES

- En ECMAScript 2017, es van introduir les funcions asíncrones.

- Ens permet etiquetar les funcions com a asíncrones, i aquestes funcions retornaran promeses, resolent finalment el valor que es retorni (o rebutjant amb qualsevol error que es generi).

- A continuació, podem invocar aquestes funcions i esperar la resolució o el fracàs d'aquestes promeses. D'aquesta manera podem escriure codi que sembli més síncron del que realment és.

FUNCIONS ASÍNCRONES

+ Aquestes funcions asíncrones es resolen immediatament, però això no és massa útil.

+ Podem utilitzar **await** per esperar altres funcions o promeses asíncrones

ASYNC/AWAIT


```
async function divisioAsincrona(a, b, ms = 500) {
  const resultat = await divisioPromesa(a, b, ms);
  console.info(resultat);
  return resultat;
}
divisioAsincrona(3,3,3000); // es una crida asíncrona, quan tinguem el resultat es resoldrà
divisioAsincrona(2,3,2000); // es una crida asíncrona, quan tinguem el resultat es resoldrà
divisioAsincrona(1,3,1000); // es una crida asíncrona, quan tinguem el resultat es resoldrà
console.info("això s'executa de manera seqüencial");
```

```
30  ///més sobre asíncrones
31  const bar = async () => 42; // el codi d'aquesta funció asíncrona retorna una promesa
32  const bar2 = async () => {
33    const x=42;
34    console.log(x);
35    return x;
36  }
37
38  console.error(bar()); // es una crida asíncrona, quan tinguem el resultat es resoldrà al resolve NO es pot recuperar així
39  bar().then(e => console.log(e)); // es una crida asíncrona, quan tinguem el resultat es resoldrà
40  console.info(bar2()); // es una crida asíncrona, quan tinguem el resultat es resoldrà
41
42
43  console.info("això s'executa de manera seqüencial");
```

```
▶ ▼ Promise {<fulfilled>: 42} i                    M15_asincrones.js:38
    ▶ __proto__: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: 42
42                                                 M15_asincrones.js:34
▼ Promise {<fulfilled>: 42} i                      M15_asincrones.js:40
  ▶ __proto__: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: 42
això s'executa de manera seqüencial                M15_asincrones.js:43
42                                                 M15_asincrones.js:39
```

ASYNC / AWAIT ENCADENADES

- Una funció asíncrona pot tenir qualsevol quantitat d'expressions d'espera, cosa que la fa útil per indicar esdeveniments asíncrons però que s'executen en paral·lel.

- És important recordar que durant aquest temps, el motor JS i qualsevol altra interfície d'usuari **no estan bloquejats** i, per tant, poden respondre a les aportacions d'altres usuaris.

- Recordeu el nostre exemple de divisió encadenada? Aquí teniu aquest exemple escrit amb async / await

ASYNC/AWAIT ENCADENADES


```
function divisioPromesa(a, b, ms = 500) {
  return new Promise((resolve, reject) => {
      setTimeout(() => {
          if (b === 0) {
              reject(new Error("divisió per zero"));
          } else {
              resolve(a / b);
          }
      }, ms);
  });
}

async function divisioAsincrona(a, b, ms = 500) {
  const resultat = await divisioPromesa(a, b, ms);
  console.info(resultat);
  return resultat;
}

async function encadenadaAsincrona() {
  const r1 = await divisioAsincrona(900, 3),
        r2 = await divisioAsincrona(r1, 2),
        r3 = await divisioAsincrona(r2, 5);
  return r3;
}

encadenadaAsincrona();
```

ASYNC / AWAIT

- Atès que **les funcions asíncrones són promeses**, les podem tractar com a tals. De fet, al màxim nivell del nostre programa, això és tot el que podem fer, ja que **és il·legal esperar un resultat fora d'una funció asincronitzada**.

- Podem triar descartar la devolució (que serà una promesa) si volem (la funció asíncrona encara s'executarà) o podem tractar la devolució com una promesa i utilitzar-la llavors i capturar el resultat.

BIBLIOGRAFIA

+ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

+ https://developers.google.com/web/fundamentals/getting-started/primers/promises