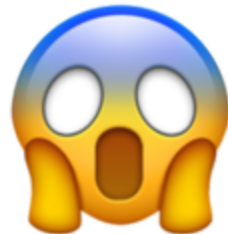


Programmation web - Client riche

# **La programmation asynchrone**



*Dans le navigateur, nos scripts JS tournent dans un seul thread : celui qui gère l'ensemble de l'affichage de notre page web*





**David Whittaker**

@rundavidrun



 Suivre

Apparently, a sufficient number of puppies can explain any computer science concept. Here we have multithreading:

 À l'origine en anglais

## *Multithreaded programming*





*Les entrées / sorties (I/O) sont asynchrones. JS peut passer à la suite du code pendant qu'il attend le retour d'une I/O*



```
setTimeout(() => console.log("2 seconds passed"), 2000)  
console.log("After setTimeout")
```



# Les callbacks

```
const getRandomAsync = cb => {  
  try {  
    const number = Math.random()  
  
    setTimeout(() => cb(null, number), 2000)  
  } catch (err) {  
    cb(err)  
  }  
}
```

```
getRandomAsync((err, result) => {  
  if (err) {  
    console.error(err)  
    return  
  }  
  
  console.log("The result is", result)  
})
```



```
getRandomAsync((err, result) => {
  if (err) {
    console.error(err)
    return
  }

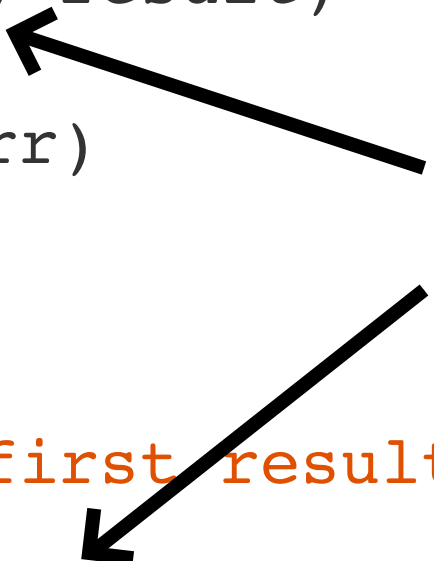
  console.log("The first result is", result)

  getRandomAsync((err, result) => {
    if (err) {
      console.error(err)
      return
    }

    console.log("The second result is", result)
  })
})
```

```
getRandomAsync((err, result) => {  
  if (err) {  
    console.error(err)  
    return  
  }  
  
  console.log("The first result is", result)  
  
  getRandomAsync((err, result) => {  
    if (err) {  
      console.error(err)  
      return  
    }  
  
    console.log("The second result is", result)  
  
    getRandomAsync((err, result) => {  
      if (err) {  
        console.error(err)  
        return  
      }  
  
      console.log("The third result is", result)
```

```
getRandomAsync((err, result) => {  
  if (err) {  
    console.error(err)  
    return  
  }  
  
  console.log("The first result is", result)  
  
  getRandomAsync((err, result) => {  
    if (err) {  
      console.error(err)  
      return  
    }  
  
    console.log("The second result is", result)  
  })  
})
```



Même nom, mais variables différentes

```
getRandomAsync((err, result) => {
```

```
  if (err) {  
    console.error(err)  
    return  
  }
```

← Gestion des erreurs  
dupliquée

```
  console.log("The first result is", result)
```

```
  getRandomAsync((err, result) => {
```

```
    if (err) {  
      console.error(err)  
      return  
    }
```

```
    console.log("The second result is", result)  
  })
```

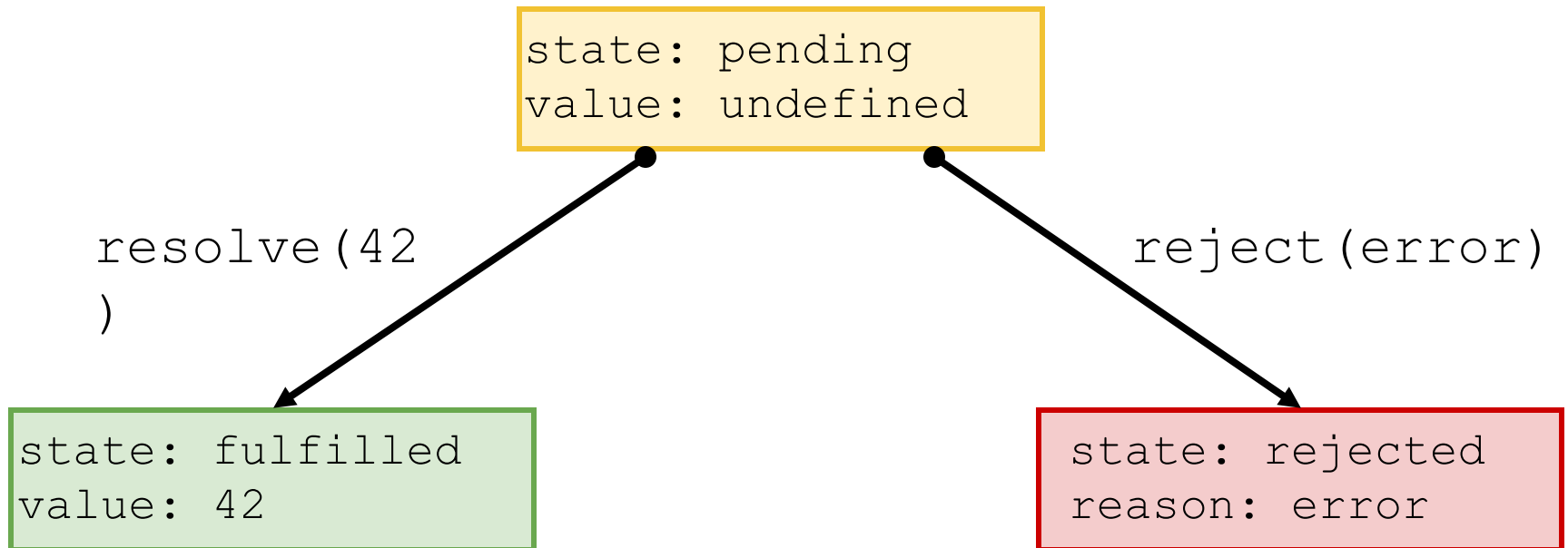
```
})
```

# Les Promises

```
getRandomAsync( )  
  .then(result => {  
    console.log("The first result is", result)  
  })  
  .then(getRandomAsync)  
  .then(result => {  
    console.log("The second result is", result)  
  })  
  .then(getRandomAsync)  
  .then(result => {  
    console.log("The third result is", result)  
  })  
  .catch(err => console.error(err))
```

```
const getRandomAsync = () => {  
  return new Promise((resolve, reject) => {  
    try {  
      const number = Math.random()  
  
      setTimeout(() => resolve(number), 2000)  
    } catch (err) {  
      reject(err)  
    }  
  })  
}
```

```
new Promise((resolve, reject) => ...)
```





**async/await**

```
const main = async () => {  
  try {  
    const result1 = await getRandomAsync()  
    console.log("The first result is", result1)  
  
    const result2 = await getRandomAsync()  
    console.log("The second result is", result2)  
  
    const result3 = await getRandomAsync()  
    console.log("The third result is", result3)  
  } catch (err) {  
    console.error(err)  
  }  
}  
  
main()
```



*Le mécanisme aysnc/await se base sur les Promises. Une fonction déclarée comme « async » renverra une Promise. Sa valeur de retour sera la valeur de résolution de la Promise. Si elle lève une erreur ou renvoie un appel à Promise.reject, la promise sera rejected.*

```
async function asyncFn() {  
    return 42  
}
```

```
console.log(asyncFn())  
// => Promise { <state>: "fulfilled", <value>: 42 }
```

```
console.log(await asyncFn())  
// => 42
```

```
async function asyncThrow() {  
    throw new Error("error message")  
}
```

```
console.log(asyncThrow())  
// => Promise { <state>: "rejected" }
```

```
console.log(await asyncThrow())  
// => Error: error message
```

# Les requêtes asynchrones



*Une page web a très souvent besoin de faire des appels asynchrones vers un serveur. Le but est de récupérer des données en arrière plan, sans affecter la navigation de l'utilisateur*

*Pour ça, nous allons utiliser [l'API fetch](#), qui est basée sur les promises*



*Ces requêtes sont souvent appelées « requêtes AJAX » (pour Asynchronous Javascript And XML) ou XHR (pour « [XMLHttpRequest](#) », l'objet historique sous-jacent). Toutefois, les données reçues en réponse ne sont pas forcément du XML. On utilise même plutôt JSON la plupart du temps.*



*JSON (pour JavaScript Object Notation) est un format de données textuelles qui s'inspire de la syntaxe des objets en JS. Voir [la page Wikipédia](#) correspondante pour plus d'informations*



```
fetch( "url/to/resource" )  
  .then(response => {  
    // On parse la réponse  
    return response.text()  
  })  
  .then(text => {  
    // On affiche le texte obtenu  
    document.body.append(text)  
  })  
  .catch(error => {  
    document.body.append(error.message)  
  })
```

```
try {  
  const response = await fetch("url/to/resource")  
  const text = await response.text()  
  document.body.append(text)  
} catch (error) {  
  document.body.append(error.message)  
}
```



[https://codesandbox.io/embed/zqq8kykov4?  
autoresize=1&hidenavigation=1](https://codesandbox.io/embed/zqq8kykov4?autoresize=1&hidenavigation=1)

**Des questions ?**



TD