



# L'asynchrone en JS sans le cringe

Une présentation de Christophe Porteneuve à DevFest Lille 2023

# whoami

```
1 const christophe = {  
2   family: { wife: 'Élodie', sons: ['Maxence', 'Elliott'] },  
3   city: 'Paris, FR',  
4   company: 'Delicious Insights',  
5   trainings: ['TypeScript', 'React PWA', 'Node.js', 'ES Total'],  
6   jsSince: 1995,  
7   claimsToFame: [  
8     'Prototype.js',  
9     'script.aculo.us',  
10    'Bien Développer pour le Web 2.0',  
11    'NodeSchool Paris',  
12    'Paris Web',  
13    'dotJS'  
14  ]  
15}
```

# Usual Suspects

# async sans await ni enrobage promesse voulu

```
1  export async function getAllRoles(req, res) {
2      res.send({ data: ROLES })
3  }
4
5  export async function getAllRolesWithAbilities(req, res) {
6      const data = computeCombinedAbilitiesByRole(Object.keys(ROLES))
7      res.send({ data })
8  }
9
10 async function create(createData) {
11     return GeneralParameter.create(createData)
12 }
```

# async sans await ni enrobage promesse voulu : fix

```
1 export function getAllRoles(req, res) {
2   res.send({ data: ROLES })
3 }
4
5 export function getAllRolesWithAbilities(req, res) {
6   const data = computeCombinedAbilitiesByRole(Object.keys(ROLES))
7   res.send({ data })
8 }
9
10 function create(createData) {
11   return GeneralParameter.create(createData)
12 }
```

# map à tort sur un callback async

```
1 const mailSequence = mails.map(async (mail) => await sendMail(mail))
```

# map à tort sur un callback async : fix n°1

```
1 const mailSequence = mails.map(async (mail) => await sendMail(mail))
```

↓

```
1 const mailSequence = mails.map((mail) => sendMail(mail))
```

(Éventuellement, si tu peux **garantir** que `sendMail` n'utilise que son premier argument, et ne sera donc pas gêné par des arguments supplémentaires :)

```
1 const mailSequence = mails.map(sendMail)
```

# map à tort sur un callback async : fix n°2

```
1 const mailSequence = mails.map(async (mail) => await sendMail(mail))
```

↓

Parallélisé (court-circuit sur 1ère erreur temporelle) :

```
1 const mailSequence = await Promise.all(mails.map((mail) => sendMail(mail)))
```

Séquencé (court-circuit sur première erreur itérative) :

```
1 const mailSequence = []
2 for (const mail of mails) {
3   mailSequence.push(await sendMail(mail))
4 }
```

# return await superflu (ou son équivalent)

```
1  async function getUserId(id) {
2    const user = await User.findByPk(...)
3    return user
4  }
5
6  async function upsertSetting(formObject) {
7    // ...
8    return noRecord ? await create(fields) : await update(fields)
9  }
10
11 async function renewToken({ commit }) {
12   const { token, refreshToken } = await renewToken()
13   const setToken = await commit('setToken', { token, refreshToken })
14   return setToken
15 }
```

## return await superflu (ou son équivalent) : fix

```
1  function getUserId(id) {
2    return User.findByPk(...)
3  }
4
5
6  function upsertSetting(formObject) {
7    // ...
8    return noRecord ? create(fields) : update(fields)
9  }
10
11 async function renewToken({ commit }) {
12   const { token, refreshToken } = await renewToken()
13   return commit('setToken', { token, refreshToken })
14 }
```

# Contre-exemple pour `await` suivi de `return`

Si on transforme le résultat (par exemple en ne renvoyant qu'une partie), on doit forcément faire un `await` local pour ensuite transformer avant de renvoyer :

```
1  async function logIn(req, res) {  
2      const { token } = await attemptLogIn(req.body)  
3      return token  
4  }
```

# Le cas légitime pour `return await`

```
1  async function process(items) {
2    try {
3      ...
4      return await subProcess(items)
5    } catch (error) {
6      console.error(`Couldn't run subprocess for ${items}: ${error}`)
7      throw error // Or possibly provide a fallback value, or something.
8    }
9 }
```

Si on peut traiter localement l'erreur que la promesse est susceptible de lever, il faut un `await` pour que celle-ci soit levée au sein du `try...catch`.

# Séquencer au lieu de paralléliser

Une parallélisation n'est pas toujours préférable, mais quand elle l'est, séquencer « par défaut » laisse de la performance sur la table.

```
1  async function bulkCreateOrUpdate(data) {  
2    ...  
3    for (let i = 0; i < data.length; i++) {  
4      await User.upsert(data[i], { transaction })  
5    }  
6    ...  
7 }
```

# Séquencer au lieu de paralléliser : fix

Cadeau bonus : ça permet dans ce cas précis de virer **cette fichue boucle numérique** qui aurait dû être une jolie `for ... of`. Y'avait rien qu'allait dans ce code.

```
1  async function bulkCreateOrUpdate(data) {  
2    ...  
3    await Promise.all(data.map((userData) => User.upsert(userData, { transacti  
4    ...  
5  })
```

Et si on est **limités dans la parallélisation** (*ex. connexions à la base de données*), pas de souci, on a des solutions pour plafonner :

```
1  import { map as cappedAll } from 'awaiting'  
2  
3  await cappedAll(data, 5, (userData) => User.upsert(userData, { transaction })
```

# Mélangier `.then()` et `async / await`

Non mais , quoi.

```
1  async function down(queryInterface, Sequelize) {
2    const transaction = await queryInterface.sequelize.transaction()
3    try {
4      await queryInterface
5        .bulkDelete('user_org_roles', null, {})
6        .then(() => queryInterface.bulkDelete('user', null, {}))
7        .then(() => queryInterface.bulkDelete('person', null, {}))
8      await transaction.commit()
9    } catch (error) {
10      await transaction.rollback()
11      throw error
12    }
13 }
```

# Mélangier `.then()` et `async / await` : fix

Utilise juste `async / await`, enfin !

```
1  async function down(queryInterface, Sequelize) {
2    const transaction = await queryInterface.sequelize.transaction()
3    try {
4      await queryInterface.bulkDelete('user_org_roles')
5      await queryInterface.bulkDelete('user')
6      await queryInterface.bulkDelete('person')
7      await transaction.commit()
8    } catch (error) {
9      await transaction.rollback()
10     throw error
11   }
12 }
```

# ZOMGWTFBBQ

J'étais tombé sur ce multi-récidiviste :

```
1  async function deleteUser(req, res) {  
2      return userService.destroyUser(req.params.id).then(async (user) => {  
3          res.status(200).send(user)  
4      })  
5  }
```

Purée, y'a **rien** qui va.

```
1  async function deleteUser(req, res) {  
2      const user = await userService.destroyUser(req.params.id)  
3      res.status(200).send(user)  
4  }
```

# Utiliser des chaînes de promesses manuelles

C'est une variante « moins grave » du mélange des styles, mais c'est quand même *so 2015*. Je suis tombé sur ce clusterfuck récemment :

```
1 export function findAllUsers(query) {  
2     ...  
3     return User.findAndCountAll(...)  
4         .then(ensureAtLeastOne)  
5         .catch((error) => {  
6             throw new Error(error)  
7         })  
8     }  
9 }
```

- Il y a un risque de **double mode d'erreur** (synchrone et asynchrone).
- Ce `catch` est aussi utile que le H de Hawaï.
- Les chaînes de promesses restent **plus dures à orchestrer** (pas de structures de contrôle).

# Aparté : *scope juggling* dans une chaîne manuelle

```
1  function getUsersLastPost(userId) {
2      let user
3      let post
4      return User.findByPk(userId)
5          .then((u) => {
6              user = u
7              return u.posts.sort('-createdAt').findOne()
8          })
9          .then((p) => {
10             post = p
11             return p.comments.sort('-createdAt').limit(10).find()
12         })
13         .then((comments) => {
14             return { user, post, comments }
15         })
16     }
```

# Utilise `async` / `await`

(Je sais, je me répète.)

```
1  export async function findAllUsers(query) {  
2    ...  
3    return ensureAtLeastOne(await User.findAndCountAll())  
4  }  
5  
6  // Sans doute optimisable par eager-loading, mais c'est un autre sujet,  
7  // et on ne fait pas de N+1 en plus ici, alors bon.  
8  async function getUsersLastPost(userId) {  
9    const user = await User.findByPk(userId)  
10   const post = await user.posts.sort('-createdAt').findOne()  
11   const comments = await post.comments.sort('-createdAt').limit(10).find()  
12   return { user, post, comments }  
13 }
```

# Enrobage manuel à tort ou superflu

Alias « Eeeeeh j'ai découvert `Promise.resolve()` et `Promise.reject()` ! »

```
1  async (error) => {  
2    ...  
3    return Promise.reject(error)  
4 }
```

Mais **pourquoi** ?! Ta fonction `async` enrobe automatiquement son code en promesse. Sers-t'en !

```
1  async (error) => {  
2    ...  
3    throw error  
4 }
```

# Contre-exemple : enrobage manuel intentionnel

Fonctions non `async` car elles n'utilisent pas `await`, mais censées renvoyer des promesses :

```
1 http.interceptors.response.use(  
2   (response) => {  
3     store.commit('loading/setLoading', false)  
4     return Promise.resolve(response.data)  
5   },  
6   (error) => {  
7     store.commit('loading/setLoading', false)  
8     return Promise.reject(error)  
9   }  
10 )
```

- Le `Promise.resolve` est plus explicite que de déclarer la fonction `async` avec un `return response.data`.
- Le `Promise.reject` est plus performant que de déclarer la fonction `async` avec un `throw error`.

# En résumé...

- `async` / `await` est **nettement supérieur** aux chaînes manuelles
- `await suspend`, il ne *bloque* pas.
- `await` est possible en racine de module (*Top-Level Await*, ou TLA) et dans le corps immédiat d'une fonction `async`.
- Toute fonction peut être `async`.
- Les fonctions `async` enrobent implicitement leurs corps comme promesse.
- Tu ne devrais jamais faire un `return await` (ou équivalent) hors d'un `try...catch`

# Viens nous voir !

On est sympas.

Chez Delicious Insights, on fait des **formations qui déchirent tout**, notamment sur TypeScript, 100% de JS pur, React et les PWA, Node.js et Git.

*(Franchement, elles envoient du bois.)*

On peut aussi venir gronder ton archi / ta codebase (mais gentiment), voire réaliser tes **preuves de concept** pour toi, en mode pas jetable du tout™.

À côté de ça, tu devrais **carrément** t'abonner à notre fabuleuse chaîne YouTube, qui déborde de tutos, cours, livestreams, talks en conférences, etc. et c'est évidemment **gratuit** !

# Budget très très serré ?

On a une super nouvelle pour toi. D'ailleurs, je l'annonce *en exclusivité mondiale™* ici à DevFest Lille.

On lance aujourd'hui nos **workshops** : des remixes du meilleur de nos formations sur une seule journée, 100% en ligne et hyper vivants, interactifs et fun, à des **prix extrêmement réduits**, avec jusqu'à 40 personnes.

Ça commence le **13 juillet** prochain, avec notre workshop **JS Masterclass** : les parties les plus utiles de notre formation ES Total, à **249 € TTC** seulement (la formation, sur 3 jours, coûte 1 500 € HT).

Et **jusqu'au 10 juin**, tarif de lancement à **199 € TTC** ! 😊

Tous les détails sont sur [bit.ly/js-masterclass](https://bit.ly/js-masterclass).

Réserve ta place dès maintenant, ça va être une énorme tuerie !



# Merci !



Laisse tes impressions ici ! Ça ne prend qu'un instant.

Cette présentation est sur [bit.ly/async-js-no-cringe](https://bit.ly/async-js-no-cringe).

[@porteneuve](https://twitter.com/porteneuve) / [@DelicioInsights](https://twitter.com/DelicioInsights) / [YouTube](https://www.youtube.com/@DelicioInsights)

Crédits : photo de couverture par [JESHOUTS.COM](https://JESHOUTS.COM) sur Unsplash