AJAX

Matthieu Nicolas
Licence Pro CIASIE

Plan

- Requêtes Asynchrones
- JSON
- L’API `Fetch`
- Le type `Promise`

Requêtes asynchrones

AJAX

Requêtes synchrones

- Le navigateur (client) interagit avec un serveur par le biais de requêtes
- Requêtes synchrones
  - Accès à une URL
  - Via un formulaire

Limites

- L’utilisateur “perd” la main
  - Page devient blanche en attendant de recevoir la réponse du serveur et que le navigateur l’affiche
- Retransmet et ré-affiche la page entière
  - Pour potentiellement mettre à jour qu’un champ

Expérience utilisateur

- Autrefois, les connexions étaient lentes…
  - Interruptions représentent un problème d’UX
- … en fait, c’est toujours le cas
  - Régions sinistrées d’Internet
  - Connexions mobiles instables

But des requêtes asynchrones

- Permettre d’échanger des informations entre le client et le serveur en “tâche de fond”…
- …sans interrompre l’utilisateur dans son activité

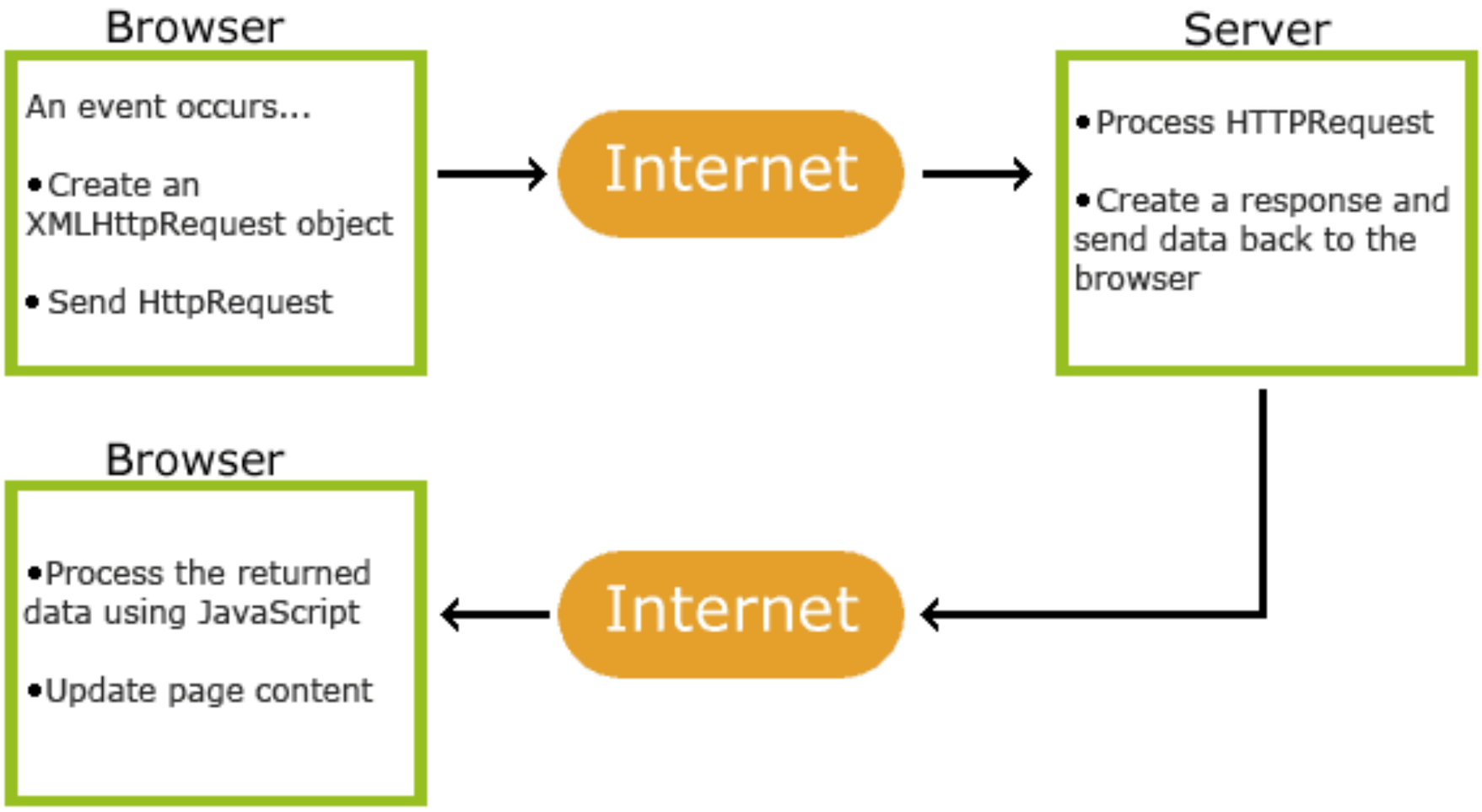
Utilisations possibles

- Récupération de nouvelles ressources
  - Résultats de recherche en temps réel
  - Scrolling infini
- Création de ressources
  - Ajout de commentaires
  - Like / Retweet

AJAX

- **A**synchronous **J**avaScript **A**nd **X**ML (AJAX)
- Ensemble de techniques pour effectuer des requêtes asynchrones dans le navigateur
- Permet de faire transiter des données au format XML
    - Mais aussi du simple texte, du JSON ou même du binaire…

Comment ça marche?



https://www.w3schools.com/xml/ajax_intro.asp

JSON

AJAX

Format JSON

- **J**ava**S**cript **O**bject **N**otation (JSON)
- Permet de représenter des objets sous forme textuelle, en reprenant la syntaxe JS
- Format généralement utilisé pour communiquer entre clients et serveurs

`JSON.stringify`

- `JSON.stringify(object)`
  - Retourne une `string` correspondante à `object` au format JSON
  - Utilise la méthode `toJSON()` de l'objet pour obtenir le résultat

```
» JSON.stringify({ id: 0, title: "test", content: "Hello" })
← "{\"id\":0,\"title\":\"test\",\"content\":\"Hello\"}"
```

JSON.parse

- `JSON.parse(json)`
  - Retourne `object` à partir de la `string` au format JSON correspondante

```
>> JSON.parse("{\"id\":0,\"title\":\"test\",\"content\":\"Hello\"}")
<- ▶ Object { id: 0, title: "test", content: "Hello" }
```

L'API `Fetch`

AJAX

Fetch

- Nouvelle API permettant d'effectuer des requêtes asynchrones
- Remplace
  - `XMLHttpRequest`
  - `jQuery.ajax()`

Utilisation

- Une fonction principale: `fetch(url, data)`

```
fetch("/article", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({ id: 0, title: "test", content: "Hello" })
})
```

`data`

- Un objet pouvant posséder plusieurs propriétés
  - `method`: verbe HTTP (GET, POST, PUT…)
  - `headers`: elle-même un objet
    - `Content-Type`: le type du contenu de la requête (`application/json`…)
  - `body`: le contenu de la requête
- Voir https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch

Type de retour? - 1

- `fetch()` permet de déclencher des requêtes asynchrones

- Cool de faire une requête…
- … mais récupérer le résultat, c’est mieux

- Mais qu’est-ce que `fetch()` peut retourner?

Type de retour? - 2

- Comme il s'agit d'une requête, ne possède pas le résultat instantanément
  - Latence réseau
  - Temps de calcul côté serveur

- `fetch()` ne peut donc pas retourner le résultat
- À la place, retourne une `Promise`

Le type `Promise` AJAX

Promise

- Type de données adopté récemment par les langages de programmation
- Permet de représenter une tâche asynchrone, son résultat ou son échec
- “Je te donnerais le résultat quand je l’aurai”

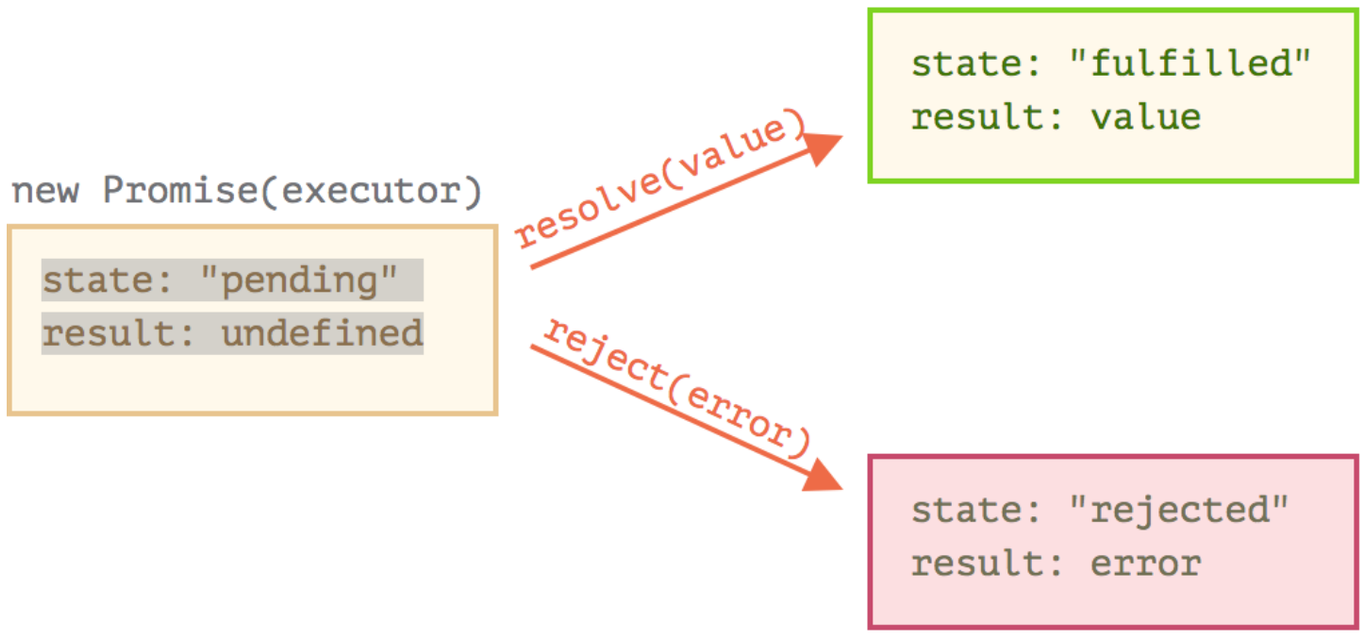
Construction

- Prend une fonction en paramètre de son `constructor`
- Cette fonction possède deux fonctions en paramètres, `resolve` et `reject`

```
new Promise(function (resolve, reject) {
    // Do stuff
})
```

État initial

- Initialement, est dans l'état `pending`
- `resolve` et `reject` vont permettre de compléter la `Promise`



https://javascript.info/promise-basics

`resolve`

- La fonction `resolve` indique le succès de la tâche asynchrone, et permet de retourner le résultat obtenu

```
new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve(42)
    }, 1000)
})
```

`reject`

- La fonction `reject` indique l'échec de la tâche asynchrone, et permet d'en spécifier la raison

```
new Promise(function (resolve, reject) {
    setTimeout(function () {
        reject("Failure reason")
    }, 1000)
})
```

Traitement du résultat

- Comme le code est asynchrone, doit fournir au navigateur le traitement à effectuer une fois le résultat obtenu
  - Utilisation de `callbacks`
  - Similaire aux `EventListeners`
- Deux méthodes permettent d'assigner ces `callbacks`, `then()` et `catch()`

`then()`

- Permet d'assigner la `callback` à exécuter lorsque la `Promise` réussit
- `callback` prend le résultat en paramètre

```
const promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve(42)
    }, 1000)
})

promise.then(function (res) {
    console.log(res)
})
```

`catch()`

- Permet d'assigner la `callback` à exécuter lorsque la `Promise` échoue
- `callback` prend le message d'erreur en paramètre

```
const promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        reject("Failure reason")
    }, 1000)
})

promise.catch(function (err) {
    console.error(err)
})
```

Combinaison

- Possible de spécifier le traitement en cas de succès et d'erreur sur une même `Promise`

```
const promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        if (Math.random() > 0.5) {
            resolve("Success!")
        } else {
            reject("No luck this time ... ")
        }
    }, 1000)
})

promise.then(function (msg) {
    console.log(msg)
}).catch(function (err) {
    console.error(err)
})
```

L'API `Fetch` - suite AJAX

Type de retour

- `fetch()` retourne une `Promise`

```
» fetch("/article", {
    method: "POST",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify({ id: 0, title: "test", content: "Hello" })
  })
← ▶ Promise { <state>: "pending" }
```

Échec de `fetch()`

- `fetch()` échoue si n'arrive pas à effectuer la requête
  - Pas de co ou serveur down

```
» const promise = fetch("/article", {
    method: "POST",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify({ id: 0, title: "test", content: "Hello" })
  })
  promise.catch(function (err) {
    console.error(err)
  })
← ▶ Promise { <state>: "pending" }
✖ ▶ TypeError: "NetworkError when attempting to fetch resource."
```

Succès de `fetch()`

- `fetch()` réussit dans tous les autres cas
  - Lorsqu'on obtient une réponse du serveur
    - Même si la réponse est négative (code HTTP 4xx/5xx)
- Fournit un objet `Response` à la callback appelée en cas de succès

L'objet `Response` - 1

- Possède plusieurs propriétés
  - `status`: le code HTTP de la réponse
  - `ok`: indique si le status est 2xx ou non

```
» promise.then(function (response) {
    console.log("response.status: ", response.status)
    console.log("response.ok: ", response.ok)
  })
  response.status:  200
  response.ok:  true
```

Gestion des “succès”

- Lever une exception dès que `ok === false`

```
» promise.then(function (response) {
      if (!response.ok) {
          throw new Error(response.statusText)
      }
  }).catch(function (err) {
    console.error(err)
  })
← ▶ Promise { <state>: "pending" }
✖ ▶ Error: "Bad Request"
```

L'objet `Response` - 2

- Possède plusieurs méthodes pour accéder au `body` de la réponse
- Ces méthodes retournent elles-même des `Promises`
- Le format sous lequel le contenu est récupéré dépend de la méthode utilisé
- Voir https://developer.mozilla.org/en-US/docs/Web/API/Response

L'objet `Response` - 3

- Exemple
  - `json()`: récupère le contenu sous forme d'un objet JS

```
» const promise = fetch("/article", { method: "GET" })
  promise.then(function (response) {
    response.json().then(function (data) {
      console.log(data)
    })
  })
← ▶ Promise { <state>: "pending" }
  ▼ (1) […]
    ▶ 0: Object { id: 0, title: "test", content: "Hello" }
      length: 1
    ▶ <prototype>: Array []
```

Chaînage de `Promise`

- Lourd d'imbriquer des `Promises` dans des `Promises` (*promise/callback/cascade hell*)
- Peut retourner une `Promise` depuis la `callback` d'une `Promise` pour les chaîner

```
>> const promise = fetch("/article", { method: "GET" })
   promise.then(function (response) {
     return response.json()
   }).then(function (data) {
     console.log(data)
   })
<- ▶ Promise { <state>: "pending" }

   ▼ (1) […]
     ▶ 0: Object { id: 0, title: "test", content: "Hello" }
       length: 1
     ▶ <prototype>: Array []
```

TD

- Reprend la TODO-list
- Utilise maintenant un serveur pour conserver la liste
- Va utiliser `fetch()` pour interagir avec
  - récupérer les todos existant
  - en créer de nouveaux
  - en supprimer
- Lien dispo sur Arche