Utiliser Fetch

L'API Fetch fournit une interface JavaScript pour l'accès et la manipulation des parties de la pipeline HTTP, comme les requêtes et les réponses. Cela fournit aussi une méthode globale `fetch()` qui procure un moyen facile et logique de récupérer des ressources à travers le réseau de manière asynchrone.

Ce genre de fonctionnalité était auparavant réalisé avec `XMLHttpRequest`. Fetch fournit une meilleure alternative qui peut être utilisée facilement par d'autres technologies comme `Service Workers`. Fetch fournit aussi un endroit unique et logique pour la définition d'autres concepts liés à HTTP comme CORS et les extensions d'HTTP.

L'état actuel du support par les navigateurs

Le support de Fetch est à ses débuts, mais les choses progressent. Il a été activé par défaut sur Firefox 39 et supérieur, et sur Chrome 42 et supérieur.

Si vous souhaitez l'utiliser maintenant, il y a un polyfill Fetch disponible qui recrée la fonctionnalité pour les navigateurs qui ne le supportent pas. Gardez à l'esprit qu'il est au stade expérimental et pas encore complètement fonctionnel.

> **Note** : Certaines préoccupations ont été soulevées sur le fait que la spécification de Fetch est en contradition avec la spécification de Streams; cependant, les prévisions montrent une intention d'intégrer Streams avec Fetch: pour plus d'informations reportez vous à Fetch API integrated with Streams.

Détection de la fonctionnalité

Le support de l'API Fetch peut être détecté en vérifiant l'existence de `Headers`, `Request`, `Response` ou `fetch()` sur la portée de `Window` ou de `Worker`. Par exemple, vous pouvez faire cela dans votre script :

```
if (window.fetch) {
    // exécuter ma requête fetch ici
} else {
```

```
} else {
    // Faire quelque chose avec XMLHttpRequest?
}
```

Créer une requête fetch

Une requête fetch basique est vraiment simple à initier. Jetez un coup d'œil au code suivant :

```
const myImage = document.querySelector('img');

fetch('flowers.jpg')
.then(function(response) {
  return response.blob();
})
.then(function(myBlob) {
  const objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
});
```

Ici nous récupérons une image à travers le réseau et l'insérons dans un élément `<img>`. L'utilisation la plus simple de `fetch()` prend un argument — le chemin de la ressource que nous souhaitons récupérer — et retourne une promesse (promise) contenant, en réponse, un objet (de type `Response`).

Bien sûr, il s'agit seulement d'une réponse HTTP, pas exactement de l'image. Pour extraire le contenu de l'image de la réponse, nous utilisons la méthode `blob()` (définie sur le mixin `Body`, qui est implémenté autant par les objets `Request` que par les objets `Response`).

> **Note** : Le mixin Body a aussi des méthodes similaires pour extraire d'autres types contenu ; pour plus d'informations regardez la section Corps.

Un objet `objectURL` est ensuite créé à partir du `Blob` extrait, puis est inseré dans `img`.

Les requêtes Fetch sont controllées par la directive `connect-src` du Content Security Policy plutôt que par la directive de la ressource dont il s'agit de la récupération.

Fournir des options à la requête

La méthode `fetch()` accepte un second paramètre, optionnel ; un objet `init` qui vous permet de contrôler un certain nombre de réglages :

```
var myHeaders = new Headers();

var myInit = { method: 'GET',
               headers: myHeaders,
               mode: 'cors',
               cache: 'default' };


fetch('flowers.jpg',myInit)
.then(function(response) {
  return response.blob();
})
.then(function(myBlob) {
  var objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
});
```

Reportez-vous à `fetch()` pour la liste complète des options disponibles, et plus de détails.

Vérifier que la récupération a réussi

Une promesse `fetch()` va retourner une `TypeError` quand un problème réseau s'est produit. Cependant, il peut aussi s'agir d'un problème de permission ou quelque chose de similaire — un code HTTP 404 ne constitue pas une erreur réseau, par exemple. Un bon test de la réussite de `fetch()` devrait inclure la vérification que la promesse soit résolue, puis vérifier que la propriété `Response.ok` ait la valeur *true*. Le code devrait ressembler à ce qui suit:

```
fetch('flowers.jpg').then(function(response) {
  if(response.ok) {
    response.blob().then(function(myBlob) {
      var objectURL = URL.createObjectURL(myBlob);
      myImage.src = objectURL;
    });
  } else {
    console.log('Mauvaise réponse du réseau');
  }
})
.catch(function(error) {
  console.log('Il y a eu un problème avec l\'opération fetch: ' + error.mess
});
```

Fournir votre propre objet requête

Plutôt que de transmettre le chemin de la ressource que vous souhaitez récupérer avec l'appel `fetch()`, vous pouvez créer un objet de requête en utilisant le constructeur `Request()`, et le transmettre à la méthode `fetch()` en tant qu'argument:

```
var myHeaders = new Headers();

var myInit = { method: 'GET',
               headers: myHeaders,
               mode: 'cors',
               cache: 'default' };

var myRequest = new Request('flowers.jpg',myInit);

fetch(myRequest,myInit)
.then(function(response) {
  return response.blob();
})
.then(function(myBlob) {
  var objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
});
```

`Request()` accepte exactement les mêmes paramètres que la méthode `fetch()`. Vous pouvez même lui transmettre un objet Request existant pour en créer une copie :

```
var anotherRequest = new Request(myRequest,myInit);
```

C'est très pratique, si le corps de la requête et de la réponse ne sont utilisés qu'une fois seulement. Cette manière de faire une copie permet de ré-utiliser la requête/réponse, en changeant juste les options du `init` si nécessaire.

> **Note** : Il y a aussi une méthode `clone()` qui créer une copie. Cela a une sémantique légèrement différente à l'autre méthode de copie— La première va échouer si l'ancien corps de la requête a déjà été lu (même pour copier une réponse), alors qu'avec `clone()` non.

En-têtes (Headers)

L'interface `Headers` vous permet de créer vos propres objets d'en-têtes via le constructeur `Headers()`. Un objet en-tête est un simple ensemble de plusieurs clé-valeurs:

```
var content = "Hello World";
var myHeaders = new Headers();
myHeaders.append("Content-Type", "text/plain");
myHeaders.append("Content-Length", content.length.toString());
myHeaders.append("X-Custom-Header", "ProcessThisImmediately");
```

On peut atteindre le même résultat en transmettant un tableau de tableaux ou un objet littéral au constructeur:

```
myHeaders = new Headers({
  "Content-Type": "text/plain",
  "Content-Length": content.length.toString(),
  "X-Custom-Header": "ProcessThisImmediately",
});
```

Le contenu peut être interrogé et récupéré:

```
console.log(myHeaders.has("Content-Type")); // true
console.log(myHeaders.has("Set-Cookie")); // false
myHeaders.set("Content-Type", "text/html");
myHeaders.append("X-Custom-Header", "AnotherValue");

console.log(myHeaders.get("Content-Length")); // 11
console.log(myHeaders.getAll("X-Custom-Header")); // ["ProcessThisImmediatel

myHeaders.delete("X-Custom-Header");
console.log(myHeaders.getAll("X-Custom-Header")); // [ ]
```

Certaines de ces opérations sont seulement utiles dans `ServiceWorkers`, mais elles fournissent une bien meilleur API pour la manipulation des en-têtes.

Toutes les méthodes d'en-tête provoquent une erreur TypeError si un nom d'en-tête utilisé n'est pas un nom d'en-tête HTTP valide. Les opérations de mutation vont provoquer une erreur TypeError si il y a une protection immutable (voir ci-dessous). Sinon elles vont échouer en silence. Par exemple :

```
var myResponse = Response.error();
try {
  myResponse.headers.set("Origin", "http://mybank.com");
} catch(e) {
  console.log("Ne peut pas prétendre être une banque!");
}
```

Un bon cas d'utilisation des en-têtes est de vérifier que le type de contenu récupéré est correct avant de poursuivre le traitement. Par exemple:

```
fetch(myRequest).then(function(response) {
  var contentType = response.headers.get("content-type");
  if(contentType && contentType.indexOf("application/json") !== -1) {
    return response.json().then(function(json) {
      // traitement du JSON
    });
  } else {
    console.log("Oops, nous n'avons pas du JSON!");
  }
});
```

Protection (Guard)

Puisque les en-têtes peuvent être envoyés dans les requêtes et reçus dans les réponses, et ont diverses limitations sur quelles informations peuvent et doivent être mutables, les objets en-tête ont une propriété *guard*. Ce n'est pas exposé au Web, mais cela définit quelle opération de mutation est autorisée sur l'objet en-tête.

Les valeurs possibles de la propriété *guard* sont:

- `none` : défaut.
- `request` : guard pour l'en-tête obtenu d'une requête (`Request.headers`).
- `request-no-cors` : guard pour l'en-tête obtenu d'une requête créé avec `Request.mode` `no-cors` .
- `response` : guard pour l'en-tête obtenu d'une réponse (`Response.headers`).
- `immutable` : majoritairement utilisé pour les ServiceWorkers; retourne un objet en-tête en lecture seule.

> **Note**: Vous ne pouvez pas ajouter ou définir sur une requête protegée une en-tête `Content-Length` . De manière similaire, ajouter `Set-Cookie` dans l'en-tête de réponse n'est pas autorisé: les ServiceWorkers ne sont pas autorisés à gérer des cookies via des réponses synthétisées.

Réponses

Comme vous l'avez vu ci-dessus, des instances de `Response` sont retournées quand la

promesse de `fetch()` est résolue.

Elles peuvent aussi être programmées dans le code via JavaScript, mais c'est seulement utile concernant les `ServiceWorkers`, quand vous retournez, pour une requête reçue, une réponse personnalisée en utilisant la méthode `respondWith()` :

```
var myBody = new Blob();

addEventListener('fetch', function(event) {
  event.respondWith(new Response(myBody, {
    headers: { "Content-Type" : "text/plain" }
  });
)});
```

Le constructeur `Response()` prend deux arguments optionnels —le corps de la réponse, et un objet d'options (similaire à l'objet que `Request()` accepte).

Les propriétés de réponse les plus communes que vous allez utiliser sont:

- `Response.status` —Un entier (valeur par défaut 200) contenant le code de statut de la réponse.
- `Response.statusText` — Une chaine de caractères (valeur par défaut "OK"), qui correspond au message du statut HTTP.
- `Response.ok` —vu précedemment, c'est un raccourci pour vérifier que le code de statut et bien compris entre 200 et 299 inclus. Retourne un `Boolean`.

> **Note**: La méthode statique `error()` retourne simplement une réponse d'erreur. De manière similaire, `redirect()` retourne une réponse de redirection vers une URL spécifique. Elles sont aussi pertinentes pour les Service Workers.

Corps

Autant une requête qu'une réponse peut contenir un corps avec des données. Un corps est une instance de n'importe lequel des types suivants:

- `ArrayBuffer`
- `ArrayBufferView` (Uint8Array et ses proches)
- `Blob` /Fichier
- chaine de caractères
- `URLSearchParams`

- `FormData`

Le mixin `Body` définit les méthodes suivantes pour extraire le corps (implémenté autant par la `Request` que par la `Response`). Elles retournent toutes une promesse qui sera éventuellement résolue avec le contenu actuel.

- `arrayBuffer()`
- `blob()`
- `json()`
- `text()`
- `formData()`

Ceci rend l'usage de données non textuelles plus facile qu'avec XHR.

Le corps des requêtes peut être défini en passant les paramètres du corps:

```
var form = new FormData(document.getElementById('login-form'));
fetch("/login", {
  method: "POST",
  body: form
})
```

Les Requêtes et Réponses (et par extension la fonction `fetch()`), vont tenter de déterminer le type de contenu. Une requête va automatiquement définir un en-tête `Content-Type` si rien n'est défini dans le dictionnaire [NDLT: configuration d'initialisation].

Spécifications

Spécification	Statut	Commentaire
Fetch	Standard évolutif	Définition initiale

Compatibilité navigateur

Report problems with this compatibility data on GitHub

`fetch`	
Chrome	42

Edge	14
Firefox	39
Internet Explorer	**No**
Opera	29
Safari	10.1
WebView Android	42
Chrome Android	42
Firefox Android	39
Opera Android	29
iOS Safari	10.3
Samsung Internet	4.0

Support for blob: and data:

Chrome	48
Edge	79
Firefox	?
Internet Explorer	**No**
Opera	?
Safari	?
WebView Android	43
Chrome Android	48
Firefox Android	?
Opera Android	?
iOS Safari	?
Samsung Internet	5.0

referrerPolicy

Chrome	52
Edge	79

Firefox	52
Internet Explorer	**No**
Opera	39

Safari	11.1
WebView Android	52
Chrome Android	52
Firefox Android	52
Opera Android	41
iOS Safari	**No**
Samsung Internet	6.0

signal

Chrome	66
Edge	16
Firefox	57
Internet Explorer	**No**
Opera	53
Safari	11.1
WebView Android	66
Chrome Android	66
Firefox Android	57
Opera Android	47
iOS Safari	11.3
Samsung Internet	9.0

Streaming response body

Chrome	43
Edge	14

Edge	14
Firefox	Yes
Internet Explorer	**No**
Opera	29
Safari	10.1
WebView Android	43
Chrome Android	43
Firefox Android	**No**
Opera Android	**No**
iOS Safari	10.3
Samsung Internet	4.0

☐ Full support

☐ No support

☐ Compatibility unknown

Experimental. Expect behavior to change in the future.

See implementation notes.

User must explicitly enable this feature.

Voir aussi

- API ServiceWorker

- API ServiceWorker
- HTTP access control (CORS)
- HTTP
- Polyfill pour Fetch
- Exemples de Fetch sur Github

Last modified: 16 oct. 2020, by MDN contributors