Services Web

S4 - R4.01 - 2022/2023

Plan de cette séance

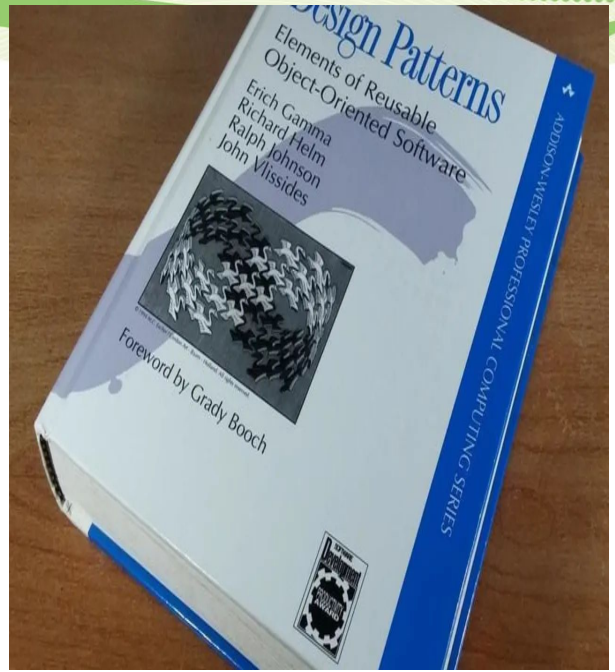
- Gestion des événements
  - Le modèle de réacteur
  - Callback pattern
  - Observer pattern
  - EventEmitter
- Pipeline ETL
  - Extract
  - Transform
  - Load
- Web scraping
- Demo

Design Pattern et Anti-pattern

Quelle est la différence entre un design pattern (patron de conception) et un antipattern ?

Design Pattern

- Un patron de conception est une solution réutilisable à un problème récurrent. Le terme est très large dans sa définition et peut couvrir plusieurs domaines d'une application.
- Le terme est souvent associé à un ensemble bien connu de modèles orientés objet qui ont été popularisés dans les années 90 par le livre **Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education**.
- L'application de patrons de conception orientés objet en JavaScript n'est pas aussi linéaire et formelle qu'elle le serait dans un langage orienté objet classique.



JavaScript est basé sur des prototypes

- JavaScript est un langage basé sur des prototypes, ce qui signifie qu'il utilise **l'héritage de prototype** pour implémenter la programmation orientée objet. Dans un langage basé sur des prototypes, il n'y a pas de classes, mais les **objets peuvent hériter des propriétés et des méthodes d'autres objets.**
- En JavaScript, chaque objet a une propriété spéciale appelée **prototype**, qui est **une référence à un autre objet**. Lorsque vous essayez d'accéder à une propriété ou à une méthode d'un objet, JavaScript vérifie d'abord si l'objet possède cette propriété ou cette méthode. Si ce n'est pas le cas, JavaScript cherchera la propriété ou la méthode dans le prototype de l'objet, et si ce n'est pas là, il cherchera dans le prototype du prototype, et ainsi de suite. Cette chaîne de prototypes est appelée “**prototype chain**”.

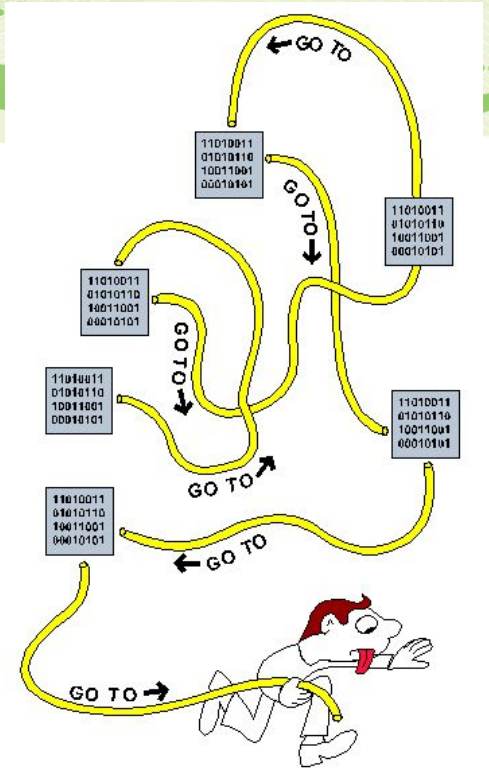
JavaScript est basé sur des prototypes

- Un prototype est un objet qui sert de modèle (template) à d'autres objets. Lorsqu'un objet est créé, il hérite des propriétés et des méthodes de son prototype.
- Vous pouvez considérer les prototypes comme un moyen de partager des propriétés et des méthodes entre plusieurs objets sans avoir à les définir plusieurs fois.

```
// Définir l'objet de base
let baseObject = {
  name: 'base object',
  getName: function() {
    return this.name;
  }
};

// // Créer un nouvel objet qui hérite des propriétés et des méthodes de l'objet de base
let derivedObject1 = Object.create(baseObject);

// Ajouter une nouvelle propriété à l'objet dérivé
derivedObject1.age = 30;

console.log(derivedObject1.getName()); // Output: 'base object'
console.log(derivedObject1.age); // Output: 30

// Créer un autre objet qui hérite de l'objet dérivé
let derivedObject2 = Object.create(derivedObject1);

// Ajouter une nouvelle propriété au deuxième objet dérivé
derivedObject2.gender = 'male';

console.log(derivedObject2.getName()); // Output: 'base object'
console.log(derivedObject2.age); // Output: 30
console.log(derivedObject2.gender); // Output: 'male'
```

Anti-pattern

- Un antipattern est une solution courante mais inefficace à un problème de conception de logiciel. C'est un modèle de comportement ou d'action qui est souvent utilisé mais qui est généralement considéré comme une mauvaise pratique et qui conduit à un résultat négatif.
- Les antimodèles/antipatterns peuvent se produire dans de nombreux domaines du développement logiciel, tels que l'architecture, le codage, les tests et la gestion de projet.



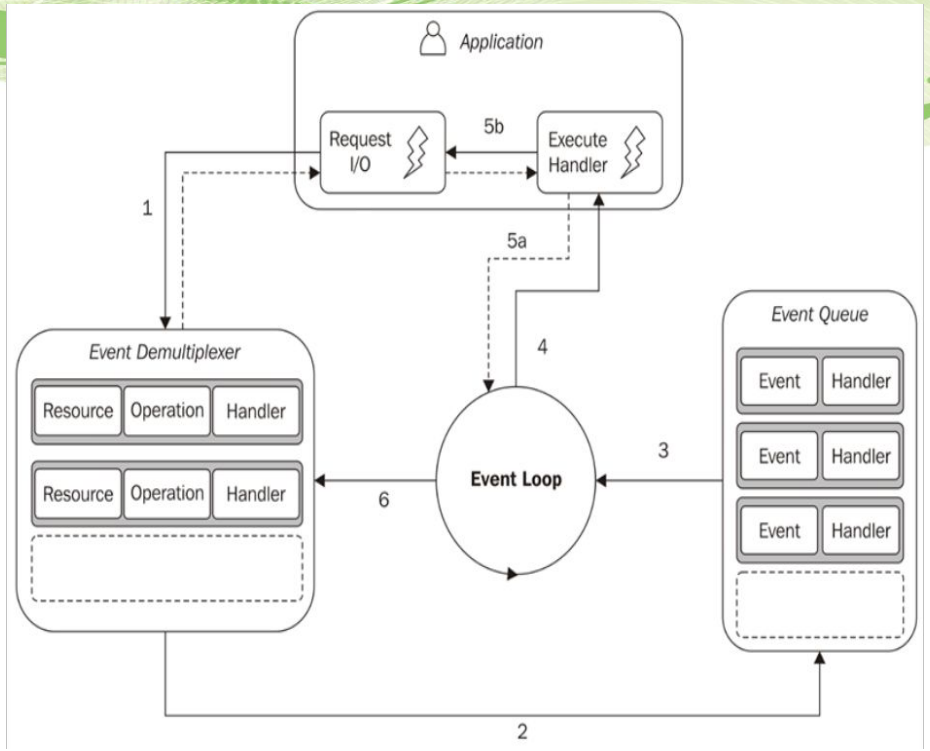
Anti-pattern

- **Code spaghetti** : une base de code difficile à comprendre, à maintenir et à modifier en raison de son manque de structure et d'organisation.
- **Objet divin** : une classe ou un module qui a trop de responsabilités et qui est étroitement couplé à d'autres parties du système.
- **Optimisation prématurée** : action d'optimiser le code avant qu'il ne soit nécessaire, ce qui peut entraîner une complexité inutile et des performances médiocres.
- **Marteau d'or** : La tendance à utiliser un outil, un langage ou un framework particulier pour chaque problème, même lorsqu'il n'est pas le mieux adapté.
- **Programmation copier-coller** : La pratique consistant à copier et coller du code au lieu de le réutiliser, ce qui entraîne des problèmes de duplication de code et de maintenance.

Il est important de noter qu'un anti-modèle n'est pas toujours mauvais, parfois il est nécessaire pour s'adapter au contexte et cela pourrait être la meilleure solution. Il est important d'être conscient des anti-modèles possibles et de comprendre le contexte et les compromis avant d'en utiliser un.

Reactor Pattern

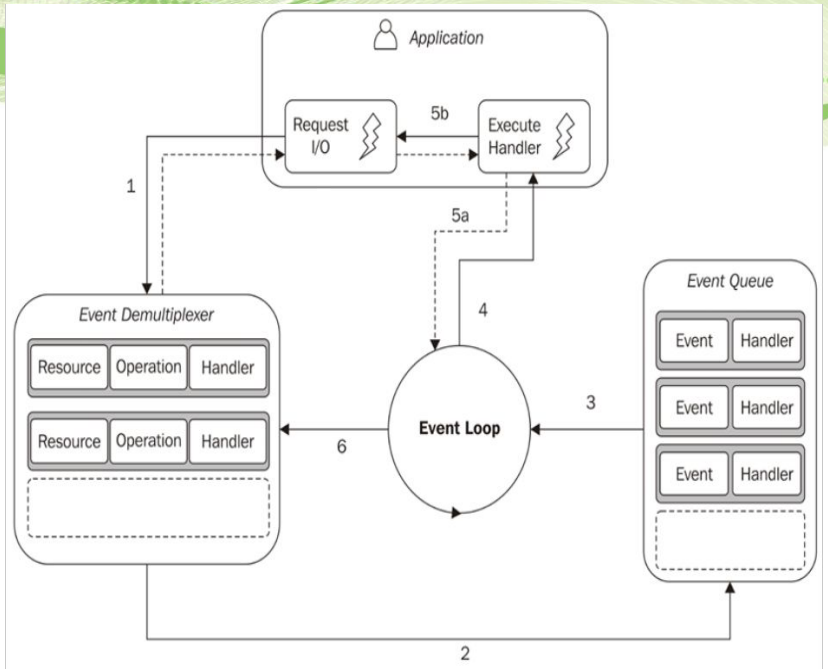
- L'idée principale derrière le modèle de réacteur est d'avoir un gestionnaire associé à chaque opération d'E/S. Un gestionnaire dans Node.js est représenté par une fonction de rappel.
- Le gestionnaire sera invoqué dès qu'un événement est produit et traité par la boucle d'événements.



Référence : Node.js Design Patterns - Troisième édition

Reactor Pattern

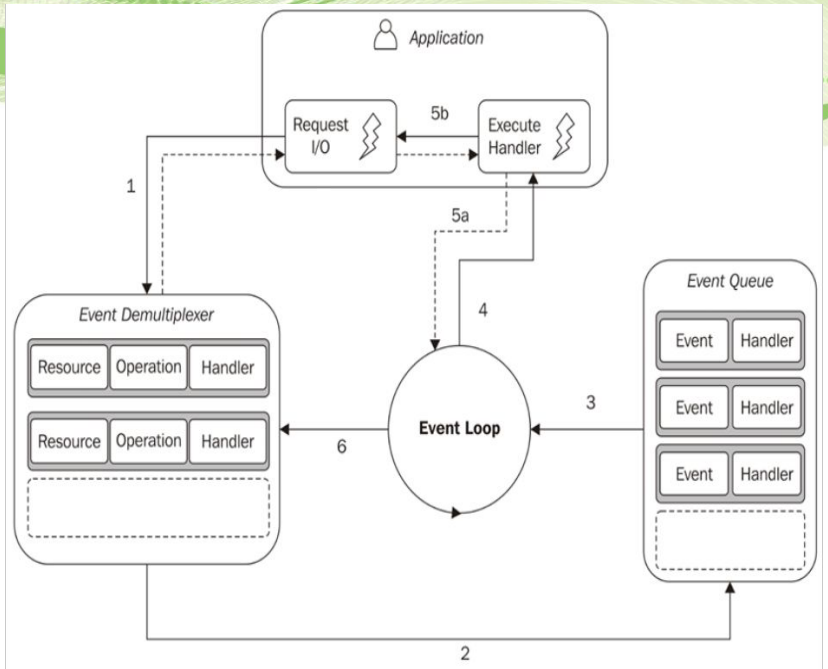
- 1) L'application génère une nouvelle opération d'E/S en soumettant une demande au démultiplexeur d'événements. L'application spécifie également un gestionnaire, qui sera appelé une fois l'opération terminée. La soumission d'une nouvelle demande au démultiplexeur d'événements est un appel non bloquant et il rend immédiatement le contrôle à l'application.
- 2) Lorsqu'un ensemble d'opérations d'E/S est terminé, le démultiplexeur d'événements pousse un ensemble d'événements correspondants dans la file d'attente d'événements.
- 3) À ce stade, la boucle d'événements parcourt les éléments de la file d'attente d'événements.



Référence : Node.js Design Patterns - Troisième édition

Reactor Pattern

- 4) Pour chaque événement, le gestionnaire associé est appelé.
- Le gestionnaire, qui fait partie du code de l'application, redonne le contrôle à la boucle d'événements lorsque son exécution est terminée (5a). Pendant que le gestionnaire s'exécute, il peut demander de nouvelles opérations asynchrones (5b), provoquant l'ajout de nouveaux éléments au démultiplexeur d'événements (1).
- 6) Lorsque tous les éléments de la file d'attente d'événements sont traités, la boucle d'événements se bloque à nouveau sur le démultiplexeur d'événements, qui déclenche un autre cycle lorsqu'un nouvel événement est disponible.



Référence : Node.js Design Patterns - Troisième édition

Reactor Pattern

- Le comportement asynchrone est maintenant devenu clair. L'application exprime son intérêt à accéder à une ressource à un moment donné (sans blocage) et fournit un gestionnaire, qui sera ensuite appelé à un autre moment une fois l'opération terminée.
- Une application Node.js se fermera lorsqu'il n'y aura plus d'opérations en attente dans le démultiplexeur d'événements et qu'il n'y aura plus d'événements à traiter dans la file d'attente d'événements.
- Le modèle de réacteur gère les E/S en bloquant (attend ou arrête l'exécution du programme) jusqu'à ce que de nouveaux événements soient disponibles à partir d'un ensemble de ressources observées, puis réagit en envoyant chaque événement à un gestionnaire associé.

Remplissez le vide

En programmation asynchrone, certaines opérations, comme la lecture d'un fichier ou l'exécution d'une requête réseau, sont lancées puis exécutées "en arrière-plan". Lorsque nous invoquons une opération asynchrone, l'instruction qui suit est exécutée immédiatement, même si l'opération asynchrone précédente n'est pas encore terminée. Dans ce scénario, nous avons besoin d'un moyen d'être averti lorsqu'une opération asynchrone se termine, puis de continuer le flux d'exécution en utilisant les résultats de l'opération. Le mécanisme le plus basique pour être averti de l'achèvement d'une opération asynchrone dans Node.js est le _____ **????**

Remplissez le vide

Le rappel (CALLBACK) est le bloc de construction le plus basique sur lequel reposent tous les autres mécanismes asynchrones. En fait, sans rappels, nous n'aurions pas de _____, et donc même pas _____ ; nous n'aurions pas non plus de flux (STREAMS) ou _____. C'est pourquoi il est important de savoir comment fonctionnent les rappels.

Callback Pattern

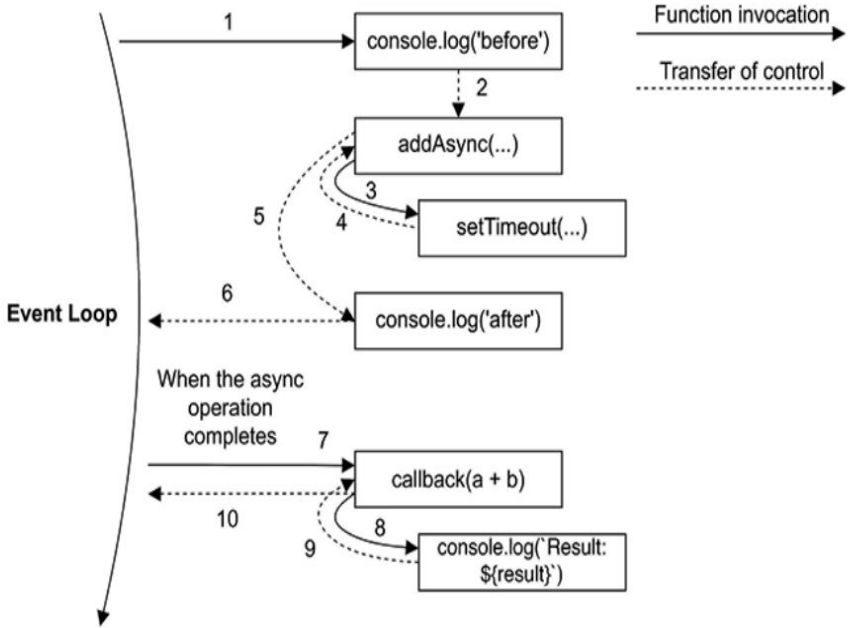
- Les rappels sont la matérialisation des gestionnaires du modèle Reactor. Ils font partie de ces empreintes qui donnent à Node.js son style de programmation distinctif.
- Les rappels sont des fonctions qui sont invoquées pour propager le résultat d'une opération, et c'est exactement ce dont nous avons besoin lorsqu'il s'agit d'opérations asynchrones.
- Dans le monde asynchrone, ils remplacent l'utilisation de l'instruction de retour (**RETURN**), qui, à son tour, s'exécute toujours de manière synchrone.
- JavaScript est le langage idéal pour les rappels, car les fonctions sont des **objets de première classe** et peuvent être facilement affectées à des variables, transmises en tant qu'arguments, renvoyées à partir d'un autre appel de fonction ou stockées dans des structures de données.

Les fonctions en JS : objets de première classe

```
// vous pouvez affecter une fonction à une variable
let myFunction = function() { console.log("Hello, World!") }

// vous pouvez passer une fonction comme argument à une autre fonction
let sayHello = function(greeting) {
  console.log(greeting);
}

let callFunction = function(fn) {
  fn("Hello, World!");
}

callFunction(sayHello);

// vous pouvez également renvoyer une fonction à partir d'une fonction
function createFunction() {
  return function() { console.log("Hello, World!"); }
}

myFunction = createFunction();
myFunction();

// vous pouvez stocker des fonctions dans des tableaux
let functions = [function() { console.log("Hello, World!"); },
                 function() { console.log("Goodbye, World!"); }];

functions[0]();
functions[1]();
```

Callback Pattern

```
function additionAsync (a, b, callback) {
  setTimeout(() => callback(a + b), 100)
}

console.log('before')
additionAsync(1, 2, result => console.log(`Result: ${result}`))
console.log('after')
```



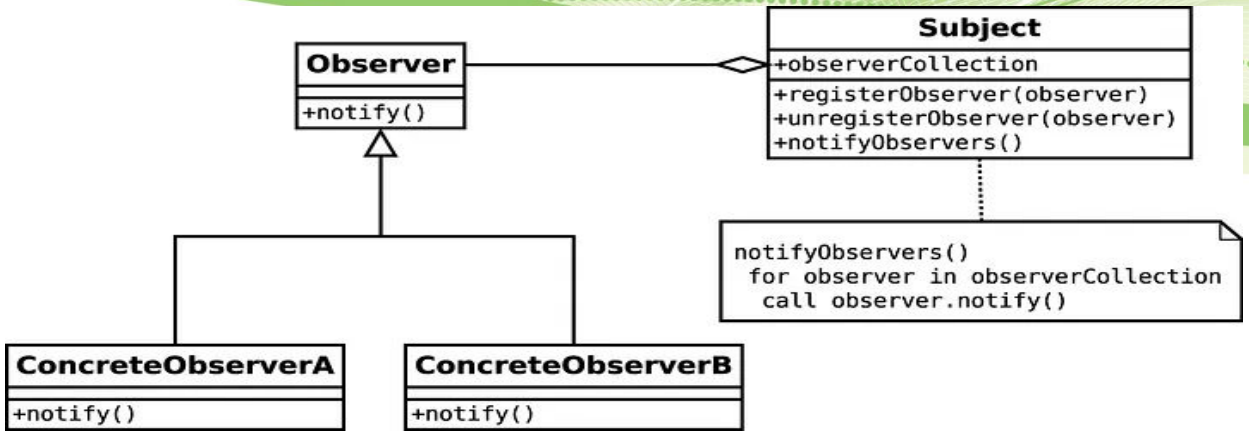
Référence : Node.js Design Patterns - Troisième édition

Observer Pattern

- Un autre modèle important et fondamental utilisé dans Node.js est le modèle Observer. Avec le modèle Reactor et les rappels, le modèle Observer est une exigence absolue pour maîtriser le monde asynchrone de Node.js.
- Le pattern Observer est la solution idéale pour modéliser la nature réactive de Node.js et un complément parfait pour les callbacks.

> Le pattern Observer définit un objet (appelé sujet) qui peut notifier un ensemble d'observateurs (ou listeners) lorsqu'un changement d'état se produit.

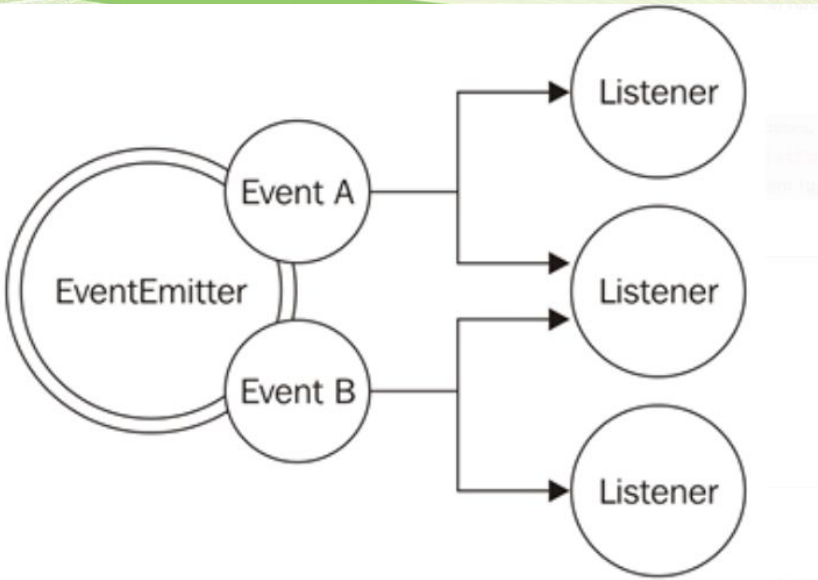
Observer Pattern



La principale différence avec le modèle de rappel est que le sujet peut en fait notifier plusieurs observateurs, tandis qu'un rappel traditionnel propagera généralement son résultat à un seul observateur, le rappel.

Emetteur d'événement (EventEmitter)

- Dans la programmation orientée objet traditionnelle, le modèle Observer nécessite des interfaces, des classes concrètes et une hiérarchie.
- Dans Node.js, tout cela devient beaucoup plus simple. Le modèle Observer est déjà intégré au noyau et est disponible via la classe **EventEmitter**.
- La classe **EventEmitter** nous permet d'enregistrer une ou plusieurs fonctions en tant qu'écouteurs, qui seront invoquées lorsqu'un type d'événement particulier est déclenché.

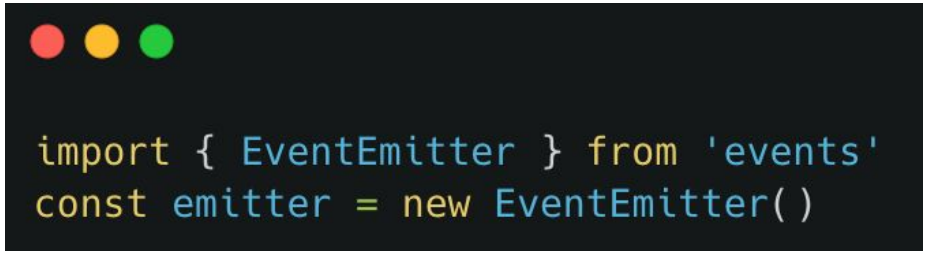


Référence : Node.js Design Patterns - Troisième édition

Emetteur d'événement (EventEmitter)

L'EventEmitter est exporté à partir du module de base des événements (“**events**”).

Le code suivant montre comment nous pouvons obtenir une référence à celui-ci :



- **on(event, listener)** : cette méthode nous permet d'enregistrer un nouvel écouteur (une fonction) pour le type d'événement donné.
- **once(event, listener)** : cette méthode enregistre un nouvel écouteur, qui est ensuite supprimé une fois l'événement émis pour la première fois.
- **emit(event, [arg1], [...])** : cette méthode produit un nouvel événement et fournit des arguments supplémentaires à transmettre aux écouteurs.
- **removeListener(event, listener)** : cette méthode supprime un écouteur pour le type d'événement spécifié.

EventEmitter

Chaque fois que vous devrez gérer des événements quelconques, vous ferez intervenir des objets ou des classes d'objets pouvant supporter la gestion des événements. Ces objets, pour avoir la capacité de recevoir de tels événements, doivent hériter des fonctionnalités de la classe **events.EventEmitter** défini dans Node.

EventEmitter

Node définit déjà en standard des objets ou des classes supportant la gestion des événements. Par exemple, la classe **http.Server** définie dans le module **http** de Node, et permettant de créer des serveurs HTTP, dérive de la classe **events.EventEmitter**, ce qui lui permet de recevoir des événements tels que les requêtes des utilisateurs.

EventEmitter

```
var http = require("http");

var server = http.createServer( requestListener: (req :... ,res :... )=>{
    res.setHeader("Content-Type", "text/html");
    res.write("<h2>Bonjour !!!!</h2>");
    res.end();
})

server.listen( port: 3000);
console.log("Le serveur ecoute sur le port 3000");
```

```
var http = require("http");
var events = require("events");

var server = http.createServer( requestListener: (req :... ,res :... )=>{
    res.setHeader("Content-Type", "text/html");
    res.write("<h2>Bonjour !!!!</h2>");
    res.end();
})

server.listen( port: 3000);
console.log("Le serveur ecoute sur le port 3000");
console.log(server instanceof events.EventEmitter);
```

```
Le serveur ecoute sur le port 3000
true
```

EventEmitter

Problème

Vous souhaitez utiliser une approche basée sur les événements pour résoudre un problème. Vous avez une classe que vous aimeriez faire fonctionner lorsque des événements asynchrones se produisent. Les interfaces utilisateur Web, Desktop et mobiles ont une chose en commun : elles sont basées sur les événements.

Les événements sont un excellent paradigme pour traiter quelque chose d'intrinsèquement asynchrone : l'apport (input) des êtres humains. Pour montrer comment fonctionne **EventEmitter**, nous utiliserons un **lecteur de musique** comme exemple. Il ne jouera pas vraiment de musique, mais le concept sous-jacent est un excellent moyen d'apprendre à utiliser les événements.

EventEmitter

```
const EventEmitter = require("events").EventEmitter;

var AudioDevice = {
    play: function(track) {
        console.log("Playing : " + track);
    },
    stop: function() {
        console.log("Stopped playing...")
    }
};

// Nous créons une classe événementielle personnalisée.
no usages
class MusicPlayer extends EventEmitter {
    no usages
    constructor() {
        super();
        this.playing = false;
    }
}
```

```
var musicPlayer = new MusicPlayer();

musicPlayer.on( eventName: 'play', listener: function(track) {
    this.playing = true;
    AudioDevice.play(track);
});
musicPlayer.on( eventName: 'stop', listener: function() {
    this.playing = false;
    AudioDevice.stop();
});

musicPlayer.emit( eventName: 'play', args: 'The Roots - The Fire');

setTimeout( callback: function() {
    musicPlayer.emit( eventName: 'stop');
}, ms: 1000);
```

```
Playing : The Roots - The Fire
Stopped playing...
```

Déclenchement d'un événement trop tôt

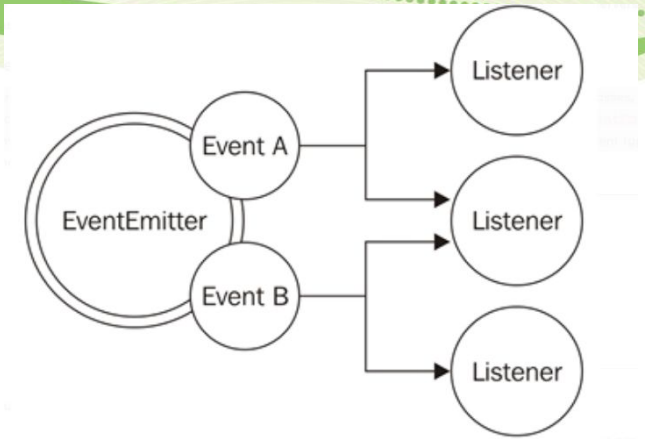
```
1  var events = require("events");
2
3  var obj1 = new events.EventEmitter();
4
5  obj1.emit( eventName: "event1");
6
7  obj1.addListener( eventName: "event1",  listener: function(){
8      console.log("L'objet obj1 a recu un événement event1");
9  });
```

Si l'événement était déclenché avant que la fonction de callback soit déclenchée, ce déclenchement ne serait pas pris en compte par la fonction de callback, car il serait déclenché trop tôt.

Utiliser plusieurs fois la méthode addListener()

```
var events = require("events");

var obj1 = new events.EventEmitter();

obj1.addListener( eventName: "event1", listener: function(){
    console.log("1- L'objet obj1 a recu un événement event1");
});
obj1.addListener( eventName: "event1", listener: function(){
    console.log("2- L'objet obj1 a recu un événement event1");
});

obj1.emit( eventName: "event1");
obj1.emit( eventName: "event1");
```

```
(base) josephazar@Josephs-MBP-2 webserv_events % node exemple_events3.js
1- L'objet obj1 a recu un événement event1
2- L'objet obj1 a recu un événement event1
1- L'objet obj1 a recu un événement event1
2- L'objet obj1 a recu un événement event1
```



L'intérêt de la méthode **addListener()** est qu'elle peut s'utiliser à plusieurs reprises sur le même objet, permettant ainsi de cumuler les fonctions de traitement de l'événement.

Supprimer un gestionnaire d'événements

On a vu comment ajouter une fonction de traitement de l'événement à l'aide de la méthode **addListener()**. Node a également défini une méthode inverse qui permet de supprimer une fonction de traitement ajoutée au préalable par **addListener()**. Il s'agit de la méthode **removeListener()**. Pour supprimer les gestionnaires un par un, on peut utiliser **removeAllListeners(event)**.

```
(base) josephazar@Josephs-MBP-2 webserv_events % node exemple_events3.js
1- L'objet obj1 a recu un événement event1
2- L'objet obj1 a recu un événement event1
1- L'objet obj1 a recu un événement event1
2- L'objet obj1 a recu un événement event1
1- L'objet obj1 a recu un événement event1
```

```
var events = require("events");

var obj1 = new events.EventEmitter();

obj1.addListener( eventName: "event1",  listener: function(){
    console.log("1- L'objet obj1 a recu un événement event1");
});
obj1.addListener( eventName: "event1",  listener: f = function(){
    console.log("2- L'objet obj1 a recu un événement event1");
});

obj1.emit( eventName: "event1");
obj1.emit( eventName: "event1");

obj1.removeListener( eventName: "event1", f);
obj1.emit( eventName: "event1");

obj1.removeAllListeners( event: "event1");
obj1.emit( eventName: "event1");
```

EventEmitter et fuites de mémoire

Lors de l'abonnement à des observables à longue durée de vie, il est extrêmement important de désabonner nos écouteurs une fois qu'ils ne sont plus nécessaires. Cela nous permet de libérer la mémoire utilisée par les objets dans la portée d'un écouteur et d'éviter les fuites de mémoire. Les écouteurs **EventEmitter** non désabonnés sont la principale source de fuites de mémoire dans Node.js (et JavaScript en général).

Une fuite de mémoire est un défaut logiciel par lequel la mémoire qui n'est plus nécessaire n'est pas libérée, ce qui entraîne une augmentation indéfinie de l'utilisation de la mémoire d'une application.

```
const thisTakesMemory = 'A big string....'
const listener = () => {
  console.log(thisTakesMemory)
}
emitter.on('an_event', listener)
```

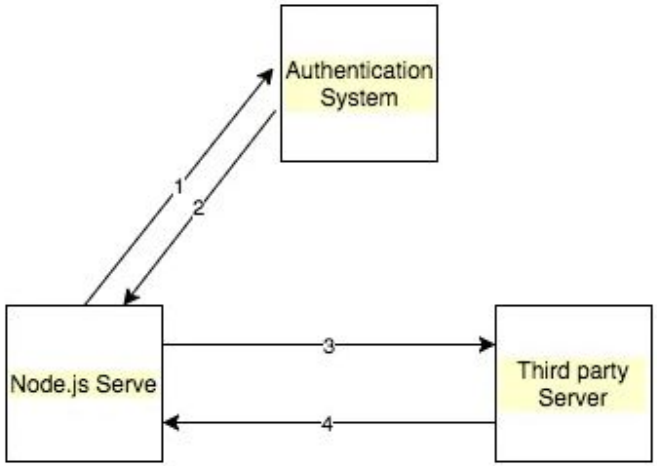
La variable **thisTakesMemory** est référencée dans l'écouteur et sa mémoire est donc conservée jusqu'à ce que l'écouteur soit libéré de l'émetteur.

EventEmitter : un cas d'utilisation pratique

Considérez la situation suivante dans laquelle vous obtenez des données d'un serveur tiers à l'aide du jeton d'authentification.

- Vous demandez le jeton d'authentification
- Jeton d'authentification reçu.
- Envoyer la requête avec un jeton d'authentification
- Reçu la réponse souhaitée.

Et une fois le jeton expiré, nous en récupérons un nouveau, le sauvegardons et continuons à faire cette nouvelle requête avec ce nouveau jeton d'authentification.



Référence : Abhinav Dhasmana, Auth token management with Node.js Observer pattern

EventEmitter : un cas d'utilisation pratique

NodeJS server

```
const Hapi = require('hapi');
var RequestPromise = require('request-promise');

// Create a server with a host and port
const server = new Hapi.Server();
server.connection({
    host: '0.0.0.0',
    port: 8000,
    routes: { cors: true }
});

// Add the route
let token = 'abc'
server.route({
    method: 'GET',
    path:'/',
    handler: function (request, reply) {
      if (token === 'abc') { // If the token has expired, get a new token
        RequestPromise(`http://0.0.0.0:8081/?token=${token}`).then((res) => {
          console.log(res);
          token = res.newToken;
          // make API calls to the Third party service. world
          return reply('hello world');
        });
      } else {
        console.log('No need for new request');
        return reply('done');
      }
    }
});

// Start the server
server.start((err) => {
    if (err) {
        throw err;
    }
    console.log('Server running at:', server.info.uri);
});
```

Auth / token generator

```
const Hapi = require('hapi');

// Create a server with a host and port
const server = new Hapi.Server();
server.connection({
    host: '0.0.0.0',
    port: 8081,
    routes: { cors: true }
});

// Add the route
server.route({
    method: 'GET',
    path:'/',
    handler: function (request, reply) {
      console.log('I am handling token:', request.query.token);
      setTimeout(() => {
        return reply({newToken: 'def'});
      }, 10000);
    }
});

// Start the server
server.start((err) => {

            if (err) {
                throw err;
            }
            console.log('Server running at:', server.info.uri);
        });
```

EventEmitter : un cas d'utilisation pratique

Maintenant, si nous exécutons les deux applications sur les ports respectifs (8080 et 8081) et exécutons ce script de test:

```
curl http://0.0.0.0:8000/
curl http://0.0.0.0:8000/
curl http://0.0.0.0:8000/
curl http://0.0.0.0:8000/
curl http://0.0.0.0:8000/
curl http://0.0.0.0:8000/
curl http://0.0.0.0:8000/
curl http://0.0.0.0:8000/
```

```
// serviceHandler output
{“newToken”:”def”}
No need for new request
No need for new request
No need for new request
No need for new request
No need for new request
No need for new request
No need for new request
No need for new request

// Token generator output
I am handling token: abc
```

EventEmitter : un cas d'utilisation pratique

Continuons et modifions notre script de test pour exécuter les commandes curl en parallèle. Cela simule plusieurs requêtes entrantes vers votre serveur Node:

```
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
curl http://0.0.0.0:8000/ &
```

```
// serviceHandler.js

{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}

// tokenGenerator
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
```

EventEmitter : un cas d'utilisation pratique

Comme vous pouvez le voir sur la sortie, toutes les requêtes ont été faites en parallèle et ont toutes fini par appeler le service d'authentification.

Ce qui devrait arriver : une seule requête a été faite au service d'authentification pour obtenir le nouveau jeton tandis que les autres requêtes attendent que la génération du jeton soit terminée. Une fois l'authentification terminée, il doit notifier aux autres requêtes d'utiliser le nouveau jeton.

```
// serviceHandler.js

{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}
{"newToken":"def"}

// tokenGenerator
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
I am handling token: abc
```

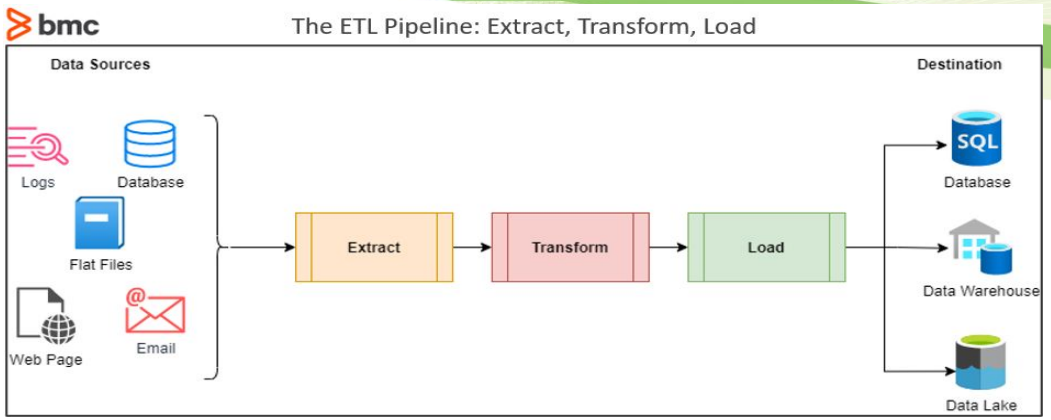
EventEmitter : un cas d'utilisation pratique

```
const util = require('util');
var RequestPromise = require('request-promise');
const EventEmitter = require('events').EventEmitter;

let inProgress = false;

function MyObservable() {
  EventEmitter.call(this);
}


// This is the function responsible for getting a refresh token
MyObservable.prototype.getToken = function() {
  let token = 'abc'
  if (!inProgress) {
    console.log('not in progress', token);
    if (token === 'abc') {
      inProgress = true;
      const rp = RequestPromise(`http://0.0.0.0:8081/?token=${token}`);
      rp.then((res) => {
        console.log(res);
        token = res.newToken;
        inProgress = false;
        this.emit('done', 'token refreshed');
      });
    } else {
      console.log('No need for new request');
    }
  }
}

util.inherits(MyObservable, EventEmitter);
module.exports = MyObservable;
```

```
const Hapi = require('hapi');
const MyObservable = require('./MyObservable');

const observable = new MyObservable();

// Create a server with a host and port
const server = new Hapi.Server();
server.connection({
    host: '0.0.0.0',
    port: 8000,
    routes: { cors: true }
});
// Add the route
server.route({
    method: 'GET',
    path:'/',
    handler: function (request, reply) {
      observable.once('done', (message) => {
          // Make API request here
        return reply('request completed');
      });
      observable.getToken();
    }
});

// Start the server
server.start((err) => {
    if (err) {
        throw err;
    }
    console.log('Server running at:', server.info.uri);
});
```

```
// betterServiceHandler
{"newToken":"def"}

// tokenGenerator
I am handling token: abc
```

ETL : Extract - Transform - Load



ETL est un processus d'extraction de données à partir d'un emplacement, de transformation d'une manière ou d'une autre, puis de chargement ailleurs. Nous pouvons utiliser ce processus pour convertir de grandes quantités de données dans Node.js d'un format à un autre.

ETL : Extract - Transform - Load

ETL est un processus d'extraction, de transformation et de chargement de données d'une ou plusieurs sources vers une destination :

Extraire : récupérez des données brutes où qu'elles se trouvent, qu'il s'agisse d'une base de données, d'une API ou de toute autre source.

Transformer : modifier les données d'une manière ou d'une autre. Il peut s'agir de restructurer les données, de renommer les clés dans les données, de supprimer des points de données invalides ou inutiles, de calculer de nouvelles valeurs ou de tout autre type de traitement de données qui convertit ce que vous avez extrait dans le format dont vous avez besoin.

Charger (Load): Déplacez les données formatées vers leur destination finale (telle qu'une base de données, un fichier CSV/JSON ou une autre structure) où d'autres personnes pourront y accéder.

ETL : Extract - Transform - Load

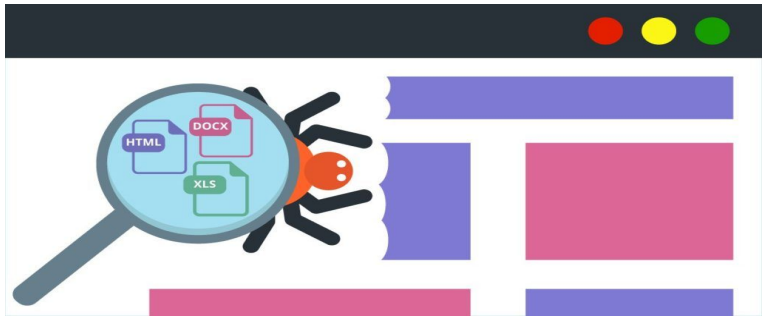
- ETL est très populaire avec les cas d'utilisation de Big Data, où les gens doivent analyser de grandes quantités de données pour découvrir des informations et des relations qui peuvent être utiles. En effet, le Big Data implique souvent de nombreux types de données différents, et une certaine quantité de nettoyage et de normalisation est nécessaire avant que l'analyse ne soit possible.
- Dans les applications Big Data, vous travaillez également souvent avec **plusieurs sources de données** en même temps, ce qui est un scénario qu'ETL maîtrise très bien. **La normalisation et l'agrégation** d'ensembles de données disparates constituent un excellent cas d'utilisation pour ETL.
- Mais ce n'est pas que du Big Data et de l'analytique ! Parfois, vous devez migrer vers un nouveau type de base de données, puis la question devient : **comment gérez-vous la traduction de votre structure de données d'un type de base de données à un autre ?** ETL peut nous aider en modifiant les données que nous avons stockées dans un type de base de données en quelque chose d'approprié pour le stockage dans un autre type de base de données.

ETL : Extract - Transform - Load

- Un processus ETL peut nécessiter beaucoup de calculs et nécessite parfois l'accès à des données qui peuvent ne pas être disponibles en temps réel. Par conséquent, les processus ETL sont généralement exécutés selon un calendrier régulier avec un lot de données (**BATCH PROCESSING**). Cependant, les solutions ETL en temps réel deviennent de plus en plus courantes (**STREAMS**).
- Le résultat d'un processus ETL est en retard par rapport à l'état réel de vos données source. Cela peut être en secondes, minutes, heures ou jours selon votre cas spécifique. Il faut du temps pour extraire, transformer et charger toutes les données nécessaires. Selon la nature et l'échelle des données avec lesquelles vous travaillez, votre destination ne sera pas à jour tant que le processus ne sera pas terminé.

Extraire des données avec le web scraping

- Le Web Scraping est utilisé pour améliorer l'efficacité industrielle en extrayant plus de détails à l'aide d'un “scraper”
- Ces données peuvent être transformées à partir d'un site Web dans un format structuré qui est très utile pour une utilisation ultérieure et du point de vue du stockage

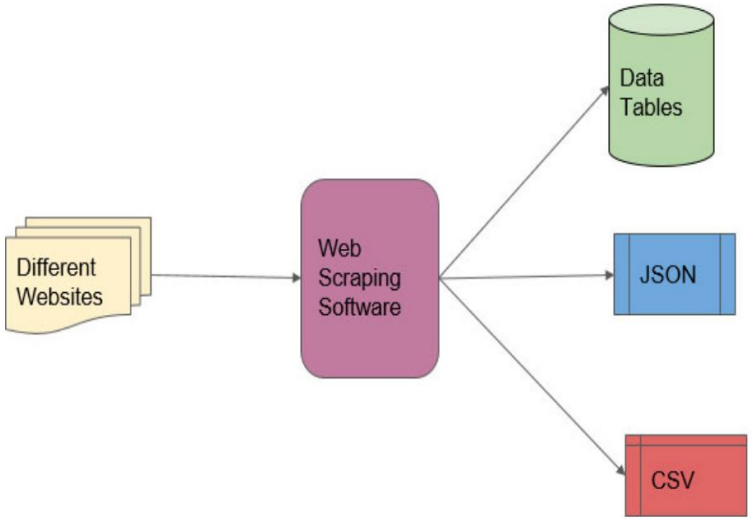


Extraire des données avec le web scraping

Scraping : Utiliser des outils/programmes informatiques pour recueillir des données que vous pouvez voir sur une page Web.

Il existe un large éventail de techniques et d'outils de web scraping. Ceux-ci peuvent être aussi simples que copier/coller, de l'analyse HTML, des API et de la programmation.

Un **scraper** extrait les données non formatées, très mal structurées des sites web.



Le web scraping, est-ce du piratage ?

- Le scraping n'est pas une activité illégale en soi.
- Lisez les termes et conditions du site Web cible pour comprendre comment vous pouvez légalement utiliser les données.
- La plupart des sites Web vous interdisent d'utiliser leurs données à des fins commerciales..
- Assurez-vous que vous ne téléchargez pas les données à un rythme trop rapide, car cela pourrait casser le site Web.
- Vous pourriez également être bloqué sur le site Web.

Le web scraping, est-ce du piratage ?

- En fait, les grandes entreprises utilisent elles-mêmes le scraping mais ne veulent pas non plus que d'autres utilisent des bots contre elles.
- La recherche Google est à la fois un robot d'exploration (**crawler**) et un **scraper**. Le robot d'exploration de Google est connu sous le nom de **Googlebot**. Grâce à l'exploration et au “scraping” des données, Googlebot découvre des pages nouvelles et mises à jour à ajouter à l'index de recherche Google.

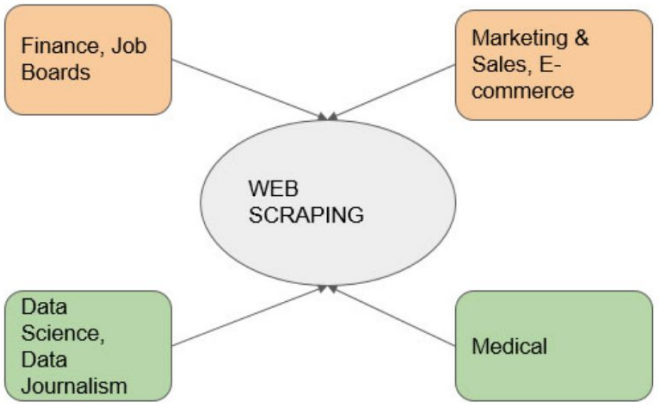
Le web scraping, est-ce du piratage ?

Bien que la collecte de données soit légale dans de nombreux pays, elle varie à travers le monde et présente une certaine ambiguïté d'un pays à l'autre. Dans cette optique, voici quelques bonnes règles à suivre :

- Veillez à ne pas causer de dommages ou utiliser les données à de mauvaises fins. Il y a beaucoup de mauvais bots et beaucoup d'entre eux finissent par être poursuivis.
- Il est très certainement illégal d'analyser, de modifier, de manipuler des données ou de les vendre à quelqu'un d'autre.

Importance et cas d'utilisation du web scraping

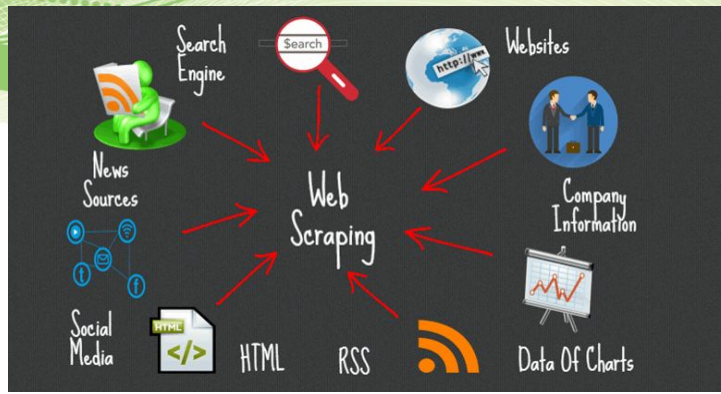
- **Apprentissage automatique :** Cela fait partie de l'intelligence artificielle où les énormes données et ensembles de données doivent être collectés et organisés de manière appropriée avant de les alimenter au système. Sur la base de ces données, cela créerait un système efficace, fonctionnel et approprié.
- **Données financières :** Il est très important d'obtenir une analyse des marchés financiers. Le scraping est utilisé pour collecter des données boursières appropriées pour une analyse appropriée.
- **Données des médias sociaux :** cette tendance s'est également accrue de nos jours pour obtenir des données plus utiles des médias sociaux afin d'assurer une publicité ciblée avec précision. C'est ainsi que les gens obtiennent des recommandations appropriées en fonction de leurs besoins sur les réseaux sociaux.



Les techniques de web scraping les plus utilisées

Extraction manuelle :

- Cela implique l'extraction manuelle et l'organisation des données des sites Web.
- Ce processus prend du temps et est inefficace.



Extraction automatisée :

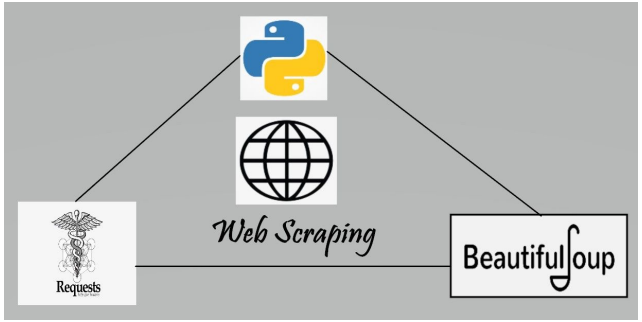
- **Parsing HTML / DOM (Document Object Model)** : Il s'agit d'une interface utilisée pour modifier et mettre à jour la structure et le contenu des documents HTML.
- **Logiciel de web scraping** : De nos jours, il existe de nombreux logiciels de web scraping disponibles, il faut donc directement fournir les en-têtes comme paramètres d'entrée et les données automatiquement pertinentes sont automatiquement disponibles dans le format confortable en CSV, JSON.

Quelques outils utiles pour le web scraping

scrapinghub

Selenium with Python

Scrapy



puppeteer TS
19.6.2 • Public • Published 3 days ago

Readme | Code Beta | 5 Dependencies | 5,579 Dependents | 789 Versions

Puppeteer

CI failing npm v19.6.2

Guides | API | FAQ | Contributing | Troubleshooting

> Puppeteer is a Node.js library which provides a high-level API to control Chrome/Chromium over the DevTools Protocol. Puppeteer runs in headless mode by default, but can be configured to run in full (non-headless) Chrome/Chromium.

Puppeteer

What can I do?

Most things that you can do manually in the browser can be done using Puppeteer! Here are a few examples to get you started:

- Generate screenshots and PDFs of pages.
- Crawl a SPA (Single-Page Application) and generate pre-rendered content (i.e. "SSR" (Server-Side Rendering)).
- Automate form submission, UI testing, keyboard input, etc.
- Create an automated testing environment using the latest JavaScript and browser features.
- Capture a timeline trace of your site to help diagnose performance issues.
- Test Chrome Extensions.

Install
`npm i puppeteer`

Repository
github.com/puppeteer/puppeteer/tree/...

Homepage
github.com/puppeteer/puppeteer/tree/...

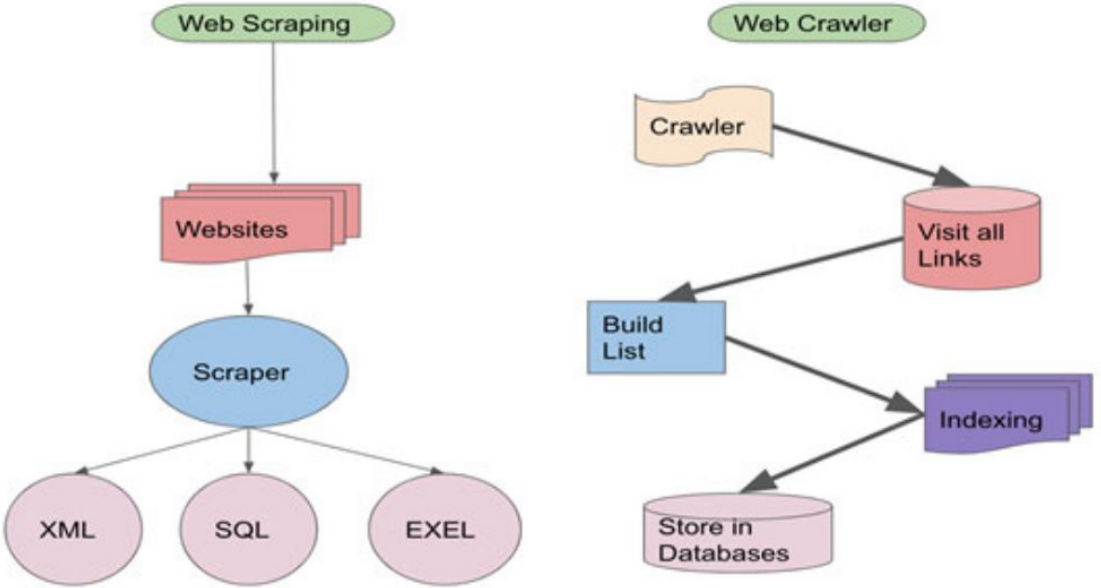
Weekly Downloads
4,366,706

Version	License
19.6.2	Apache-2.0
Unpacked Size	Total Files
329 kB	29
Issues	Pull Requests
237	9

<xpath>

Différents aspects du web scraping

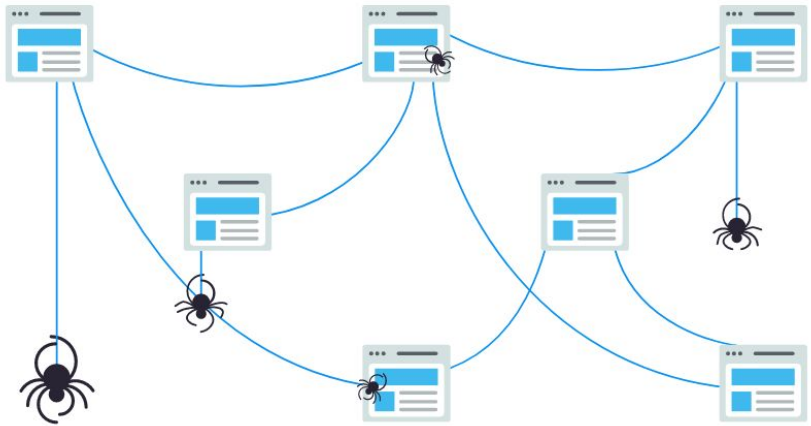
Examinons ses deux aspects les plus importants : Crawler et Scraper.



Crawler

Un robot navigue sur Internet pour indexer et rechercher du contenu pertinent. Il obtient des liens de manière logique. Il suit les étapes supplémentaires pour la collecte de données. Les étapes supplémentaires impliquées sont les suivantes :

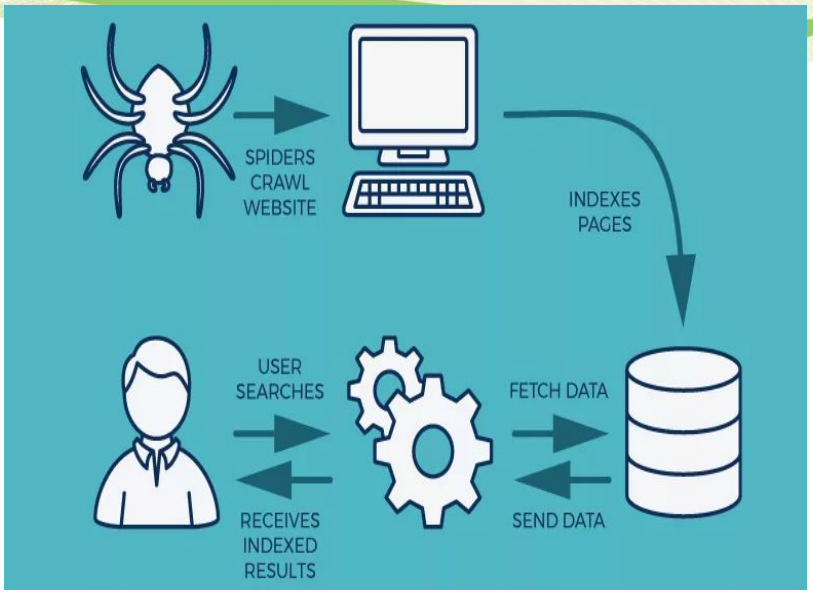
- Indexage
- Stockage dans des bases de données



Crawling est le processus de découverte dans lequel les moteurs de recherche envoient une équipe de robots (appelés crawlers ou spiders) pour trouver du contenu nouveau et mis à jour. Le contenu peut varier - il peut s'agir d'une page Web, d'une image, d'une vidéo, d'un PDF, etc. - mais quel que soit le format, le contenu est découvert par des liens

Qu'est-ce que l'indexation ?

- L'indexation consiste à stocker et à organiser les informations trouvées sur les pages. Le bot restitue le code sur la page de la même manière qu'un navigateur. Il catalogue tout le contenu, les liens et les métadonnées de la page.
- L'indexation nécessite une quantité massive de ressources informatiques, et ce n'est pas seulement le stockage de données.
- Lorsqu'un bot parcourt un site Web, il parcourt chaque page et chaque lien, jusqu'à la dernière ligne du site Web, à la recherche de TOUTE information.



Bing yahoo! G

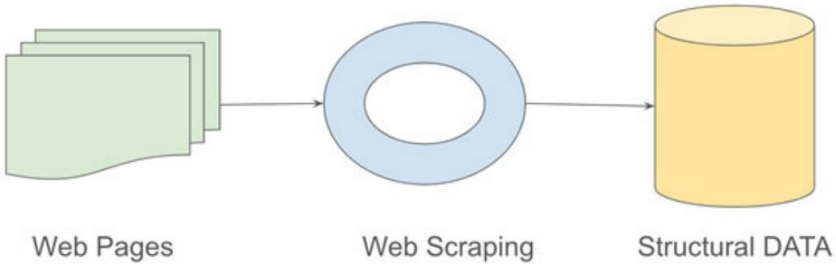
UNIVERSITÉ DE FRANCHE-COMTÉ

IUT Belfort-Montbéliard

node js

Scraper

- Il s'agit d'un outil spécialisé utilisé pour l'exploration et l'extraction précises de données. Il obtient ces valeurs à partir de HTML.
- Il peut être utilisé pour stocker des données dans un format de fichier particulier tel que JSON, XML, CSV, etc.
- Le scraper parcourt le site Web et stocke les données dans un format de fichier structuré pour une lisibilité aisée. Le scraping est également connu sous le nom de “human mining” car il génère des données faciles à lire.




DEMO_