Dev Web

côté serveur

S3 - R3.01 - 2023/2024

Semaine 2

- Récapitulatif de la première semaine
- Plus d'info sur le Javascript asynchrone
- Serveurs Web
- Communication navigateur/serveur
- Création d'un serveur web avec NodeJS (TP)

NodeJS

- Prendre un moteur JavaScript à partir du navigateur (moteur JavaScript V8 de Chrome)
  - Obtenez le même JavaScript sur le navigateur et le serveur
  - Pas besoin du DOM sur le serveur
- Inclure des événements similaires à DOM et une file d'attente d'événements
  - Tout fonctionne comme un appel de la boucle d'événement
- Créer une interface d'événement pour toutes les opérations du système d'exploitation
  - Enveloppez tous les appels de blocage du système d'exploitation (fichier et socket/réseau io)
  - Ajouter une prise en charge de la gestion des données
- Un système de module approprié

NodeJS

```
let fs = require("fs"); // require est un appel de module Node
  // l'objet fs encapsule les appels du système de fichiers du système d'exploitation
  // OS read() est synchrone mais fs.readFile de Node est asynchrone
fs.readFile("myFile", readDoneCallback); // Start read
function readDoneCallback(error, data) {
  // Convention de rappel NodeJS : le premier argument est l'objet d'erreur JavaScript
  // data est un objet Node Buffer spécial
 if (!error) {
    console.log("myFile contents", data.toString());
 }
}
```

Modules NodeJS

- Importer à l'aide de require() - Peut utiliser l'importation ES6 si le nom du fichier *.mjs
  - Module système : require("fs"); // Regarde dans les répertoires node_modules
  - Depuis un fichier : require("./XXX.js"); // Lit le fichier spécifié
  - Depuis un répertoire : require("./myModule"); // Lit myModule/index.js
- Les fichiers de module ont une portée privée
  - Require renvoie ce qui est assigné à module.exports
- De nombreux modules Node standard
  - File system, process, networking, timers, etc.
- Énorme bibliothèque de modules (npm)

```
var notGlobal;
function func1() {}
function func2() {}
module.exports = {func1: func1, func2: func2};
```

Programmation avec événements/rappels

- Différence clé
  - Threads : le blocage/l'attente est transparent
  - Événements : le blocage/l'attente nécessite un rappel
- Modèle mental
  - Si le code ne bloque pas : Identique à la programmation des threads
  - Si le code bloque (ou doit bloquer) : besoin de configurer le rappel (Callback)
  - Souvent, ce qui était une instruction de retour devient un appel de fonction

Programmation avec événements/rappels

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

```
step1(function (r1) {
 console.log('step1 done', r1);
 step2(r1, function (r2) {
   console.log('step2 done', r2);
   step3(r2, function (r3) {
      console.log('step3 done',r3);
  });
 });
}):
console.log('All Done!');
```

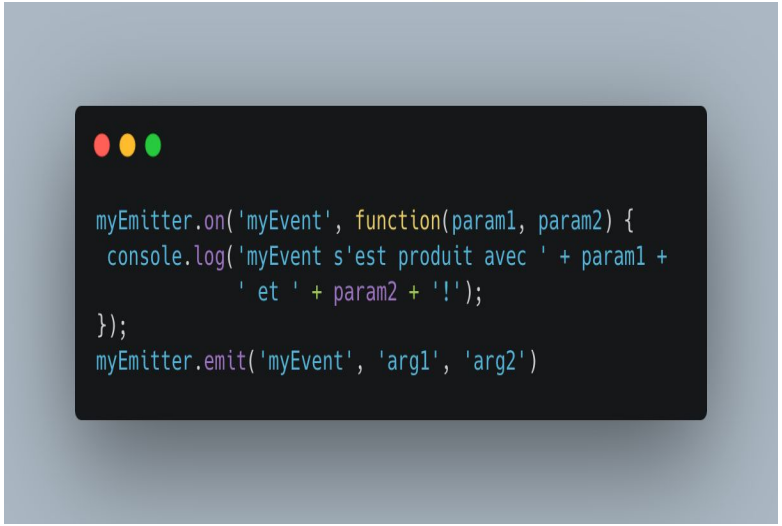
Programmation avec événements/rappels

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

```
step1(function (r1) {
 console.log('step1 done', r1);
 step2(r1, function (r2) {
   console.log('step2 done', r2);
   step3(r2, function (r3) {
      console.log('step3 done',r3);
      console.log('All Done!');
  });
 });
}):
```

Modèle d'écoute/d'émetteur (Listener/Emitter Pattern)

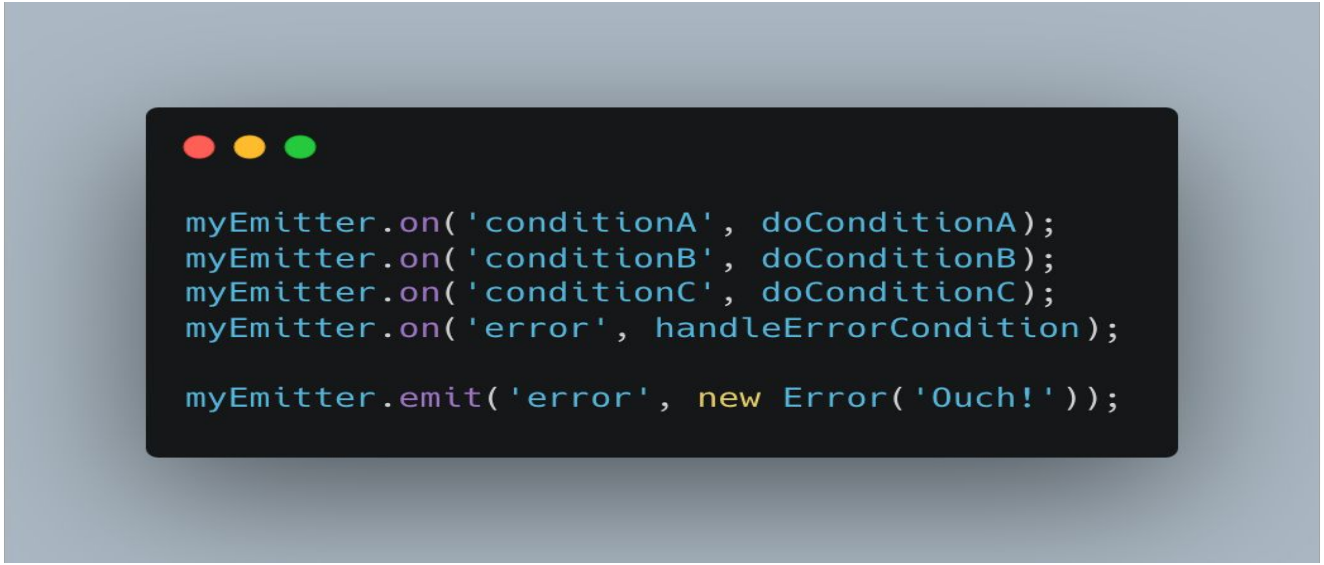
- Lors de la programmation avec des événements (plutôt que des threads), un modèle d'écouteur/émetteur est souvent utilisé.
- Listener - Fonction à appeler lorsque l'événement est signalé.
- Emetteur - Signal qu'un événement s'est produit.
- Lors de l'émission d'un appel, les écouteurs sont appelés de manière synchrone et dans l'ordre dans lequel ils ont été enregistrés.



```
myEmitter.on('myEvent', function(param1, param2) {
 console.log('myEvent s'est produit avec ' + param1 +
             ' et ' + param2 + '!');
});
myEmitter.emit('myEvent', 'arg1', 'arg2')
```

Modèles typiques d'EventEmitter

- Avoir plusieurs événements différents pour différents états ou actions.
- La gestion des "erreurs" est importante - NodeJS se ferme si l'erreur n'est pas détectée !



```
myEmitter.on('conditionA', doConditionA);
myEmitter.on('conditionB', doConditionB);
myEmitter.on('conditionC', doConditionC);
myEmitter.on('error', handleErrorCondition);

myEmitter.emit('error', new Error('Ouch!'));
```

Node.JS - de nombreux modules intégrés utiles

- Buffer
- C/C++ Addons
- Child Processes
- Cluster
- Console
- Crypto
- Debugger
- DNS
- Errors
- Events
- File System
- Globals
- HTTP
- HTTPS
- Modules
- Net
- OS
- Path
- Process
- Punycode
- Query Strings
- Readline
- REPL
- Stream
- String Decoder
- Timers
- TLS/SSL
- TTY
- UDP/Datagram
- URL
- Utilities
- V8
- VM
- ZLIB

Programmation asynchrone

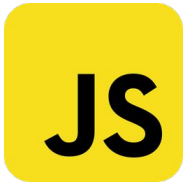
Retour rapide à Javascript (client)

- Repensez à votre premier langage de programmation (pour la plupart d'entre vous, Java/C++)
- Contrairement à un langage orienté objet comme Java, avec JS on:
  - Peut écouter des événements comme des clics
  - Peut accéder et modifier HTML/CSS avec le DOM
  - Ne pas avoir de types (par exemple, let vs int)
  - Avoir une notion (pratique ?) du vrai/faux
  - if (id("ma-boîte")) { ... // vrai si existe sur la page }
  - Peut passer des fonctions comme arguments ???

Programmation asynchrone

Alors pourquoi JS est-il si différent ?

- Java est souvent utilisé pour construire des systèmes
- Les objets sont formidables pour construire des systèmes complexes
- Les systèmes doivent être fiables - un avantage des types stricts, de la compilation et du comportement bien défini en Java
- JavaScript est utilisé pour interagir et communiquer:
  - il écoute
  - il répond
  - Il fait des requêtes



Programmation asynchrone

Javascript asynchrone

- Les programmes JS que nous avons écrits sont naturellement asynchrones
- Nous passons des fonctions en tant qu'arguments à d'autres fonctions afin que nous puissions « rappeler plus tard » une fois que nous savons que quelque chose que nous attendons s'est produit

Nous avons déjà écrit de manière asynchrone !

```
addEventListener("click", openBox)
```

Programmation asynchrone

Définir des fonctions en tant que variables

Quelle que soit la façon dont nous les créons, nous les utilisons de la même

```
function callbackFn(params) {
 ...
}

let callbackFn = function(params) {
 ...
};

let callbackFn = (params) => {
 ...
};
```

```
let result = callbackFn(params);
```

Programmation asynchrone

Passer des fonctions en arguments

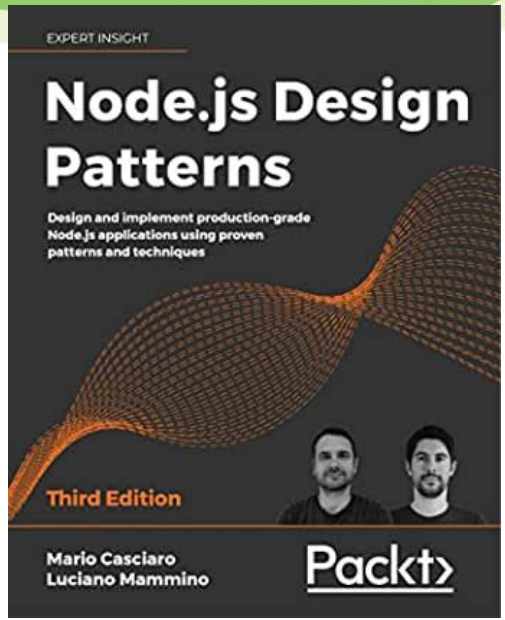
```
function askQ(qText, yesFn, noFn, handleError) {
    let answer = prompt(qText);
    if (answer === "yes") {
        yesFn();
    } else if (answer === "no") {
        noFn();
    } else {
        handleError(answer);
    }
}
```

```
function yesFn() {
    console.log("tu as répondu oui!");
}

function noFn() {
    console.log("tu as répondu non!");
}

function handleError(answer) {
    console.log("Je n'ai aucune idée de ce que signifie " +
        answer);
}
```

Callbacks

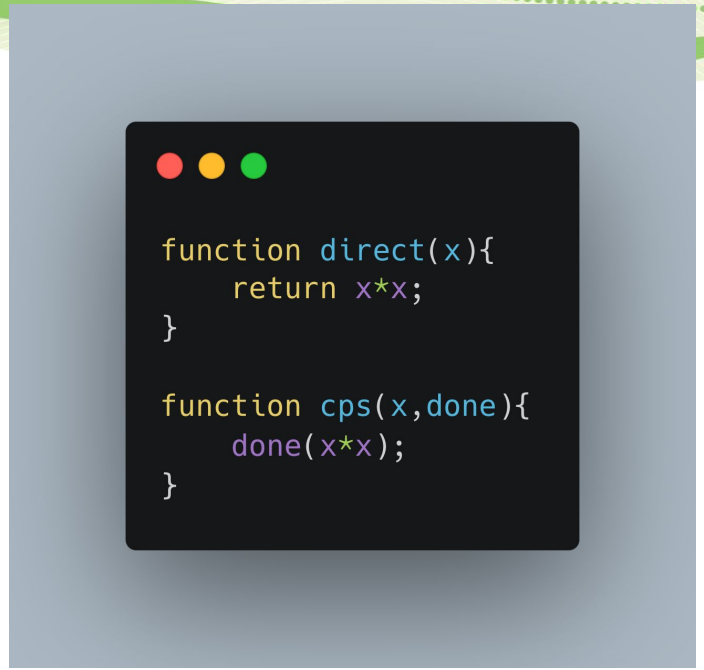
“Les rappels sont des fonctions qui sont invoquées pour propager le résultat d'une opération, et c'est exactement ce dont nous avons besoin lorsqu'il s'agit d'opérations asynchrones. Dans le monde asynchrone, ils remplacent l'utilisation de l'instruction de retour RETURN, qui, à son tour, s'exécute toujours 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.”

– Node.js Design Patterns - Third Edition, Chap 3



Callbacks

En JavaScript, un rappel est une fonction qui est transmise en tant qu'argument à une autre fonction et qui est invoquée avec le résultat lorsque l'opération est terminée. En programmation fonctionnelle, cette façon de propager le résultat est appelée style de passage de continuation (CPS: Continuous Passing Style).



```
function direct(x){
    return x*x;
}

function cps(x,done){
    done(x*x);
}
```

Callbacks

```
function addCps (a, b, callback) {
  callback(a + b)
}

// La fonction addCps() est une fonction CPS synchrone. elle est synchrone car elle ne terminera son
exécution que lorsque le rappel terminera également son exécution.

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

// Output:
// before
// Result: 3
// after
```

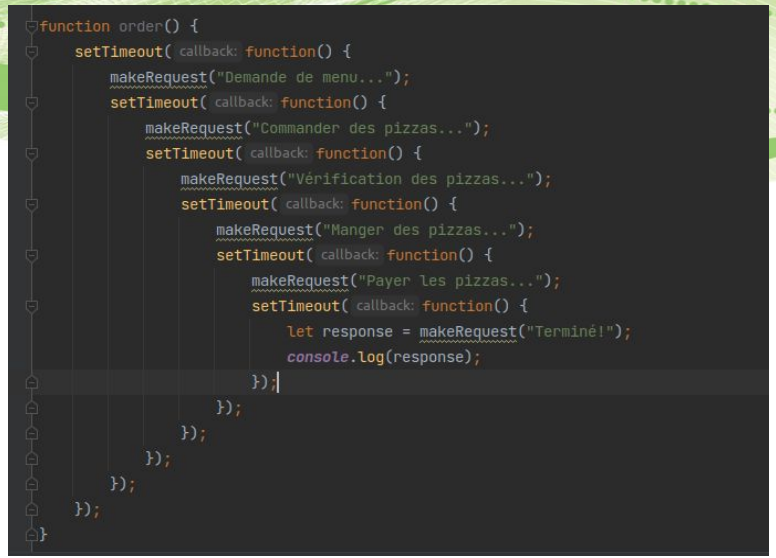
Callbacks

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

// Nous avons utilisé setTimeout() pour simuler une invocation asynchrone du rappel. setTimeout()
ajoute une tâche à la file d'attente d'événements qui est exécutée après le nombre donné de
millisecondes. Il s'agit clairement d'une opération asynchrone.

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

// Output:
// before
// after
// Result: 3
```

Programmation asynchrone

Le défi des programmes asynchrones

Les programmes asynchrones sont parfaits pour démarrer des tâches immédiatement sans bloquer d'autres fonctions. Mais lorsqu'il existe des dépendances pour les fonctions asynchrones, nous allons rencontrer le **« Callback Hell »**



```
function order() {
    setTimeout( callback: function() {
        makeRequest("Demande de menu...");
        setTimeout( callback: function() {
            makeRequest("Commander des pizzas...");
            setTimeout( callback: function() {
                makeRequest("Vérification des pizzas...");
                setTimeout( callback: function() {
                    makeRequest("Manger des pizzas...");
                    setTimeout( callback: function() {
                        makeRequest("Payer les pizzas...");
                        setTimeout( callback: function() {
                            let response = makeRequest("Terminé!");
                            console.log(response);
                        });
                    });
                });
            });
        });
    });
}
```

Le code est très déroutant et difficile à suivre le flux logique d'exécution et de données

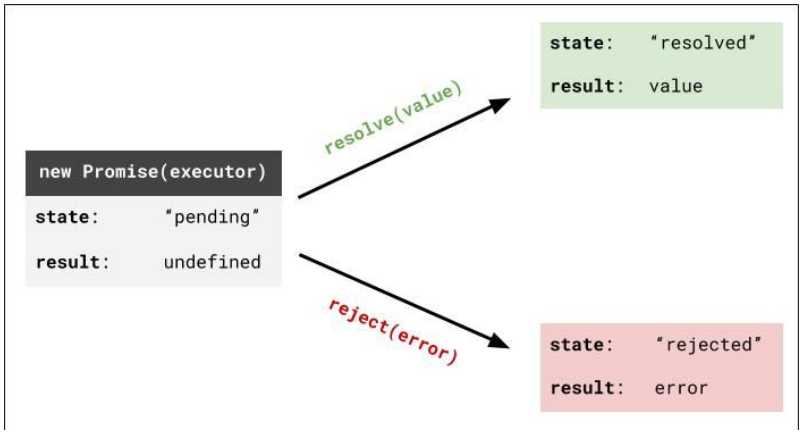
Et si nous pouvions attacher des fonctions de rappel pour chaque étape comme un pipeline pour être plus facile à suivre?

Programmation asynchrone

Promesses (Promises)

Transforment le code asynchrone en « objets » synchrones réutilisables qui peuvent être (éventuellement) résolus ou rejetés tout en laissant le programme continuer à fonctionner

Beaucoup plus facile de gérer les tâches asynchrones et synchrones sans "callback hell"



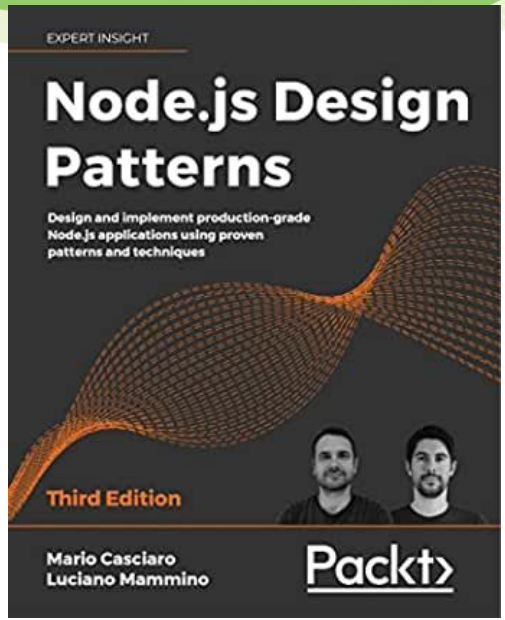
Les promesses ont trois états:

- **pending**: lors de la construction initiale
- **resolve**: une fois résolu
- **rejected**: lorsque rejeté

Promises

“Les promesses représentent un grand pas en avant vers la fourniture d'une alternative robuste aux rappels de style de passage de continuation pour propager un résultat asynchrone. L'utilisation de promesses rendra toutes les principales constructions de flux de contrôle asynchrones plus faciles à lire, moins verbeuses et plus robustes par rapport à leurs alternatives basées sur le rappel.”

– Node.js Design Patterns - Third Edition, Chap 5



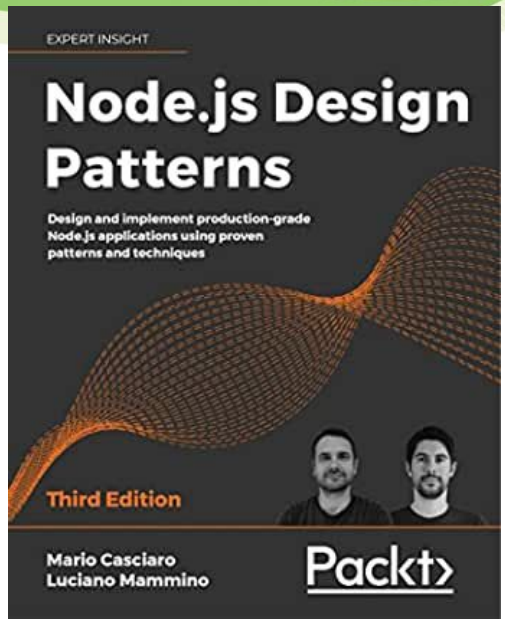
Promises

“Une promesse est un objet qui incarne le résultat éventuel (ou l'erreur) d'une opération asynchrone. Dans le jargon des promesses, nous disons qu'une promesse est en attente lorsque l'opération asynchrone n'est pas encore terminée, qu'elle est résolue lorsque l'opération se termine avec succès et rejetée lorsque l'opération se termine avec une erreur. Une fois qu'une promesse est résolue ou rejetée, elle est considérée comme réglée.

Pour recevoir la valeur de réalisation ou l'erreur (raison) associée au rejet, nous pouvons utiliser la méthode then() d'une instance Promise. Voici sa signature :

promise.then(onFulfilled, onRejected)”

– Node.js Design Patterns - Third Edition, Chap 5



Programmation asynchrone

Promesses (Promises)

```
let promise = new Promise(executorFn); // syntaxe

let customPromise = new Promise( executor: function(resolve, reject) {
    // faire quelque chose d'asynchrone (comme faire un appel ajax, définir un Timeout)
    if (success) {
        resolve(result); // Définir sur resolu
    } else {
        reject(reason);  // Définir sur rejeté
    }
});
```

Programmation asynchrone

Promesses (Promises)

```
// Promesse toujours résolu.
function alwaysYesIn1Second(resolve, reject) {
    // cette fonction est toujours certaine, pas besoin de gérer le rejet
    setTimeout( callback: () => resolve("Yes"), ms: 1000);
}

function maybeNoIn1Second(resolve, reject) {
    setTimeout( callback: () => {
        if (Math.random() < 0.5) {
            resolve("Success!");
        } else {
            reject(Error("Error!"));
        }
    }, ms: 1000);
}

// un constructeur
function maybePromise() {
    return new Promise(maybeNoIn1Second);
}
```

Programmation asynchrone

Promesses (Promises)

```
getDetails('Bob', function (err, details) {
    getLongLat(details.address, details.country, function(err, longLat) {
        getNearbyBars(longLat, function(err, bars) {
            console.log('Your nearest bar is: ' + bars[0]);
        });
    });
});
```

```
getDetails('Bob').then(function (details) {
    return getLongLat(details.address, details.country);
}).then(function (longLat) {
    return getNearbyBars(longLat);
}).then(function (bars) {
    console.log('Your nearest bar is: ' + bars[0]);
});
```

Programmation asynchrone

Promesses (Promises)

JS copy

```
const fs = require('fs')

const getFile = (fileName) => {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, data) => {
      if (err) {
        reject(err)  // calling `reject` will cause the promise to fail with or without the er
        return       // and we don't want to go any further
      }
      resolve(data)
    })
  })
}

getFile('/etc/passwd')
.then(data => console.log(data))
.catch(err => console.error(err))
```

https://nodejs.dev/learn/understanding-javascript-promises

Programmation asynchrone

Promesses (Promises)

Example:

```js
const f1 = fetch('/something.json')
const f2 = fetch('/something2.json')

Promise.all([f1, f2])
  .then(res => {
    console.log('Array of results', res)
  })
  .catch(err => {
    console.error(err)
  })
```

The ES2015 destructuring assignment syntax allows you to also do

```js
Promise.all([f1, f2]).then(([res1, res2]) => {
  console.log('Results', res1, res2)
})
```

https://nodejs.dev/learn/understanding-javascript-promises

Programmation asynchrone

async/await

- Nous avons vu les callbacks et les promesses comme des moyens de traiter un comportement incertain (asynchrone).
- Plus récemment (2017), des mots-clés async/await ont été ajoutés à JS pour que le code asynchrone se sente synchrone, tout en obtenant les avantages d'efficacité.

```
// In named functions:
async function fnName() { ... }

// Arrow functions:
async () => { ... }

// Anonymous functions
async function() { ... }
```

- Le mot-clé async étiquettera une fonction comme ayant du code asynchrone.
- Pour l'utiliser, utilisez simplement le mot-clé async avant la déclaration de fonction.
- La valeur de retour de la fonction asynchrone sera enveloppée dans une promesse.

Programmation asynchrone

async/await

Dans cet exemple, nous pouvons voir comment async encapsule une fonction qui renvoie une promesse qui se résout à la valeur renvoyée

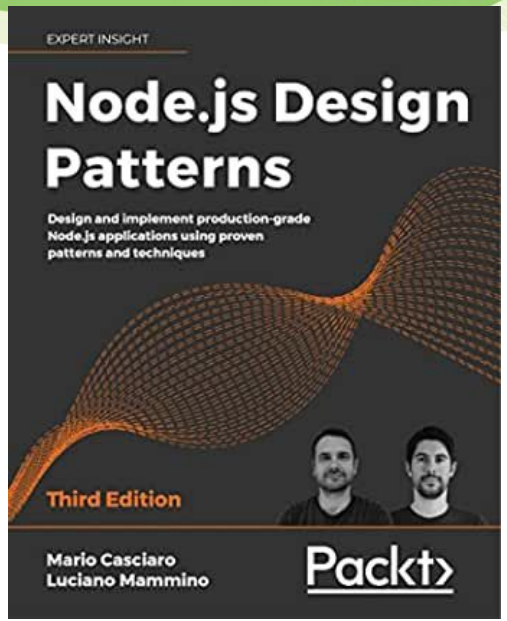
Tout comme les autres promesses, nous pouvons utiliser .then pour extraire la valeur résolue

```
function sayHello(name) {
  return "Hello " + name;
}

console.log(sayHello("dubs")); // Hello dubs

async function sayHelloAsync(name) {
   return "Hello " + name;
}

sayHelloAsync("dubs"); // Promise <pending>
sayHelloAsync("dubs").then(console.log); // Hello dubs
```

async/await

“La dichotomie async/await nous permet d'écrire des fonctions qui semblent bloquer à chaque opération asynchrone, attendant les résultats avant de continuer avec l'instruction suivante. Une fonction async est un type spécial de fonction dans lequel il est possible d'utiliser l'expression await pour "mettre en pause" l'exécution d'une promesse donnée jusqu'à ce qu'elle se résolve.”

– Node.js Design Patterns - Third Edition, Chap 5



Programmation asynchrone

async/await

- Dans une fonction asynchrone, nous pouvons maintenant utiliser await sur les appels de fonction asynchrones.
- Sur chaque ligne avec await, le moteur JS attendra la fin de la fonction.
- Le résultat de l'appel de fonction attendu sera la valeur résolue.

```
function doubleAfter1s(n) {
  return new Promise((resolve, reject) => {
    setTimeout(() => { resolve(n * 2); }, 1000);
  });
}

// standard .then solution
doubleAfter1s(2) // 4
  .then(doubleAfter1s) // 8
  .then(doubleAfter1s) // 16
  .then(result => console.log(result)); // 16 (after seconds)

// equivalent async/await solution (as anonymous function call)
(async () => {
  let a = await doubleAfter1s(2); // 4
  let b = await doubleAfter1s(a); // 8
  let c = await doubleAfter1s(b); // 16
  console.log(c); // 16 (after 3 seconds)
})();
```

Programmation asynchrone

Gestion des erreurs avec async/await

- Pour la gestion des erreurs avec async/await, utilisez try/catch au lieu de .then/.catch.
- L'instruction catch interceptera toutes les erreurs qui se produisent dans le bloc then (que ce soit dans une promesse ou une erreur de syntaxe dans la fonction), similaire au .catch dans une chaîne de promesse.
- Si vous n'avez pas try/catch dans la fonction async mais qu'une erreur se produit, la promesse sera rejetée.

```
function doubleAfter1sUnless4(n) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (n === 4) {
        reject("Rejected!");
      } else {
        resolve(n * 2);
      }
    }, 1000);
  });
}

async function delayDoubles() {
  try {
    let a = await doubleAfter1sUnless4(2); // 4
    let b = await doubleAfter1sUnless4(a); // error!
    let c = await doubleAfter1sUnless4(b); // unreached
  } catch (err) {
    console.error(err); //  "Rejected!"
  }
}
delayDoubles(); // Rejected! (after 2 seconds)
```

Programmation asynchrone

Async/Await bloque-t-il le thread principal ?

Supposons que vous souhaitiez effectuer un appel d'API à l'aide d'une promesse et que la réponse de cet appel d'API doit être transmise à une autre promesse et à nouveau la réponse de cette promesse que vous souhaitez transférer vers une autre promesse. C'est ce qu'on appelle l'enchaînement des promesses.

```
function getUserData()
promise.then(response1 => {
 promise.then(response2 => {
  promise.then(response3 => {
     console.log("Inside the promise");
  }}
 }}
})
getUserData()
```

Programmation asynchrone

Async/Await bloque-t-il le thread principal ?

Au début, si vous le regardez, tout semble bien. Mais si vous observez attentivement, jusqu'à ce que la première attente soit exécutée et que la réponse1 reçoive la réponse de l'appel d'API, le contrôle ne sera pas passé à la deuxième ligne et la même chose se poursuivra jusqu'à la réponse3. Ne pensez-vous pas qu'il bloque le thread principal ?

Juste pour vous rendre plus confus, regardez cet extrait de code et devinez le résultat.

```
async function getUserData(){
  let response1 = await fetch('https://jsonplaceholder.typicode.com/users');
  let response2 = await fetch('https://jsonplaceholder.typicode.com/users');
  let response3 = await fetch('https://jsonplaceholder.typicode.com/users');
  console.log("After all promise is executed");
}
getUserData();
console.log("Hello World");
```

Programmation asynchrone

Async/Await bloque-t-il le thread principal ?

La réponse est "Hello World" sera imprimé en premier. Bien que cela crée une confusion, en réalité async et await ne bloqueront pas le thread principal JavaScript. Comme mentionné précédemment, ce ne sont que des sucres syntaxiques pour le chaînage de promesses.

```
async function getUserData(){
  let response1 = await fetch('https://jsonplaceholder.typicode.com/users');
  let response2 = await fetch('https://jsonplaceholder.typicode.com/users');
  let response3 = await fetch('https://jsonplaceholder.typicode.com/users');
  console.log("After all promise is executed");
}
getUserData();
console.log("Hello World");
```

Programmation asynchrone

Callbacks vs Promises vs async/await

Callback vs Promise vs Async

#CALLBACK

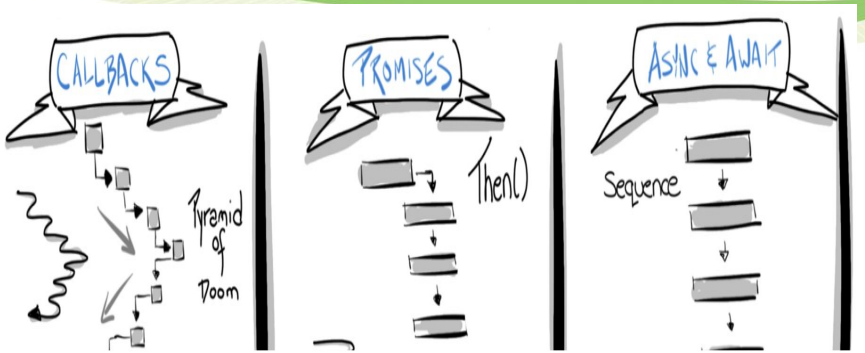
```
GetUser(function(err, user){
    GetProfile(user, function(err, profile){
        GetAccount(profile, function(err, acc){
            GetReport(acc, function(err, report){
                SendStatistics(report, function(e){
                    ...
                });
            });
        });
    });
});
```

#PROMISE

```
GetUser()
    .then(GetProfile)
    .then(GetAccount)
    .then(GetReport)
    .then(SendStatistics)
    .then(function (success) {
        console.log(success)
    })
    .catch(function (e) {
        console.error(e)
    })
```

#ASYNC/AWAIT

```
async function SendAsync() {

    let user    = await GetUser(1)
    let profile = await GetProfile(user);
    let account = await GetAccount(profile);
    let report  = await GetReport(account);

    let send = SendStatistic(report);

    console.log(send)
}
```



Les serveurs & Programmation côté serveur

Le type de programmation Web que vous avez effectué s'appelle la programmation "côté client":

- Le code que nous écrivons est exécuté dans un navigateur sur la machine de l'utilisateur (client)

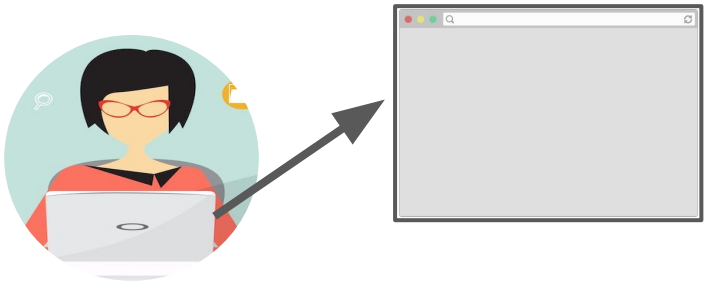
Dans ce cours, nous allons commencer à apprendre la programmation côté serveur :

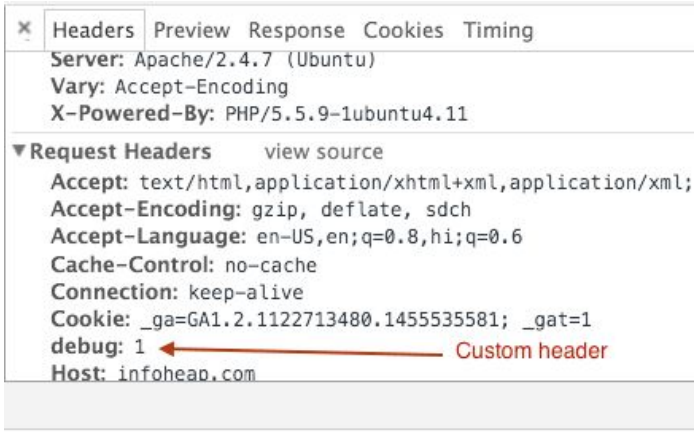
- Le code que nous écrivons est exécuté sur un serveur.

Les serveurs sont des ordinateurs qui exécutent des programmes pour générer des pages Web et d'autres ressources Web.

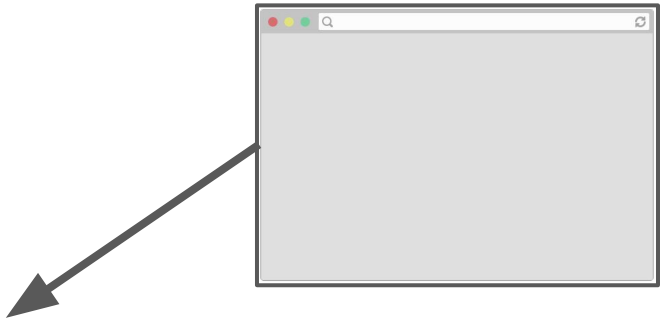
CLIENT: Vous tapez une URL dans la barre d'adresse et appuyez sur "Entrée"

Le navigateur envoie une requête HTTP GET indiquant "Veuillez me procurer le fichier index.html sur https://cours-info.iut-bm.univ-fcomte.fr/"



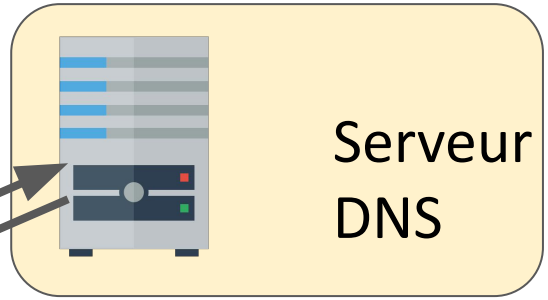


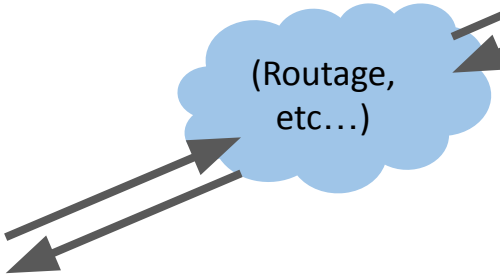
Le code C++ du navigateur crée un tableau d'octets formaté à l'aide du format de message de requête HTTP



Le navigateur demande au système d'exploitation "Hé, pouvez-vous envoyer ce message de requête HTTP Get à https://cours-info.iut-bm.univ-fcomte.fr/" ?

Le système d'exploitation envoie une requête DNS pour rechercher l'adresse IP de cours-info.iut-bm.univ-fcomte.fr



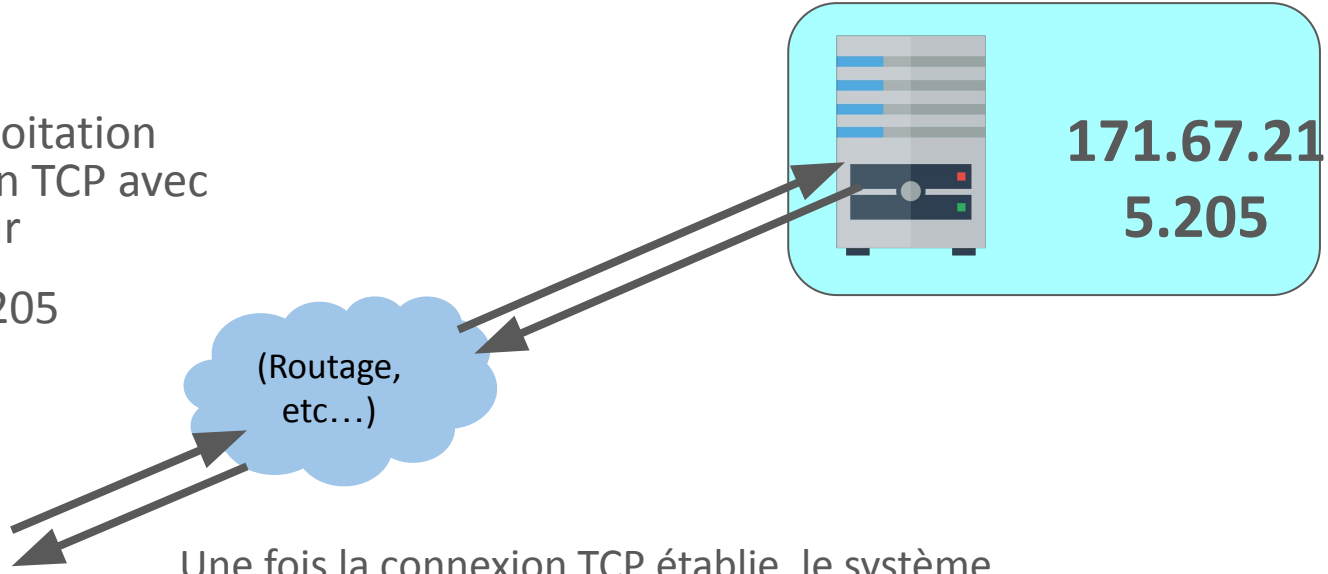


Le serveur DNS répond avec l'adresse IP, par ex. 171.67.215.205

- **DNS : Domain Name System : Traduit les noms de domaine en adresse IP de l'ordinateur associé à cette adresse.**
- **Adresse IP : Identifiant numérique unique pour chaque ordinateur connecté à Internet.**

Le système d'exploitation ouvre une connexion TCP avec l'ordinateur

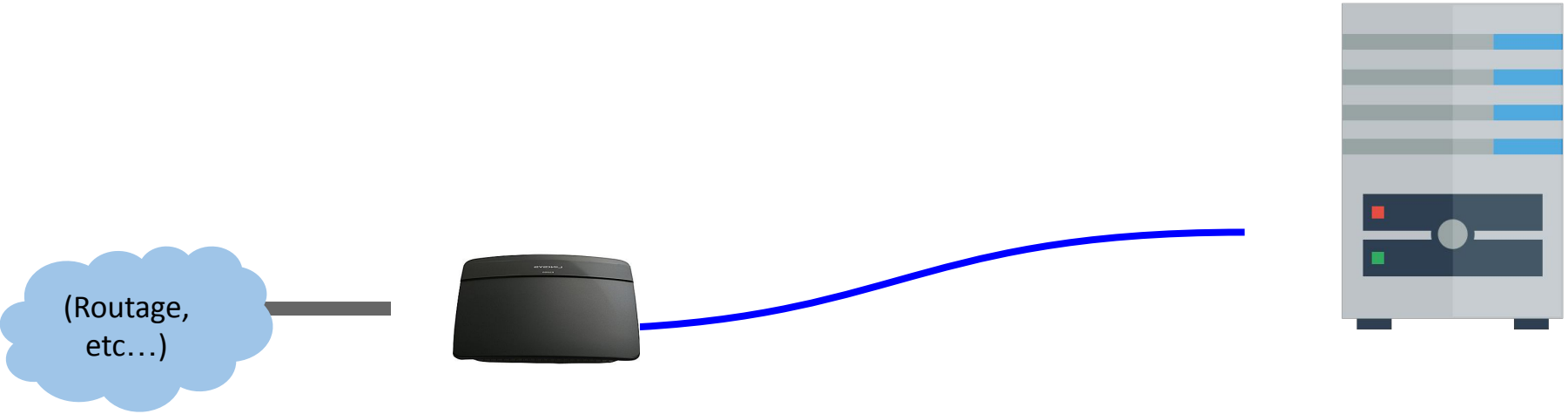
171.67.215.205



Une fois la connexion TCP établie, le système d'exploitation peut envoyer le message HTTP à 171.67.215.205 via la connexion TCP.

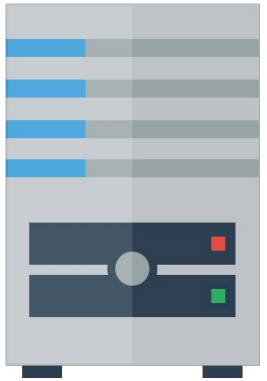
- **TCP : Transmission Control Protocol, définit le format de données pour envoyer des informations sur le fil. (Peut être utilisé pour HTTP, FTP, etc.)**

171.67.215.205



SERVEUR : Il y a un ordinateur qui est connecté à Internet à l'adresse IP 171.67.215.205.

171.67.215.205



Sur cet ordinateur se trouve un **programme de serveur Web** :

- Le programme du serveur Web écoute les messages entrants qui lui sont envoyés.
- Le programme du serveur Web peut répondre aux messages qui lui sont envoyés.

NodeJS: La plate-forme que nous utiliserons pour créer un programme de serveur Web qui recevra et répondra aux requêtes HTTP.

URLs et serveurs Web

```
https://server/path/file
```

Habituellement, lorsque vous tapez une URL dans votre navigateur:

- votre ordinateur recherche l'adresse IP du serveur à l'aide de DNS
- votre navigateur se connecte à cette adresse IP et demande le fichier donné
- le logiciel Web Server (E.G. Apache) prend ce fichier à partir du système de fichiers local du serveur, puis renvoie son contenu à vous

Certaines URL spécifient des programmes que le serveur Web doit exécuter, puis vous renvoyer leur sortie à la suite:

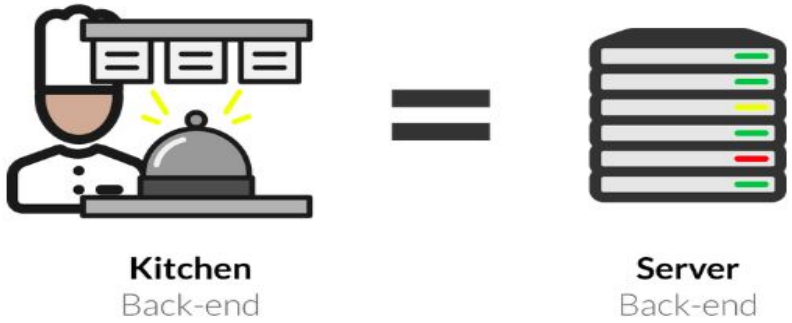
https://iut-bm.univ-fcomte.fr/test.php

L'URL ci-dessus indique au serveur iut-bm.univ-fcomte.fr d'exécuter le programme test.php et renvoyer sa sortie.

URLs et serveurs Web

Les serveurs sont des ordinateurs dédiés pour traiter les données de manière efficace et déléguant des demandes envoyées de nombreux clients (souvent à la fois).

Ces tâches ne sont pas possibles (ou appropriées) dans le navigateur du client.



Serveurs Web et HTTP

Qu'est-ce qu'un serveur Web?

Un serveur Web est un logiciel conçu pour répondre aux demandes sur Internet en chargeant ou en traitant des données. Pensez à un serveur Web comme un caissier de banque, dont le travail consiste à traiter votre demande de dépôt, de retrait ou de visualisation de l'argent sur votre compte

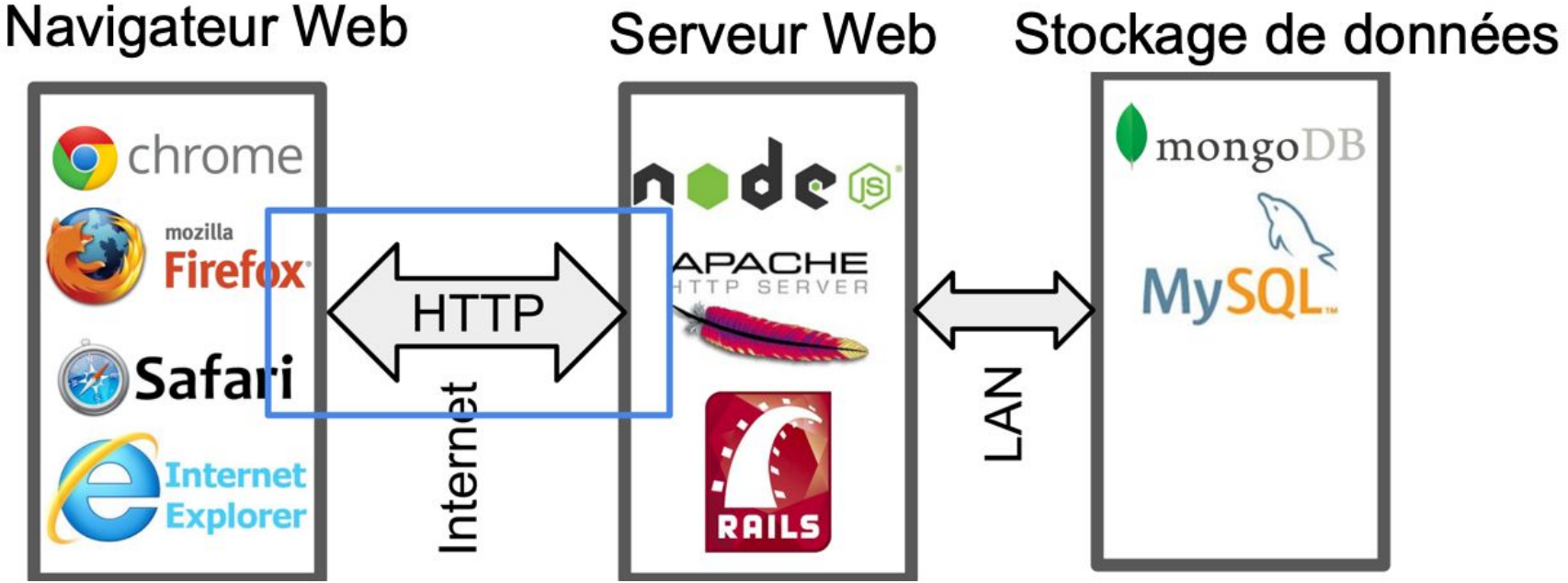
Qu'est-ce que le HTTP?

Tout comme le caissier de banque suit un protocole pour s'assurer qu'il traite correctement votre demande, les serveurs Web suivent le protocole HTTP (Hypertext Transfer Protocol), un système normalisé mondialement observé pour la visualisation de pages Web et l'envoi de données sur Internet.

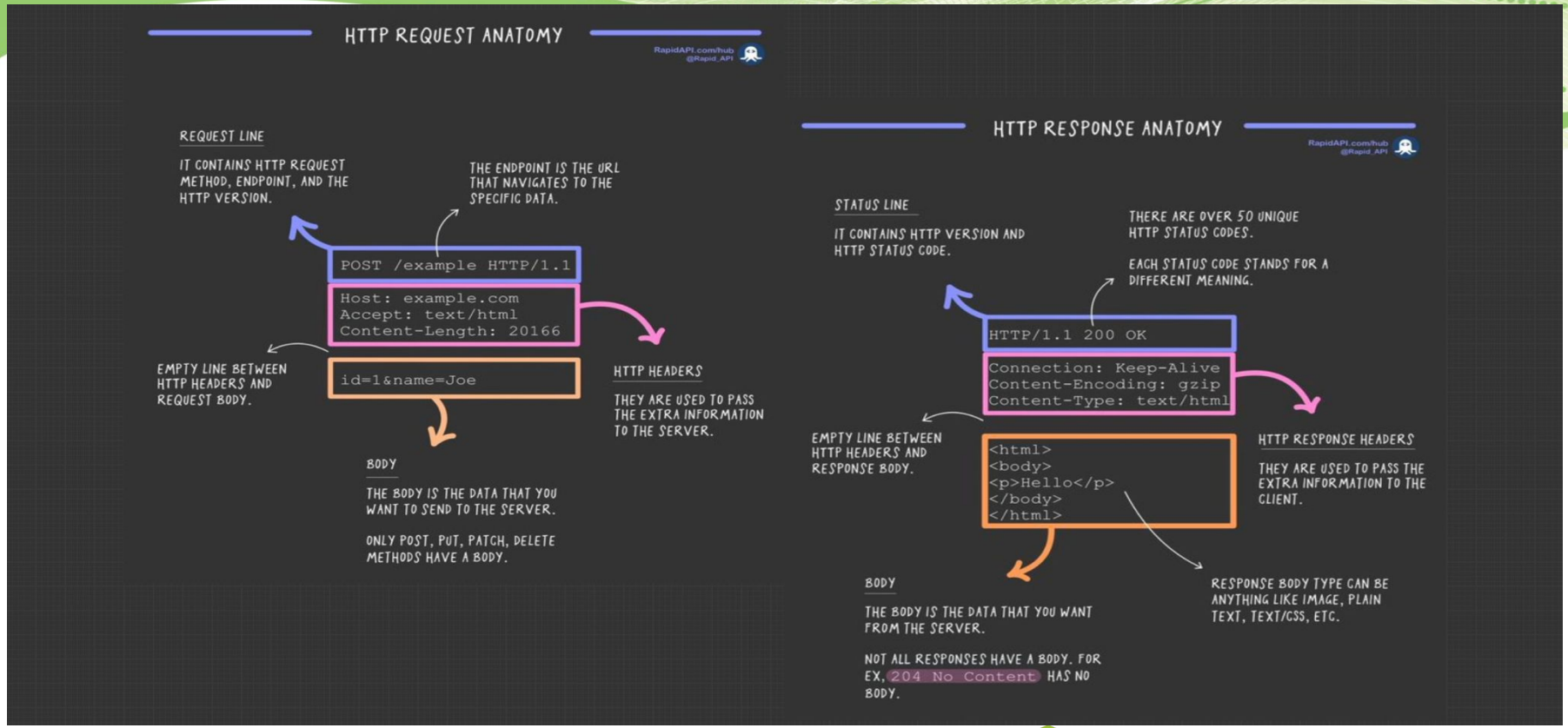
Le protocole HTTP

- HTTP = HyperText Transfer Protocol
- Le HTTP est un protocole de couche d'application qui permet aux applications Web de communiquer et d'échanger des données
- Le HTTP est le messager du web
- C'est un protocole basé sur TCP/IP
- Il est utilisé pour fournir des contenus, par exemple, des images, des vidéos, des audios, des documents
- Les ordinateurs qui communiquent via le HTTP doivent parler le protocole HTTP

Architecture d'applications Web

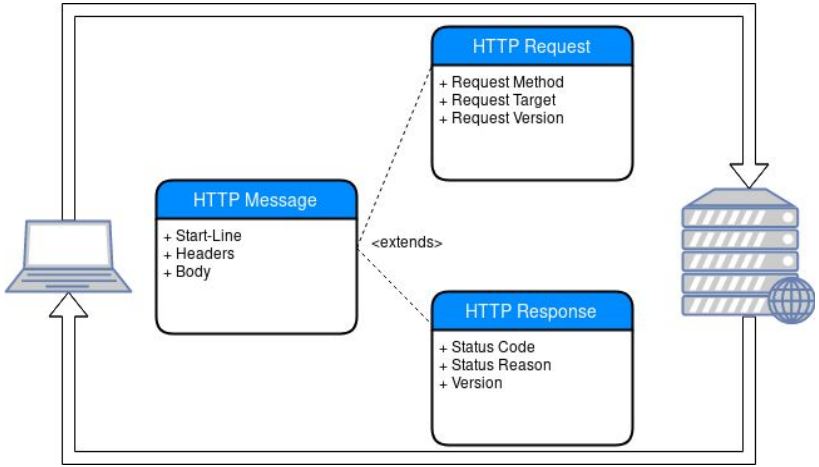


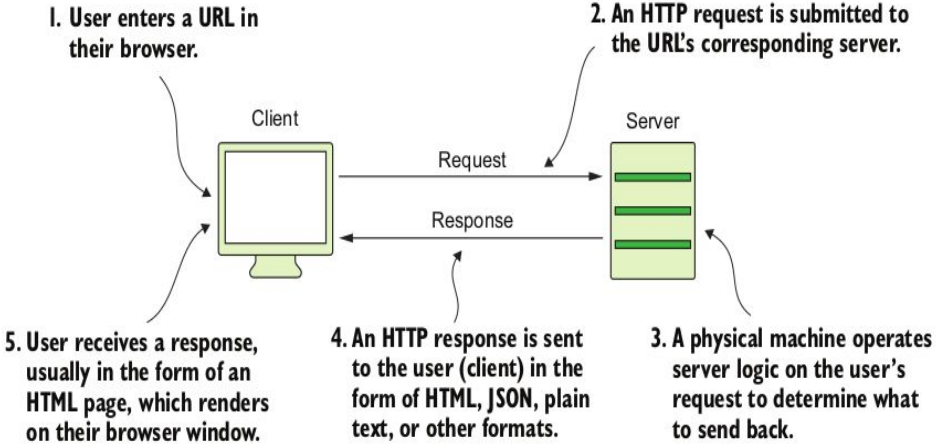
HTTP


HTTP REQUEST ANATOMY
RapidAPI.com/hub
@Rapid_API
REQUEST LINE
IT CONTAINS HTTP REQUEST METHOD, ENDPOINT, AND THE HTTP VERSION.
THE ENDPOINT IS THE URL THAT NAVIGATES TO THE SPECIFIC DATA.
POST /example HTTP/1.1
Host: example.com
Accept: text/html
Content-Length: 20166
id=1&name=Joe
EMPTY LINE BETWEEN HTTP HEADERS AND REQUEST BODY.
HTTP HEADERS
THEY ARE USED TO PASS THE EXTRA INFORMATION TO THE SERVER.
BODY
THE BODY IS THE DATA THAT YOU WANT TO SEND TO THE SERVER.
ONLY POST, PUT, PATCH, DELETE METHODS HAVE A BODY.
HTTP RESPONSE ANATOMY
RapidAPI.com/hub
@Rapid_API
STATUS LINE
IT CONTAINS HTTP VERSION AND HTTP STATUS CODE.
THERE ARE OVER 50 UNIQUE HTTP STATUS CODES.
EACH STATUS CODE STANDS FOR A DIFFERENT MEANING.
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html
<html>
<body>
Hello
</body>
</html>
EMPTY LINE BETWEEN HTTP HEADERS AND RESPONSE BODY.
HTTP RESPONSE HEADERS
THEY ARE USED TO PASS THE EXTRA INFORMATION TO THE CLIENT.
BODY
THE BODY IS THE DATA THAT YOU WANT FROM THE SERVER.
NOT ALL RESPONSES HAVE A BODY. FOR EX, 204 No Content HAS NO BODY.
RESPONSE BODY TYPE CAN BE ANYTHING LIKE IMAGE, PLAIN TEXT, TEXT/CSS, ETC.

Cycle request-response

Lorsque vous entrez l'URL que vous souhaitez voir dans votre navigateur, une requête HTTP est envoyée à un ordinateur physique ailleurs. Cette demande contient des informations indiquant si vous souhaitez charger une page Web ou envoyer des informations à cet ordinateur. Lorsque le serveur traite la demande, il renvoie un paquet de données sous la forme d'une réponse. Ce processus est la façon dont la plupart de vos interactions sur Internet sont facilitées





Méthodes HTTP (verbes)

- **GET** - La méthode HTTP GET est utilisée pour lire (ou récupérer) une représentation d'une ressource.
- **PUT** - Mettre à jour/Remplacer.
- **POST** - Envoyer des données de formulaire à une URL et obtenir une réponse en retour.
- **DELETE** - Supprimer une ressource identifiée par un URI.

GET et POST (formulaires) sont couramment utilisés.

Codes d'état de réponse HTTP courants

- 200 OK ⇒ Succès
- 307 Redirection temporaire Redirection ⇒ Le navigateur tente à nouveau d'utiliser l'en-tête d'emplacement
- 404 Introuvable ⇒ Célèbre!
- 503 Service indisponible ⇒ Quelque chose s'est écrasé sur le serveur
- 500 Erreur interne du serveur ⇒ Quelque chose ne va pas sur le serveur
- 501 Non implémenté
- 400 Bad Request ⇒ À utiliser si l'application Web envoie une fausse requête
- 401 Utilisation non autorisée ⇒ si l'utilisateur n'est pas connecté
- 403 Utilisation interdite ⇒ si même la connexion ne vous aiderait pas
- 550 Autorisation refusée ⇒ Ne pas autoriser à exécuter la demande

Codes d'état de réponse HTTP courants

TP : Création d'un serveur Web avec NodeJS