Présentation du cours

Jour 1 :

- Qu'est-ce que NodeJS?
- MEAN stack
- NPM et Package.json
- Le "Hello world"
- Routage

Jour 2 :

- Async (gestion de multiples requêtes et callbacks)
- Manipulation de fichiers
- PM2 : pour garder le serveur actif en permanence

Jour 3 :

- Tests unitaires
- MongoDB

Jour 4 :

- L'authentification
  - Simple bouton Google
  - PassportJS et les sessions
  - PassportJS et les JWT (JSON Web Token)
- Utilisation de middlewares

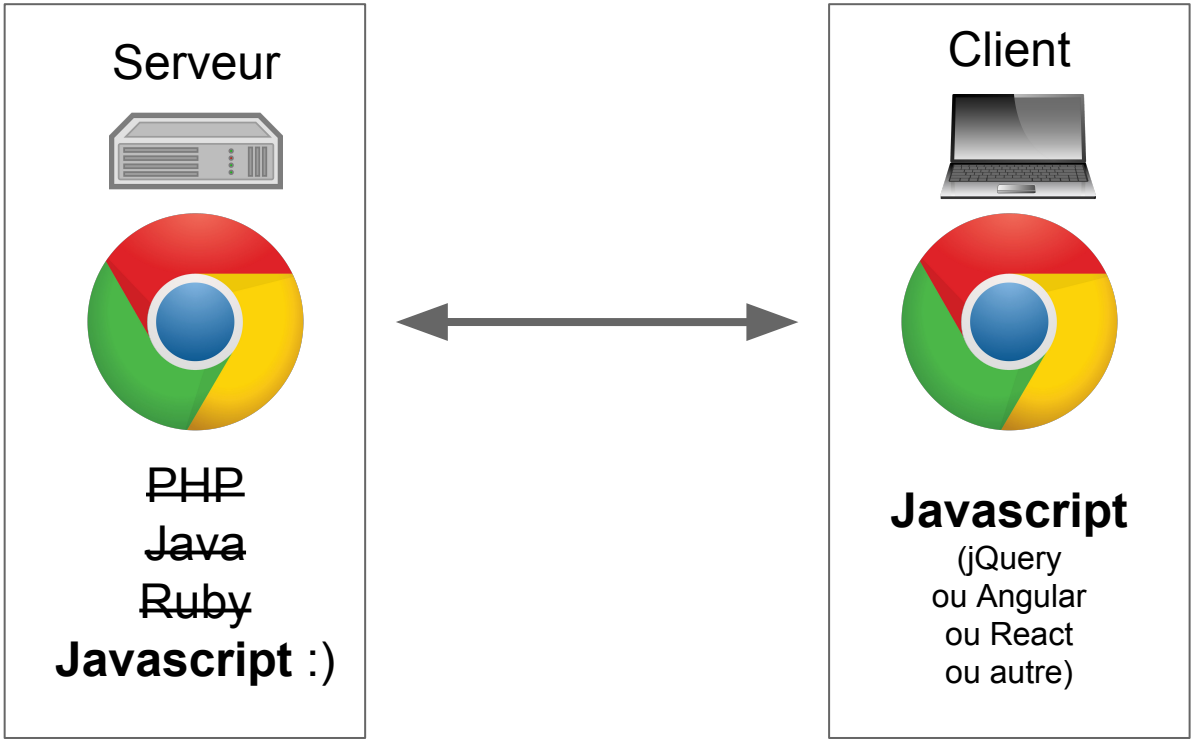
Jour 5 :

- Authentification (suite)
- Upload de fichiers avec Multer
- SocketIO

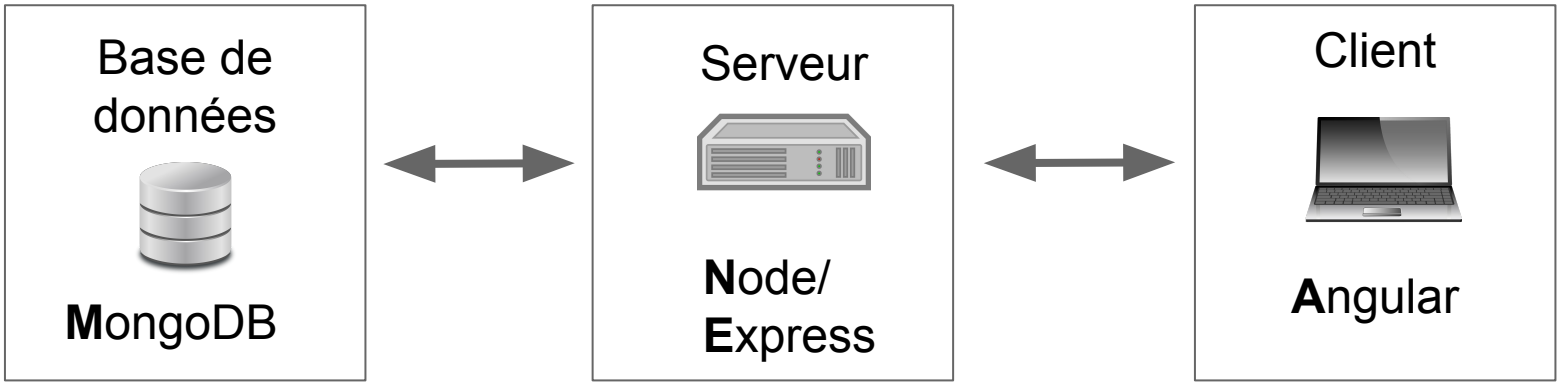
Jour 1 :

- Qu’est-ce que NodeJS?
- MEAN stack
- NPM et Package.json
- Le “Hello world”
- Routage

NodeJS = Javascript sur le serveur aussi!



mean stack = Javascript à tous les étages



Avantages

- Asynchrone donc très rapide, bien que monothread
- Bénéficie de la vitesse de V8, développé par Google, et suit son développement en parallèle
- Un seul langage pour toute l'application : JS

Inconvénients :

- Le Javascript est détesté par beaucoup car il est dynamique, non typé et il fait pas mal de bizarreries
- Facilité de créer des "callback hell"

Installation

Télécharger et installer Node sur nodejs.org

Une fois installé, on peut exécuter du javascript dans un terminal.

On peut également écrire du JS dans un fichier .js, et l'exécuter en tapant :
> node path/to/file.js

00_simple_test/test.js

Hello world! Notre tout premier serveur

01_00_hello_world/server.js

```
const http = require('http');

const server = http.createServer( (request, response) => {
    response.end('It Works!! Path Hit: ' + request.url);
});

server.listen(8080);
```

Url,
Headers,
Body,
files...

Request

Server

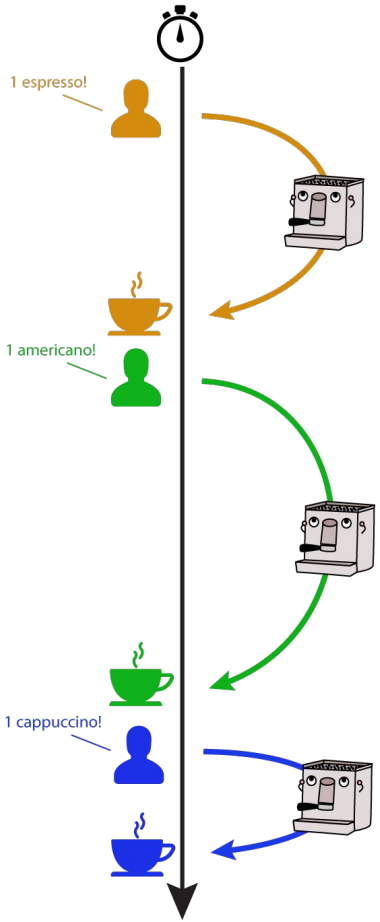
Client

Response

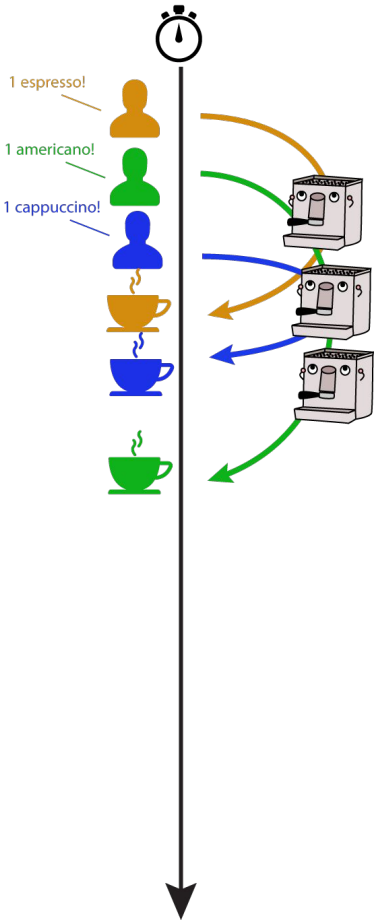
File,
String,
JSON,
Status code,
etc.

PHP (bloquant)
vs Node (non-bloquant)

PHP (Boulangerie)



Node (Starbucks)



Asynchronisme

```
const http = require('http');

const server = http.createServer( (req, res) => {
    console.log("~~~ 1");
    res.end('It Works!! Path Hit: ' + req.url);
});


server.listen(8080, () => {
    console.log("~~~ 2");
    console.log("Server listening on: http://localhost:8080");
});

console.log("~~~ 3");
```

Routes : utilisation du framework Express

```
const express = require('express');
const app = express();
const server = require('http').createServer(app);

const port = 8080;

app.get('/',       (req,res) => res.sendFile(__dirname + '/index.html'))

app.get('/test',   (req,res) => res.send('Hit route /test'))

server.listen(port);
```

```
Error: Cannot find module 'express'
```

- Installation du module avec NPM : `npm install express`
- L'application fonctionne. Express a été installé dans le dossier `node_modules` et est accessible automatiquement via `require('express')`
- Que se passe-t-il si l'on supprime le dossier `node_modules` ?
- Mémorisation des dépendances dans un fichier `package.json`
- Beaucoup, beaucoup de modules disponibles via `npm install`

Autre syntaxe pour les routes

```
app
     .get('/test',  (req,res) => res.send('...'))
     .get('/login', (req,res) => res.send('...'))
     .post('/test', (req,res) => res.send('...'))
     .post('/login',(req,res) => res.send('...'))
```

```
app
     .route('/login')
     .get( (req,res) => res.send('...'))
     .post((req,res) => res.send('...'))
app
     .route('/test')
     .get( (req,res) => res.send('...'))
     .post((req,res) => res.send('...'))
```

Comment atteindre ces routes ?

- GET :
  - simplement par le navigateur (barre d'adresse)
  - Javascript : `$.ajax({type:"GET"},...)` ou `$.get(...)` ou `Angular http.get(...)`, etc.

- POST :
  - Javascript : `$.ajax({type:"POST"},...)` ou `$.post('/test')` ou `Angular http.post(...)`, etc.
  - Outils comme Postman (on y reviendra)

Passer un objet vers Node

Front-end : (exemple avec jQuery)

```
$.post("/users", { user : "Albert" })
```

Back-end :

```
const bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: true }))

app.post("/users", (req,res) => {
      console.log(req.body.user); // "Albert"
      res.status(200).end();
})
```

Pour en savoir plus

Ne pas hésiter à demander votre meilleur ami et à consulter la documentation

Travaux pratiques time!

Commençons à construire une petite **todo list**!

- Simplement afficher une liste de tâches hard-codées : ex. faire les courses, aller à la gym, nourrir le chien, etc.
- Front-end simple en jQuery et requête ajax (`$.ajax(...)` ou `$.get(...)`)
- Back-end : tableau hard-codé de tâches (let tasks = […….])

Feedback du jour 1

- Syntaxe plus simple pour lancer un serveur express

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

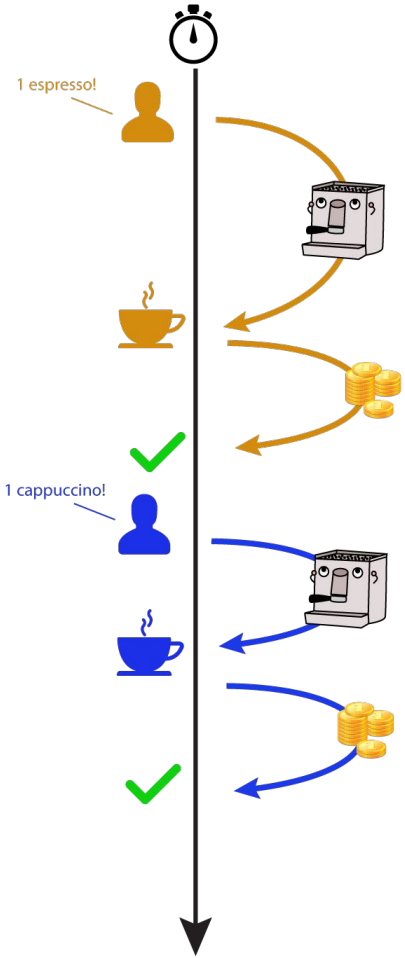
JOUR 2

- Async (gestion de multiples requêtes et callbacks)
- Manipulation de fichiers
- PM2 : pour garder le serveur actif en permanence

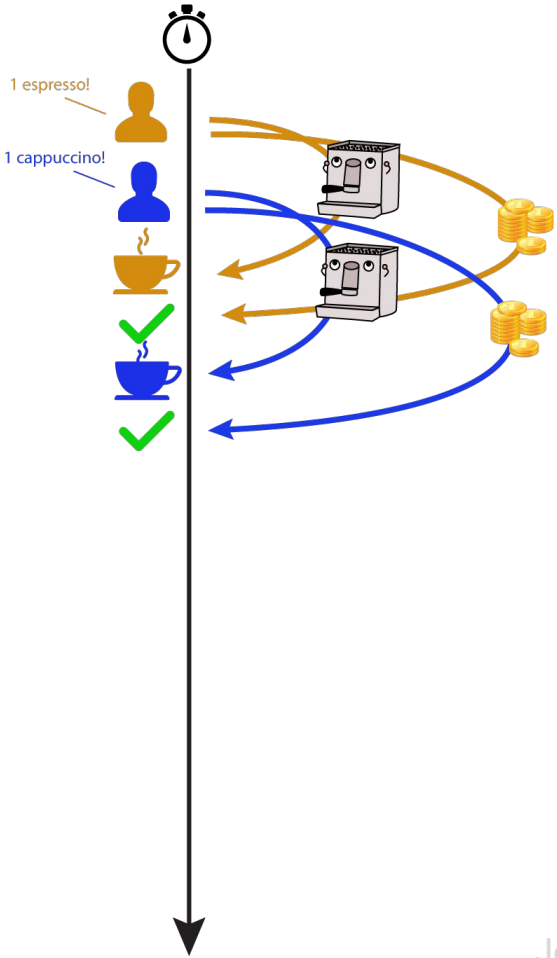
Multiples conditions requises pour terminer l'action:

- Café servi
- Commande payée

PHP



Node / Async



Sans librairie ni utilitaire :

```
setTimeout(() =>{
     console.log("Coffee done!")
},2000)

setTimeout(() =>{
     console.log("Payment done!")
},1000)

const finished = () => {
     console.log("All finished!")
}
```

(Code à compléter! Comment appeler la fonction finished?)

Callback = "Fonction à exécuter quand mon information est prête"

Avec la librairie Async :

```
async = require('async')

async.parallel([
    (cb) => {
          setTimeout(() =>{
               console.log("Coffee done!")
               cb()
          },2000)
     },
    (cb) => {
          setTimeout(() =>{
               console.log("Payment done!")
               cb()
          },1000)
     }
], (err, results) => {
    console.log("All done!")
});
```

Si la deuxième fonction nécessite d'attendre le résultat de la première :

```
fs = require('fs'); // fs = FileSystem

fs.stat( '1.jpg', (err, result) => {

     if(err) return console.log("err = " , err);

     console.log("Results = ",result);

     fs.writeFile('result.txt', JSON.stringify(result), (err) => {
          if(err) return console.log("err = " , err);

          console.log("~~~~~~~~ all done!")
     })

});
```

Problème : le callback hell !

```
func1( 'doSomething',  (result) => {
     func2( 'doSomething',  (result) => {
          func3(  'doSomething', (result) => {
               func4(  'doSomething', (result) => {
                    func5( 'doSomething',  (result) => {

                         console.log("All done in order!")

                    });
               });

          });
     });
});
```

Prermière solution au callback hell :

```
func1( 'doSomething', (result) => func2(result));

func2( 'doSomething', (result) => func3(result));

func3( 'doSomething', (result) => func4(result));

func4( 'doSomething', (result) => func5(result));

func5( 'doSomething', (result) => () => {console.log("All done in order!")});
```

Deuxième solution au callback hell : les Promises (et la librairie Q)

```
func1('doSomething')
     .then(func2('doSomething'))
     .then(func3('doSomething'))
     .then(func4('doSomething'))
     .then( (value) => {console.log("All done in order!")} )
     .catch( (err) => console.error(err) )
     .done();
```

Solution avec async.series :

```
async.series([
    (callback) => {
          setTimeout(() =>{
               console.log("Coffee done!")
               callback()
          },2000)
     },
    (callback) => {
          setTimeout(() =>{
               console.log("Payment done!")
               callback()
          },1000)
     },
    (callback) => {
          setTimeout(() =>{
               console.log("Added milk!")
               callback()
          },500)
     }

], (err, results) => {
    console.log("All done in order!")
});
```

Si l’on lance l’exemple précédent avec **async.parallel** au lieu d’**async.series**, quel sera l’ordre d’apparition?

Davantage sur la manipulation de fichiers

Voir la documentation du module fs

La persistance de votre serveur!

Utilisation de PM2

- Start
- Stop
- List
- Restart
- Logs
- Keymetrics

Travaux pratiques time!

Continuons la **todo list**

- Ajouter un champ <input> et un bouton “Add” pour ajouter des tâches
- Un autre champ <input> pour une recherche par mot-clé!

JOUR 3

- Tests unitaires avec Mocha et Jasmine
  - La Todo list doit renvoyer une liste de tous les documents
  - L’application doit supprimer une ligne
- MongoDB et Mongoose

https://github.com/jeremythille/simple_todolist_ng2.git

Réorganisation des fichiers pour l'API

modules/api.js

```
const add = (req,res) => {
     let todo= req.body.todo;
     todos.push(article);
     res.status(200).json(todo);
}

module.exports = {
     add
}
```

index.js

```
const api = require('./modules/api');

app.post('/todos', (req,res) =>
api.add(req,res))
```

Beaucoup de frameworks de tests existent : Mocha, Jasmine, Karma, Supertest, etc…

Nous allons écrire des tests unitaires avec Mocha et Jasmine.

Installation de Mocha : `npm i --save-dev mocha chai`

Chai est une librairie d'assertion ("expect") car Mocha n'en a pas à la base.

With the --production flag (`npm install --production`, or when the NODE_ENV environment variable is set to production), npm will not install modules listed in devDependencies.

- Créer un dossier `./test`

Mocha va automatiquement rechercher dans ce dossier (nom générique) et exécuter nos tests.

- Créer un fichier `./test/spec.js`
- Dans package.json, ajouter le script suivant :

```
"scripts" : {
     …
     "mocha-test" : "mocha"
     …
}
```

Pour exécuter le test, il suffit de faire : `npm run mocha-test`

```
const expect = require("chai").expect;

const api = require("../modules/api")

describe("TODO list", () => {


});
```

```
describe("TODO list", () => {

      const todo = {
            "created": new Date(),
            "text": "Some text",
            "title": "Some title"
      }

      it('should save a new todo', () => {

            const req = { body: { todo : todo } };

            const res = {
                  send: () => { },
                  json: (docReturned) => {
                        console.log("doc Returned : ", docReturned);
                        todo._id = docReturned._id;
                        expect(docReturned.text).to.equal(todo.text);
                        expect(docReturned.title).to.equal(todo.title);
                  },
                  status: (responseStatus) => {
                        expect(responseStatus).to.equal(200);
                        return res; // This line makes it chainable : res.status().json()
                  }
            }
            api.add(req, res);
      })
});
```

Exercice : écrire le test pour la suppression de tâche

Même tests avec Jasmine!

Installation de Jasmine : `npm i --save-dev jasmine-node`

- Créer un dossier `./test-jasmine`
- Créer un fichier `./test-jasmine/jasmine-server.spec.js` (.spec est obligatoire)

Les tests se lancent en exécutant la commande : `jasmine-node` *`dossier_de_specs`*

Ex : `jasmine-node test-jasmine`

Jasmine-node va scanner tout le dossier et exécuter les `.spec.js.`

```
const api = require("../modules/api")

let article = {
     "created": new Date(),
     "text": "Mocked text",
     "title": "Mocked title"
}

describe("Add", () => {

     // Test here

})
```

Avant l'assertion (`it('should do something')`), il faut exécuter l'appel à l' API (qui est asynchrone) puis tester le résultat.

```
beforeEach( () => {
     // Do something
});

it('should work' , () =>  {
     // Test what's been done in BeforeEach
})
```

Problème : `beforeEach` est asynchrone, donc il ne sera pas fini quand `it('should work')` sera exécuté. Donc?

Jasmine dispose d'un mode asynchrone qui permet d'attendre la fin d'une instruction avant de passer à la suite.

```
beforeEach( (done) => { // Informe Jasmine d'attendre la fin
     // Do something
     done();
});

it('should work' , () =>  {
     // Test what's been done in BeforeEach
})
```

```
let docToTest = null;

describe("Add", () => {
      beforeEach( (done) => {

            let req = { body: { todo: todo } };

            let res = {
                  json: (docReturned) => {
                        docToTest = docReturned;
                        done();
                  },
                  status : () => {
                        return res;
                  }
            }

            api.add(req, res);
      });


      it('should save a new todo', () => {
            expect(docToTest.text).toEqual(todo.text);
            expect(docToTest.title).toEqual(todo.title);
      }) // end it
})
```

Lancer le test :

```
> jasmine-node jasmine-test
.

Finished in 0.007 seconds
1 test, 2 assertions, 0 failures, 0 skipped
```

MongoDB! 

- Qu’est-ce que MongoDB?
- Pourquoi et quand utiliser MongoDB?
- Installation sur le système
- Exploration de la base avec RoboMongo
- Vocabulaire :
  - Database
  - Collection
  - Document

Lancement de MongoDB sur Mac :

./mongod --dbpath ../data/db&

Connection à MongoDB depuis Node

```
var MongoClient = require('mongodb').MongoClient;

MongoClient.connect("mongodb://localhost:27017/exampleDb", (err, db) => {
   if(err) return console.error(err);

   console.log("MongoDB is connected!")
});
```

Création / utilisation d'une collection (table) en particulier

```
var MongoClient = require('mongodb').MongoClient;

MongoClient.connect("mongodb://localhost:27017/exampleDb", (err, db) => {
      if(err) return console.error(err);
      console.log("MongoDB is connected!")

      db.createCollection('test',(err, collection) => {
            collection.insert( { 'doc1' : 'Hello!' } );
      });
});
```

Mongo : utilisation de la librairie Mongoose

```
const mongoose = require('mongoose');

const todoSchema = mongoose.Schema({
      title:{
            type : String,
            default : "Mon titre"
      },
      created: {
             type: Date,
            "default": Date.now
      }
});

const Todo = mongoose.model('todo', todoSchema); // Todo est un modèle Mongoose

mongoose.connect('mongodb://localhost/todolist', function(err) {
  if (err) {return console.error("Error connecting to MongoDB!");}
});
```

Mongoose donne ensuite accès à une syntaxe du type :

```
const todo = new Todo({
      title : "some Title",
      text : "Some text"
})

todo.save()


Todo.findOne( { title : "chien" }, (err,doc) => {    });

Todo.findOneAndRemove( { title : "chien" }, (err,doc) => {    });

Todo.findByIdAndUpdate( "654zf615z4r1f68ze", {new : true} , { text : "3 fois par jour" }
,(err,doc) => {    });
```

```
A.findOneAndUpdate(conditions, update, options, callback) // executes
A.findOneAndUpdate(conditions, update, options)  // returns Query
A.findOneAndUpdate(conditions, update, callback) // executes
A.findOneAndUpdate(conditions, update)           // returns Query
A.findOneAndUpdate()                             // returns Query
```

```
A.find(conditions, options).exec( (err,docs) => {  });
```

```
Todo.findOne(conditions)
.lean()
.exec( (err,doc) => {
     doc.title = "zjehf ozeh ozei";
     doc.save(); // ne fonctionne pas à cause de lean()
});
```

Syntaxe alternative :

```
Todo.findOne( { title : "chien" } )    // Retourne une requête
      .exec(err,doc) => {              // Exécute la requête
            doc.text = "3 fois par jour!";
            doc.save();
      });
```

Mode "Lean" :

```
Todo.findOne( { title : "chien" } )
      .lean() // Retourne un simple JSON
      .exec(err,doc) => {
            res.json(doc); // Ne peut pas doc.save()
      });
```

Options :

```
Todo.find( { title : "chien" },
           { limit:10,
           sort:{ created: -1 }
      })
      .exec(err,docs) => {
           res.status(200).json(docs);
      });
```

Syntaxe alternative :

```
Todo.find( { title : "chien" }),
      .limit(10)
      .sort('-created')
      .exec(err,docs) => {
           res.status(200).json(docs);
      });
```

Prochaine étape : remplacer la liste hard-codée de tâches par un appel réel à MongoDB!

api.add actuellement :

```
const add = (req,res) => {
      let todo = req.body.todo;
      todos.push(todo);
      res.status(200).json(todo);
}
```

A remplacer par :

```
const add = (req, res) => {

      const todo = new Todo(req.body.todo);

      todo.save( (err, docInserted) => {
            if (err) return res.status(500).json({ error: err })
            return res.status(200).json(docInserted);
      });
};
```

Feedback jour 3

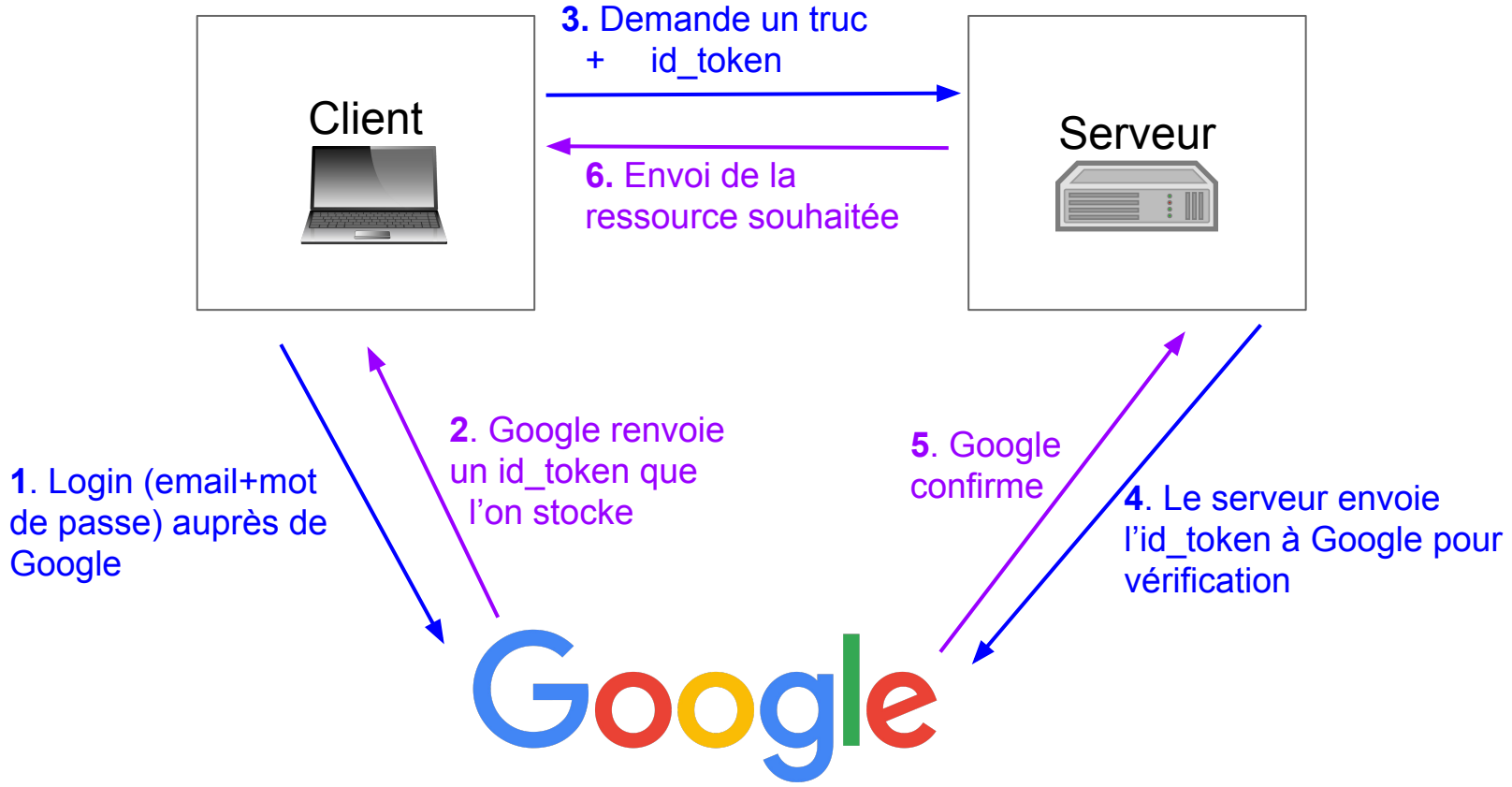
écriture plus simple de :

```
async.parallel([
      (cb) => {
            async.series([
                  makeCoffee,
                  addMilk
            ])
      },
      pay
], (err, results) => {
      console.log("All done in order!")
});
```

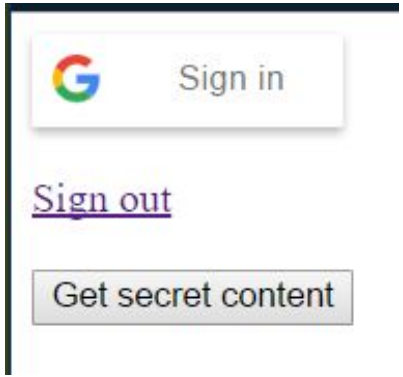
JOUR 4

- L’authentification
  - “Simple” bouton Google
  - PassportJS et les sessions et cookies
  - PassportJS et les JWT (JSON Web Token)
- Utilisation de middlewares

Authentification avec un “simple” bouton Google



Ce que nous allons construire en 10 étapes :



Étape 1 : index.html - inclusion de la librairie Google+ API

```
<meta name="google-signin-client_id" content="xxxxx.apps.googleusercontent.com">
<meta name="google-signin-scope" content="profile email">
<script src="https://apis.google.com/js/platform.js"></script>
```

Étape 2

Configurer un compte Google API : console.cloud.google.com pour obtenir le xxxxx

Étape 3 : Ajout du bouton :

```
<div id="signin"></div>
```

Étape 4 : génération du bouton (automatiquement, par la librairie Google)

```
<script>
      let id_token = null;

      gapi.signin2.render('signin', {
                  'scope': 'profile email',
                  'onsuccess': (googleUser) => {
                        token = googleUser.getAuthResponse().id_token;
                  },
                  'onfailure': (error) => console.log(error)
            });
</script>
```

Étape 5 : création d'un petit serveur (pour servir index.html)

```
const express = require('express')
const app = express();
const port = 8080;

app.use(express.static(__dirname))

app.listen(port, () => {
      console.log("App listening on port " + port)
})
```

Étape 6 :

Dans l’admin Google, autoriser :

- **`localhost:8080`** comme URL d’origine
- **`localhost:8080/auth/google/callback`** comme URL de redirection

(Nécessite quelques minutes pour se propager et prendre effet)

Tester le bouton de login dans notre application.

Si tout va bien, Google devrait renvoyer un objet contenant, entre autres, un id_token.

Étape 7 : envoyer le token_id au serveur pour authentification. On utilise jQuery pour ça

```
<button onClick="requestResource()">Get secret content</button>

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.js"></script>

requestResource = () => {
      $.get( '/secret, { id_token: id_token})
            .then((data) => {
                  console.log(data)
            })
            .catch((err) => console.error(err));
}
```

Étape 7 : Créer une route sur le serveur, qui reçoit le token

```
app.get('/secret, (req,res) => authenticate(req,res) )
```

Étape 8 : Créer la fonction qui va vérifier le token

```
const https = require('https');

const authenticate = (req, res) => {
      console.log(req.query.token);
      let rawData = '';

      https.get('https://www.googleapis.com/oauth2/v3/tokeninfo?id_token=' + req.query.token, (stream) => {
            stream
                  .on('data' , (chunk) => rawData += chunk )
                  .on('error', (e) =>  log("Got error: " + e.message) )
                  .on('end'  , () => {
                        let data = JSON.parse(rawData)
                        // ... à finir
                  });
      });
}
```

Étape 9 :

En réponse au token, Google renvoie **un objet contenant les détails de l'utilisateur** (suivant le scope autorisé).

Par exemple, ayant l'adresse email, on peut vérifier si cet utilisateur est autorisé à accéder à la ressource demandée.

Sur le serveur :

```
const admins = [ 'jeremy.thille@gmail.com' ]

stream.on('end'  , () => {
      let data = JSON.parse(rawData)
      if (admins.indexOf(data.email) > -1) {
            res.status(200).send('Authorised! :)');
      } else {
            res.status(403).send('Unauthorized :(');
      }
});
```

Étape 10 : Un petit bouton/lien pour se déconnecter

HTML :

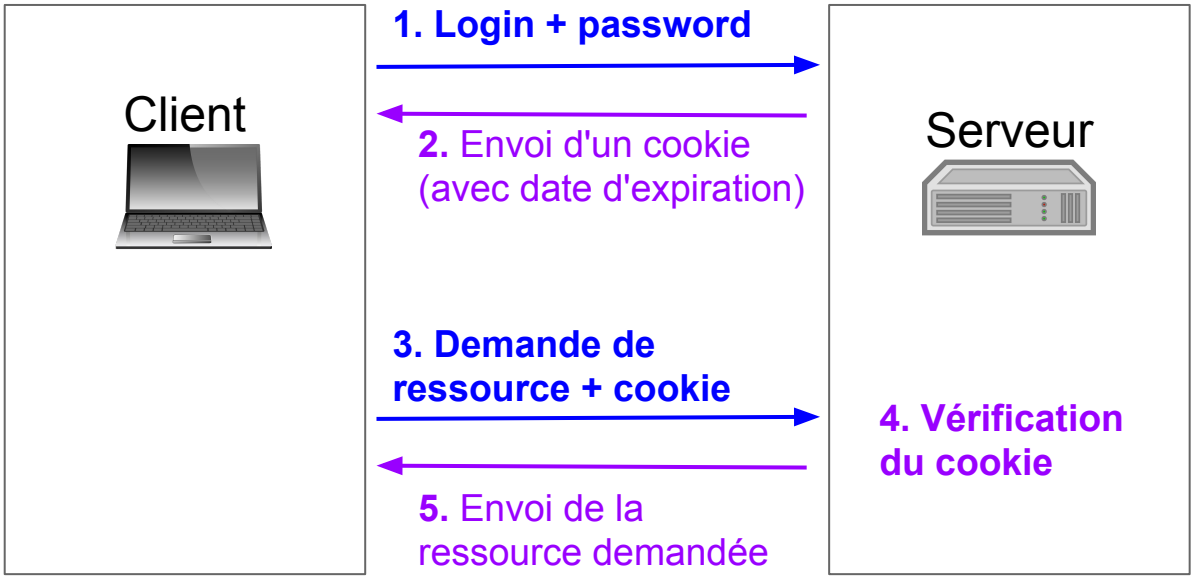
```
<a href="#" onclick="signOut();">Sign out</a>
```

JS :

```
signOut = () => {
      const auth2 = gapi.auth2.getAuthInstance();
      auth2.signOut().then( () => {
            id_token = null;
            console.log('User signed out.');
      });
}
```

PM2 => Clustering et load balancer

Utilisation de PassportJS et des Sessions



Cookies étape 1 : (Node) inclusion des modules requis et configuration

```
const bodyParser = require('body-parser');
const session = require('express-session');

const passport = require('passport'),
      LocalStrategy = require('passport-local').Strategy;


app
      .use(express.static(__dirname))
      .use(bodyParser.urlencoded({ extended: false })) // <----- Required for Passport (but
not mentioned in the docs)
      .use(session({  // Enabling Express sessions via browser cookie
            secret: '1234567890QWERTY'
      }))
      .use(passport.initialize()) // <---------------- Don't forget this!!
      .use(passport.session()) // Place this AFTER express.use(session())
```

Étape 2 : formulaire de login

```
<form action="/login" method="post">
      <div>
            <label>Username:</label>
            <input type="text" name="username" value="jer" />
      </div>
      <div>
            <label>Password:</label>
            <input type="password" name="password" value="toto"/>
      </div>
      <div>
            <input type="submit" value="Submit" />
      </div>
</form>
```

Étape 3 : réception des identifiants sur le serveur et utilisation de la stratégie 'local' de Passport

```
app.post('/login', passport.authenticate('local', {
      successRedirect: '/secret',
      failureRedirect: '/badLogin'
}));
```

Étape 4 : définition de la stratégie locale de Passport

```
passport.use(new LocalStrategy( // Reminder : LocalStrategy = require('passport-local').Strategy
      (username, password, done) => {
            let user = User.find(username, password);
            done(false, user || false); // 1st argument = error, 2nd argument = user found?
      }
));
```

Étape 5 : quand une ressource (page HTML…) est demandée, utilisation d'un middleware d'authentification

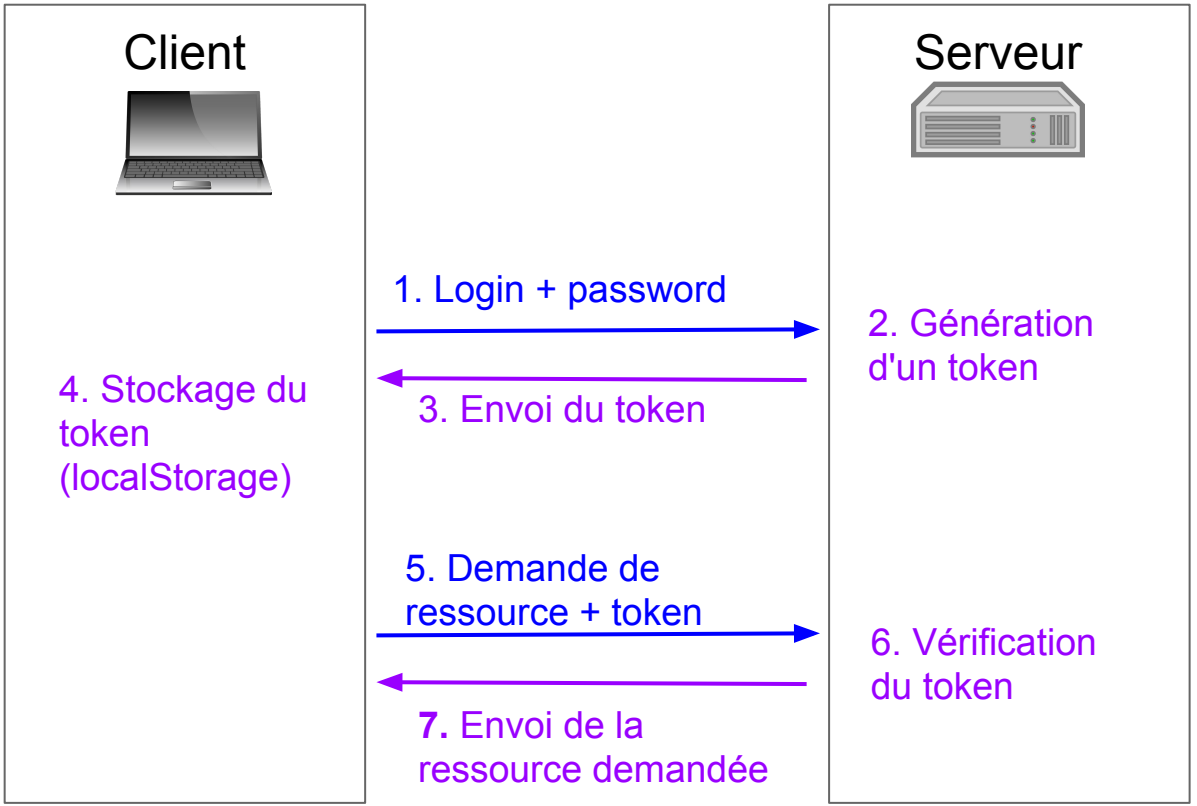
```
.get('/secret',
            passport.authenticationMiddleware(),
            (req, res) => {
                  console.log("Should not see me unless logged in")
                  res.sendFile(__dirname + '/secret.html')
            })
```

Étape 5 : définition du middleware d'authentification

```
passport.authenticationMiddleware = () => {
      return (req, res, next) => {
            if (req.isAuthenticated()) {
                  console.info("Auth middleware : authorised :)")
                  return next()
            }
            console.error("Auth middleware : not auth!")
            res.redirect('/login')
      }
}
```

`req.isAuthenticated()` va vérifier le cookie qui a automatiquement été envoyé avec le header

Utilisation de PassportJS et des JWT (JSON Web Token)



JWT étape 1 : (Node) inclusion des modules requis et configuration

```
const bodyParser = require('body-parser');
const jwt = require('jwt-simple');

const passport = require('passport'),
      LocalStrategy = require('passport-local').Strategy;



app
      .use(express.static(__dirname))
      .use(bodyParser.urlencoded({ extended: false }))
      .use(passport.initialize())
```

Étape 2 : création du bouton de login

```
<script>
      $('#login').click(() => {

            const username = $('input[name=username]').val();
            const password = $('input[name=password]').val();

            $.post("/token", {
                        username: username,
                        password: password
                  },
                  "json")
                  .success((data) => {
                        localStorage.setItem('token', data.token); // étape 5
                        window.location = "/secret"
                  })
                  .error((err) => { alert(err.statusText); })
      });
</script>
```

Étape 3 : création de la route où l'on va générer le token

```
app.post('/token',
      passport.authenticate('local', { // Same thing as with sessions
            session: false               // ...except session : false
      }), // First, authenticates with username and password. If successful, creates a token
      (req, res) => {

            // If this function gets called, authentication was successful.
            // `req.user` contains the authenticated user, without his token yet.

            console.log("Login with password successful! Now creating token")

            createToken(req.user.id, (token) => {
                  res.status(200).json({
                        token: token
                  });
            });
      });
```

Étape 4 : création de la fonction qui va générer le token

```
createToken = (id, cb) => {

      let user = User.findById(id);

      let token = jwt.encode({
            id: id
      }, tokenSecret); // const tokenSecret = 'put-a-$Ecr3t-h3re';

      // Updating user
      for (let i in users) {
            if (users[i].id === id) {
                  users[i].token = token; //Create a token and add to user and save
                  break; //Stop this loop, we found it!
            }
      }

      cb(token);

};
```

Étape 5 : stocker le token dans le LocalStorage du navigateur (voir callback dans étape 2)

Étape 6 : faire une requête AJAX en envoyant le token

```
$.ajax({
      url: "/secretContent",
      type: 'GET',
      headers: { "token": localStorage.getItem('token') }
})
.done((data) => {
      $('div#content').html(data);
})
```

Étape 7 : création de la route app.get('/secret')

```
      .get('/secretContent',
                  passport.authenticationMiddleware,
                  (req, res) => {
                        res.status(200).send('Secret image!<br><img
src="http://www.minionland.com/wp-content/uploads/2015/10/minion-stuart-dance-gif-1445349814n84gk.gif"
/><br><br>')
                  })
```

Étape 8 : Middleware d'authentification

```
passport.authenticationMiddleware = (req, res, next) => {

      console.log("req.headers.token = ", req.headers.token)

      let decoded;
      try {
            decoded = jwt.decode(req.headers.token, tokenSecret);
      } catch (e) {
            console.error("Auth middleware : Error decoding token. ~~~~~~~~~~")
            return res.redirect('/login')
      }
      console.log("decoded = ", decoded)
      user = User.findById(decoded.id);

      if (user) {
            console.info("Auth middleware : authorised :)  ~~~~~~~~")
            return next()
      }

      console.error("Auth middleware : not authorised! ~~~~~~~~~~")
      res.redirect('/login')
}
```

Upload de fichiers avec Multer

Configuration :

```
const multer = require('multer');
const storage = multer.diskStorage({
      destination : (request, file, callback) => {
            callback(null, './uploads');
      },
      filename: (request, file, callback) => {
            console.log("file" , file);
            callback(null, file.originalname)
      }
});
const upload = multer({ storage: storage })
```

Envoi d'un seul fichier :

```
<h3>Avatar :</h3>
<form enctype="multipart/form-data" action="/avatar" method="post">
    <input type="file" name="avatar" />
    <input type="submit" value="Upload Image" name="submit" />
</form>
```

```
app.post('/avatar', (request, response) => {
    upload.single('avatar')(request, response, (err) => { // 'avatar' MUST match <input type="file" name="avatar" />
        if (err) return response.send(err);
        console.log(request.file);
        response.end('Your avatar uploaded');
    })
});
```

Envoi de fichiers multiples :

```
<form enctype="multipart/form-data" action="/photos" method="post">
    <input type="file" name="photos" multiple/>
    <input type="submit" value="Upload Image" name="submit" />
</form>
```

```
app.post('/photos', (request, response) => {
    upload.array('photos')(request, response, (err) => { // 'photos' MUST match <input type="file" name="photos" />
        if (err) return response.send(err);
        console.log(request.file);
        response.end('Your photos Uploaded');
    })
});
```

Bonus : SocketIO (si on a le temps)