Node.js Précis et concis

Table des matières

Installation

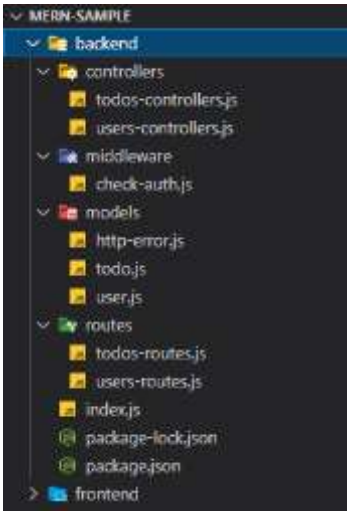
Téléchargement et installation de la version recommandée (LTS) https://nodejs.org/en/

Résumé

On peut créer :

- un dossier contenant tout le code
- ou pour une application MERN avoir un dossier « backend » (ou « api ») et un dossier « frontend » (ou « client ») pour le code react. On alors un « package.json » par « dossier ». Astuce : on peut split le terminal de VS Code et naviguer sur chaque dossier pour gérer chaque projet.

Exemple de structure



Le plus souvent Node sera utilisé pour faire un server REST.

On a :

- « package.json » avec les dépendances, dépendances de développement, scripts
- index.js
- Utilisation d'express pour le routing, body parsing
- On peut avoir besoin de définir les cors ou utiliser le plugin
- un dossier « **routes** » avec les routes (« users-routes.js » par ex)
- un dossier « **controllers** » avec les controllers utilisés par les routes (« users-controllers.js » par ex)
- On peut avoir en plus un dossier middleware avec des middlewares utilisés (protection de route par ex)
- On peut utiliser MongoDB + driver seul ou avec mongoose
- Avec mongoose on a un dossier « models » avec les schémas (« user.js » par ex)

package.json

Création de « package.json » : depuis le terminal de VS Code (ou une invite de commande)

```
npm init
```

Dépendances

```
npm i express
```

Dépendances de développement

```
npm i nodemon -D
```

Nodemon : permet le hot reloading, pas besoin de relancer le serveur pour valider et voir les modifications dans le code.

MongoDB … avec Mongoose

```
npm i mongoose
```

Sinon installation du driver

```
npm i mongodb
```

Autres dépendances utiles

« bcryptjs » encryptage de password

npm

Installation

```
npm i bcryptjs
```

Exemple dans un controller « users-controllers »

```
const bcrypt = require("bcryptjs");
```

Lors du signup, on encrypte le password reçu pour le stocker dans la base

```
let hashedPassword = await bcrypt.hash(password, 12);
```

Lors du login, on vérifie que le password reçu correspond bien à celui enregistré

```
let isValidPassword = await bcrypt.compare(password, existingUser.password);
if (!isValidPassword) {
    const error = new HttpError(
      "Invalid credentials, could not log you in.",
      403
    );
    return next(error);
}
```

« jsonwebtoken » création et vérification de tokens

npm

Installation

```
npm i jsonwebtoken
```

Exemple dans un controller « users-controllers »

```
const jwt = require("jsonwebtoken");
```

Création d'un token à la fin du login

```
  let token = jwt.sign(
```

```
    { userId: existingUser.id, email: existingUser.email },
    "secret_key",
    { expiresIn: "1h" }
  );
```

Verification du token… soit dans un try catch soit avec callback

Exemple avec try catch (création d'un middleware (« check-auth.js » par ex dans un dossier « middleware »)

```
const jwt = require("jsonwebtoken");

const HttpError = require("../models/http-error");

module.exports = (req, res, next) => {
  if (req.method === "OPTIONS") return next();

  try {
    const token = req.headers.authorization.split(" ")[1]; // Authorization: 'Bearer TOKEN'
    if (!token) throw new Error("Authentication failed!");

    const decodedToken = jwt.verify(token, "secret_key");

    req.userData = { userId: decodedToken.userId };
    next();
  } catch (err) {
    const error = new HttpError("Authentication failed!", 403);
    return next(error);
  }
};
```

… avec callback

```
 jwt.verify(token, 'secret_key', (err, decodedToken) => {
            if (err) req.user = undefined;
            req.user = decodedToken;
            next();
        });
```

Utilisation du middleware pour protéger certaines routes

```
const checkAuth = require('../middleware/check-auth');
```

```
router.use(checkAuth);
```

« express-validator » validation de formulaire

npm

```
npm i express-validator
```

Sur une **route** (« users-routes.js » par ex), on définit les **règles de validation**. Le body passé sera donc évalué

```
const { check } = require("express-validator");
```

```
router.post(
  "/signup",
  [
```

```
    check("name").not().isEmpty(),
    check("email").normalizeEmail().isEmail(),
    check("password").isLength({ min: 6 }),
  ],
  usersController.signup
);
```

Dans le **controller** (« users-controllers.js » par ex), on **vérifie** au tout début qu'il y a des **errors**

```
const signup = async (req, res, next) => {
  const errors = validationResult(req);
  console.log(errors);
  if (!errors.isEmpty()) {
    return next(
      new HttpError("Invalid inputs passed, please check your data.", 422)
    );
  }
  // etc.
```

multer

npm

Installation

```
npm install multer
```

upload de fichier + body parsing de formData . voir

« cors »

npm

Permet d'éviter d'avoir à définir les cors soi-même côté server

Installation

```
npm install cors
```

Dans « index.js »

```
const cors = require("cors");
```

```
app.use(cors());
```

Ajout de scripts

Au plus simple on ajoute le script « start » pour démarrer le server

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index.js"
  },
```

« node index.js » sir nodemon n'est pas installé

Pour lancer le serveur (depuis le terminal de VS Code ou une invite de comande)

```
npm start
```

index.js

Exemple typique

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const HttpError = require("./models/http-error");
const usersRoutes = require("./routes/users-routes");
const todosRoutes = require("./routes/todos-routes");

const PORT = process.env.PORT || 5000;

const app = express();

app.use(express.json()); // for parsing application/json

// cors
app.use(cors());

// routes
app.use("/api/users", usersRoutes);
app.use("/api/todos", todosRoutes);

// pas de route correspondante trouvée
app.use((req, res, next) => {
  const error = new HttpError("Route not found.", 404);
  throw error;
});

// interception des errors retournées avec next dans les middlewares précédents
app.use((error, req, res, next) => {
  if (res.headerSent) {
    return next(error);
  }
  res.status(error.code || 500);
  res.json({ message: error.message || 'An unknown error occurred!' });
});

// mongoose
const url = "mongodb://jerome:secret@127.0.0.1:27017/sample?authSource=admin";
mongoose.connect(url, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

app.listen(PORT, () => {
  console.log(`Backend server is running on port ${PORT}!`);
});
```

Body parser

Avant il fallait installer le package « body-parser ». Désormais il est compris avec Express

```
const express = require("express");
const multer = require("multer");

const forms = multer();
const app = express();

app.use(express.json()); // for parsing application/json
app.use(forms.array()); // formData
```

```
app.use(express.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded
```

Requêtes (client)

Json

Ajouter en header le « Content-Type » : « application/json », passer en body le json

```
fetch("http://localhost:5000/api/users/login", {
   method: "POST",
   headers: {
     "Content-Type": "application/json",
   },
   body: JSON.stringify({
     email: "test@email.com",
     password: "test12345",
   }),
 })
   .then(function (response) {
     return response.json();
   })
   .then(function (responseData) {
     console.log(responseData);
   });
```

FormData

passer le formData en paramètre du body. Pas besoin de « Content-Type » en header normalement.

```
const formData = new FormData();
formData.append("email", "test@email.com");
formData.append("password", "test12345");
```

Ou avec les infos d'un formulaire (récupérer le formulaire au chargement et faire « event.preventDefault » au « submit »). Form peut également être récupérer avec « event.target »

```
const formData = new FormData(form);
```

Puis

```
fetch("http://localhost:5000/api/users/login", {
   method: "POST",
   body: formData,
 })
   .then(function (response) {
     return response.json();
   })
   .then(function (responseData) {
     console.log(responseData);
   });
```

Form (application/x-www-form-urlencoded)

Plutôt à éviter

```
<form id="userForm" action="http://localhost:5000/api/users/login" method="POST" enctype="application/x-www-form-urlencoded">
    <input type="email" name="email">
    <input type="password" name="password">
    <button type="submit">Send</button>
</form>
```

Cors

Cors : système de sécurité qui bloque les requêtes HTTP effectuées entre des serveurs différents (exemple : « localhost :3000 » et « localhost :5000 »). On peut le définir soi-même

```
app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Headers','Origin, X-Requested-With, Content-
Type, Accept, Authorization');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PATCH, DELETE');
  next();
});
```

Ou utiliser le package « cors » voir

Mongoose

Schéma

Création d'un schema. Exemple « user.js » dans un dossier « models »

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userSchema = new Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true, minlength: 6 },
  created_date: { type: Date, default: Date.now },
});

module.exports = mongoose.model("User", userSchema);
```

Schémas avec relations

Exemple: Todo List pour chaque user connecté. Un « todo » a un « user » (nommé ici « creator »)

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const todoSchema = new Schema({
  message: { type: String, required: true },
  completed: { type: Boolean, default: false },
  created_date: { type: Date, default: Date.now },
  creator: { type: mongoose.Types.ObjectId, required: true, ref: "User" },
});

module.exports = mongoose.model("Todo", todoSchema);
```

Un « user » a un array de « todos » (en fait des _id des todos)

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userSchema = new Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true, minlength: 6 },
  created_date: { type: Date, default: Date.now },
```

```
  todos: [{ type: mongoose.Types.ObjectId, required: true, ref: 'Todo' }]
});

module.exports = mongoose.model("User", userSchema);
```

On peut utiliser la méthode "**populate**" pour avoir l'ensemble des informations de l'élément en relation et pas seulement l'_id. Exemple

```
let todo = await Todo.findById(todoId).populate("creator");
```

```
{
  completed: false,
  _id: 610d3fd10e183e2f4870f7c2,
  message: 'todo 2',
  creator: {
    todos: [ 610d3fc80e183e2f4870f7bd, 610d3fd10e183e2f4870f7c2 ],
    _id: 610d3f510e183e2f4870f7ba,
    name: 'romagny13',
    email: 'romagny13@yahoo.fr',
    password: '$2a$12$EPXv6HR7G6..722Oj.goYODM/RTrF0i.NQjcC9UxGxcrnjcUwH11S',
    created_date: 2021-08-06T13:55:29.262Z,
    __v: 2
  },
  created_date: 2021-08-06T13:57:37.321Z,
  __v: 0
}
```

Connexion

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

…

```
const url = "mongodb://jerome:secret@127.0.0.1:27017/sample?authSource=admin";
mongoose.connect(url, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
```

Une variante: connexion à MongoDB puis server Node.js

```
mongoose
  .connect("mongodb://jerome:secret@127.0.0.1:27017/sample?authSource=admin")
  .then(() => {
    app.listen(5000);
  })
  .catch((err) => {
    console.log(err);
  });
```

Routes

Besoin de mongoose, du controller, du router d'Express. Définition des routes et export du router.

Attention à l'ordre des routes, une route peut être exécutée avant une autre et que la route désirée ne soit pas appelée.

Exemple « users-routes.js »

```
const express = require("express");
const { check } = require("express-validator");
const usersController = require("../controllers/users-controllers");

const router = express.Router();
```

```
router.post(
  "/signup",
  [
    check("name").not().isEmpty(),
    check("email").normalizeEmail().isEmail(),
    check("password").isLength({ min: 6 }),
  ],
  usersController.signup
);

router.post("/login", usersController.login);

module.exports = router;
```

Route avec middleware, ici pour valider les infos passées

Autre exemple « todos-routes.js »

```
const express = require("express");
const {check} = require("express-validator");
const checkAuth = require("../middleware/check-auth");
const todosControllers = require("../controllers/todos-controllers");

const router = express.Router();

router.use(checkAuth);

// api/todos/123abc
router.get("/:uid", todosControllers.getTodosByUserId);

// api/todos
router.post("/", [check("message").not().isEmpty()], todosControllers.createTo
do);

// api/todos/completetodo//xyz321
router.put("/completetodo/:pid", todosControllers.setCompleted);

// api/todos/xyz321
router.delete("/:pid", todosControllers.deleteTodo);

module.exports = router;
```

Protection des routes avec middleware, seuls les users authentifiés peuvent y accéder

Controller

C'est avant tout des functions + exports de celles-ci.

Les functions récupèrent les infos depuis « req.body » grâce au body parser et retournent une response au format json avec un status (200 par défault) ou une error.

Exemple de controller « users-controllers.js »

```
const { validationResult } = require("express-validator");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const HttpError = require("../models/http-error");
const User = require("../models/user");

const signup = async (req, res, next) => {

  // validation
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return next(
      new HttpError("Invalid inputs passed, please check your data.", 422)
    );
  }

  const { name, email, password } = req.body;

  // check if a user with this email exists
  let existingUser = await User.findOne({ email: email });
  if (existingUser) {
    const error = new HttpError(
      "User exists already, please login instead.",
      422
    );
    return next(error);
  }

  // hash password
  let hashedPassword = await bcrypt.hash(password, 12);

  // save new user
  const newUser = new User({
    name,
    email,
    password: hashedPassword,
  });
  await newUser.save();

  // create token
  let token = jwt.sign(
    { userId: newUser.id, email: newUser.email },
    "secret_key",
    { expiresIn: "1h" }
  );

  // sand response user created + token
  res
    .status(201)
    .json({ userId: newUser.id, email: newUser.email, token: token });
};

const login = async (req, res, next) => {
  const { email, password } = req.body;

  // find the user by email
  let existingUser = await User.findOne({ email: email });
  if (!existingUser) {
    const error = new HttpError(
      "Invalid credentials, could not log you in.",
      403
    );
    return next(error);
```

```
  }

  // check password
  let isValidPassword = await bcrypt.compare(password, existingUser.password);
  if (!isValidPassword) {
    const error = new HttpError(
      "Invalid credentials, could not log you in.",
      403
    );
    return next(error);
  }

  // create token
  let token = jwt.sign(
    { userId: existingUser.id, email: existingUser.email },
    "secret_key",
    { expiresIn: "1h" }
  );

  // send user id, email and token
  res.json({
    userId: existingUser.id,
    email: existingUser.email,
    token: token,
  });
};

exports.signup = signup;
exports.login = login;
```

CRUD

Get by user id

```
const getTodosByUserId = async (req, res, next) => {
  const userId = req.params.uid;

  let todos;
  try {
    todos = await Todo.find({ creator: userId });
  } catch (err) {
    const error = new HttpError(
      "Fetching todos failed, please try again later.",
      500
    );
    return next(error);
  }
  res.json({ todos: todos });
};
```

Status 200 + object avec « todos » contenant l'array

Exemple de requête (client):

```
var data = null;

var xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});

xhr.open("GET", "http://localhost:5000/api/todos/610d3f510e183e2f4870f7ba");
xhr.setRequestHeader(
  "authorization",
  "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2MTBkM2Y1MTBlMTgzZTJmNDg3MGY3YmEiLCJlbWFpbCI6InJvbWFnbnkxM0B5YWhvby5mciIsImlhdCI6MTYyODI1ODEyOSwiZXhwIjoxNjI4MjYxNzI5fQ.FV6hZ1xdmIjZw-pCC5YximNwWCPrGOystKIIpuhWhnw"
);

xhr.send(data);
```

userId

token

Create

Avec transaction

```
const createTodo = async (req, res, next) => {
  const userId = req.userData.userId;
  const { message } = req.body;

  // user
  let user;
  try {
    user = await User.findById(userId);
  } catch (err) {
    const error = new HttpError("Creating todo failed, please try again.", 500);
    return next(error);
  }

  if (!user) {
    const error = new HttpError("Could not find user for provided id.", 404);
    return next(error);
  }

  // todo
  const createdTodo = new Todo({
    message,
    creator: userId,
  });

  try {
    const session = await mongoose.startSession();
    session.startTransaction();
    await createdTodo.save({ session: session });
    user.todos.push(createdTodo);
    await user.save({ session: session });
    await session.commitTransaction();
  } catch (err) {
    console.log(err);
    const error = new HttpError("Creating todo failed, please try again.", 500);
    return next(error);
  }

  res.status(201).json({ todo: createdTodo });
};
```

Status 201 (created) + todo créée

Exemple de requête (client)

```
var data = null;

var xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});

xhr.open("GET", "http://localhost:5000/api/todos/610d3f510e183e2f4870f7ba");
xhr.setRequestHeader(
  "authorization",
  "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2MTBkM2Y1MTBlMTgzZTJmNDg3MGY3YmEiLCJlbWFpbCI6InJvbWFnbnkxM0B5YWhvby5mciIsImlhdCI6MTYyODI1ODEyOSwiZXhwIjoxNjI4MjYxNzI5fQ.FV6hZ1xdmIjZw-pCC5YximNwWCPrGOystKIIpuhWhnw"
);

xhr.send(data);
```

todoId

token

Update

```
const setCompleted = async (req, res, next) => {
  const userId = req.userData.userId;
  const todoId = req.params.pid;

  let todo;
  try {
    todo = await Todo.findById(todoId).populate("creator");
  } catch (err) {
    const error = new HttpError(
      "Something went wrong, could not delete todo.",
      500
    );
    return next(error);
  }

  if (!todo) {
    const error = new HttpError("Could not find todo for this id.", 404);
    return next(error);
  }

  if (todo.creator.id !== userId) {
    const error = new HttpError(
      "You are not allowed to delete this todo.",
      401
    );
    return next(error);
  }

  todo.completed = true;

  try {
    await todo.save();
  } catch (err) {
    const error = new HttpError(
      "Something went wrong, could not update place.",
      500
    );
    return next(error);
  }

  res.status(200).json({ todo: todo.toObject({ getters: true }) });
};
```

Exemple de requête (client)

```
var data = null;

var xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});

xhr.open(
  "PUT",
  "http://localhost:5000/api/todos/completetodo/610d3fd10e183e2f4870f7c2"
);
xhr.setRequestHeader(
  "authorization",
  "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2MTBkM2Y1MTBlMTgzZTJmNDg3MGY3YmEiLCJlbWFpbCI6InJvbWFnbnkxM0B5YWhvby5mciIsImlhdCI6MTYyODI1ODEyOSwiZXhwIjoxNjI4MjYxNzI5fQ.FV6hZ1xdmIjZw-pCC5YximNwWCPrGOystKIIpuhWhnw"
);

xhr.send(data);
```

Delete

```
const deleteTodo = async (req, res, next) => {
  const userId = req.userData.userId;
  const todoId = req.params.pid;

  // todo
  let todo;
  try {
    todo = await Todo.findById(todoId).populate("creator");
    console.log(todo);
  } catch (err) {
    const error = new HttpError(
      "Something went wrong, could not delete todo.",
      500
    );
    return next(error);
  }

  if (!todo) {
    const error = new HttpError("Could not find todo for this id.", 404);
    return next(error);
  }

  if (todo.creator.id !== userId) {
    const error = new HttpError(
      "You are not allowed to delete this todo.",
      401
    );
    return next(error);
  }

  try {
    const session = await mongoose.startSession();
    session.startTransaction();
    await todo.remove({ session: session });
    todo.creator.todos.pull(todo);
    await todo.creator.save({ session: session });
    await session.commitTransaction();
  } catch (err) {
    const error = new HttpError(
      "Something went wrong, could not delete todo.",
      500
    );
    return next(error);
  }

  res.status(200).json({ message: "Deleted todo." });
};
```

Exemple de requête (client)

```
var data = null;

var xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.addEventListener("readystatechange", function () {
  if (this.readyState === 4) {
    console.log(this.responseText);
  }
});

xhr.open("DELETE", "http://localhost:5000/api/todos/610d3fd10e183e2f4870f7c2");
xhr.setRequestHeader(
  "authorization",
  "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2MTBkM2Y1MTBlMTgzZTJmNDg3MGY3YmEiLCJlbWFpbCI6InJvbWFnbnkxM0B5YWhvby5mciIsImlhdCI6MTYyODI1ODEyOSwiZXhwIjoxNjI4MjYxNzI5fQ.FV6hZ1xdmIjZw-pCC5YximNwWCPrGOystKIIpuhWhnw"
);

xhr.send(data);
```

Sécurité

Processus :

- Inscription de l'utilisateur.
- Utilisateur se connecte et reçoit un token (jwt) que l'on stocke dans le localStorage du browser. Le token contient les informations (userId et email)
- Lors des requêtes sur des routes protégées on passe le token en header (Authorization: 'Bearer TOKEN').
- Côté server, on vérifie le token pour l'accès à la ressource, sinon on retourne une error.

Avec passport

Passport semble à utiliser seulement si on a des vues côté serveur et moins adapter pour une API

Exemple

```
const express = require("express");
const mongoose = require("mongoose");
const path = require("path");
const User = require("./models/user");
const PORT = 3000;

const app = express();

app.use(express.static("public"));

// afficher une page index.html (contient un lien <a href="http://localhost:3000/auth/google">Google</a
>)
// app.get("/", (req, res) => {
//   res.sendFile(path.join(__dirname, "public", "index.html"));
// });

const GOOGLE_CLIENT_ID =
  "363153640746-td34482goq9he3l009dom4dmqt9qpvvn.apps.googleusercontent.com";
const GOOGLE_CLIENT_SECRET = "hu_Bib4xBgZDp-mlj_AyFi7l";
const GOOGLE_CALLBACK_URL = "http://localhost:3000/auth/google/redirect";

// const jwt = require("jsonwebtoken");

// STRATEGIES
const passport = require("passport");
const GoogleStrategy = require("passport-google-oauth20").Strategy;
passport.use(
  new GoogleStrategy(
    {
      callbackURL: GOOGLE_CALLBACK_URL,
      clientID: GOOGLE_CLIENT_ID,
      clientSecret: GOOGLE_CLIENT_SECRET,
      //scope: ["profile"],
    },
    async (accessToken, refreshToken, profile, cb) => {
      //   console.log("PROFILE", profile);
      const email = profile.emails[0].value;
      const user = await User.findOne({ email: email });
      if (!user) {
        const providerData = profile._json;
        providerData.accessToken = accessToken;
        providerData.refreshToken = refreshToken;

        const user = new User({
          name: profile.displayName,
          email: profile.emails[0].value,
          profileImageURL: providerData.picture
            ? providerData.picture
```

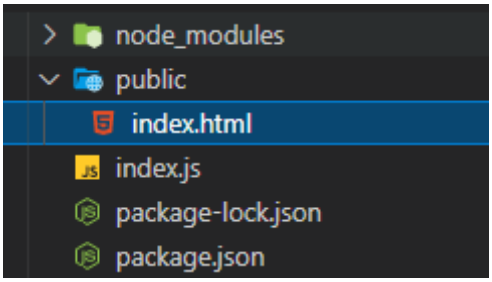
création d'un projet https://console.developers.google.com/ :

- ajout de API Google+
- identifiants "Client ID OAuth"
- …puis ajout uri et redirect uri

```
            : undefined,
          google: providerData,
        });
        console.log(user);
        await user.save();
      }
      return cb(null, profile);
    }
  )
);

// routes
app.get(
  "/auth/google",
  passport.authenticate("google", {
    scope: ["profile", "email"],
    session: false,
  })
);
app.get(
  "/auth/google/redirect",
  passport.authenticate("google", {
    session: false,
    //  failureRedirect: `https://localhost:3000/login`,
  }),
  (req, res) => {
    // console.log("redirect", req.user);
    res.redirect("/");
  }
);

// serializers
passport.serializeUser(function (user, cb) {
  cb(null, user);
});

passport.deserializeUser(function (obj, cb) {
  cb(null, obj);
});

// initialization
app.use(passport.initialize());

const url = "mongodb://jerome:secret@127.0.0.1:27017/sample?authSource=admin";
mongoose.connect(url, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

app.listen(PORT, () => {
  console.log("Backend server is running");
});
```

On pourrait ajouter d'autres stratégies (local, facebook, etc.)

Autres routes selon stratégies (local, facebook, etc.)



La page d'index ne contient qu'un lien dans le body

```
<a href="http://localhost:3000/auth/google">Google</a>
```

Exemple de schéma

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const userSchema = new Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String },
  profileImageURL: { type: String },
  roles: {
    type: [
      {
        type: String,
        enum: ["user", "admin"],
      },
    ],
    default: ["user"],
  },
  google: { type: Object },
  created: { type: Date, default: Date.now },
});

module.exports = mongoose.model("User", userSchema);
```

En résumé, on clique sur le lien, on est redirigé vers la page de consentements de Google, puis google appelle l'url de redirection fournie ce qui permet de récupérer des informations telles que le profil (défini dans le scope). Enfin on se contente de rediriger vers la page d'accueil du site.

Ressources

- Passport
- passport-google-oauth20
- Exemple avec Google

Passport jwt (https://github.com/mikenicholson/passport-jwt) peut être utilisé pour vérifier automatiquement les tokens jwt pour une API ou site node.js.

Socket.io

Installation

```
npm i socket.io
```

Chat : serveur et page index dans public

Création d'un serveur

```
const express = require("express");
const app = express();
const server = require("http").Server(app);
const io = require("socket.io")(server);
const port = 3000;

// attention c'est le server retourné avec http, pas l'app d'express
server.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});

app.use(express.static("public"));

app.get("/", (req, res) => {
  res.sendFile("index.html");
});
```

```
io.on("connection", (socket) => {
  console.log(`user connected ${socket.id}`);

  socket.on("join", (data) => {
    socket.join(data.room);
    io.in(data.room).emit("message", `New user joined ${data.room} room!`);
  });

  // ecoute les messages envoyés par le client
  socket.on("message", (message) => {
    console.log(`message: ${message}`);
    io.emit("message", message);
  });

  socket.on("disconnect", () => {
    console.log("user disconnected");
    io.emit("message", "user disconnected");
  });
});
```

Création d'un page index.hml dans le dossier public

```
<body>
        <div>
            <ul id="messages"></ul>
        </div>

            <form id="chat-form">
                <input id="message" name="message" class="form-
control" autocomplete="off" placeholder="Message...">
                <input type="submit">
            </form>

        <script src="/socket.io/socket.io.js"></script>
        <script>
            const form = document.getElementById("chat-form");
            const messageInput = document.getElementById("message");
            const messageList = document.getElementById("messages");

            const socket = io(); // connexion
            const room = "Javascript";

            socket.on('connect', () => {
                socket.emit('join', {room: room});
            });

            socket.on("message", (message) => {
                const li = document.createElement("li");
                li.innerText = message;
                messageList.appendChild(li);
            });

            form.onsubmit = function (ev) {
                ev.preventDefault();
                // envoi au server du message
                const value = messageInput.value;
                socket.emit("message", value);

                // clear input form
                messageInput.value = "";
            };
        </script>
    </body>
```

Liste affichant les messages

Form pour l'envoi de messages

Injecté au build par socket.io

On envoie un message au serveur indiquant qu'on join une room

Retour du serveur, ajout du message à la liste

On envoie un message au serveur

On peut faire une variante en créant un « namespace »

```
const express = require("express");
const app = express();
const server = require("http").Server(app);
const io = require("socket.io")(server);
const port = 3000;

// attention c'est le server retourné avec http, pas l'app d'express
server.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});

app.use(express.static("public"));

app.get("/", (req, res) => {
  res.sendFile("index.html");
});

const chat = io.of("/chat");

chat.on("connection", (socket) => {
  console.log(`user connected ${socket.id}`);

  socket.on("join", (data) => {
    socket.join(data.room);
    chat.in(data.room).emit("message", `New user joined ${data.room} room!`);
  });

  // ecoute les messages envoyés par le client
  socket.on("message", (message) => {
    console.log(`message: ${message}`);
    chat.emit("message", message);
  });

  socket.on("disconnect", () => {
    console.log("user disconnected");
    chat.emit("message", "user disconnected");
  });
});
```

Et côté client

```
const socket = io("/chat"); // connexion
```

Avec React

Côté server on peut adapter le code pour le rendre un peu plus propre

```
const http = require("http");
const express = require("express");
const socketIo = require("socket.io");
//const cors = require("cors");
const app = express();
const port = 5000;

//app.use(cors()); // si d'autres endpoints que pour le chat

const server = http.createServer(app);
server.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});

const io = socketIo(server, {
  cors: {
    origin: "http://localhost:3000",
    methods: ["GET", "POST"],
    credentials: true,
  },
});
```

```
const chat = io.of("/chat");

chat.on("connection", (socket) => {
  console.log(`user connected ${socket.id}`);

  socket.on("join", (data) => {
    socket.join(data.room);
    chat.in(data.room).emit("message", `New user joined ${data.room} room!`);
  });

  socket.on("message", (message) => {
    console.log(`message: ${message}`);
    chat.emit("message", message);
  });

  socket.on("disconnect", () => {
    console.log("user disconnected");
    chat.emit("message", "user disconnected");
  });
});
```

Côté client

```
import React, { useState, useEffect } from "react";
import io from "socket.io-client";

const socket = io.connect("http://localhost:5000/chat");

const MessageList = () => {
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    socket.on("message", (msg) => {
      setMessages((messages) => [...messages, msg]);
    });
  }, []);

  return (
    <ul>
      {messages.map((msg, index) => (
        <li key={index}>{msg}</li>
      ))}
    </ul>
  );
};

const MessageForm = () => {
  const [message, setMessage] = useState("");

  useEffect(() => {
    socket.emit("join", { room: "javascript" });
  }, []);

  const inputHandler = (ev) => {
    setMessage(ev.target.value);
  };

  const submitHandler = (ev) => {
    ev.preventDefault();
    // validation ...

    socket.emit("message", message);
    // clear
    setMessage("");
  };
  return (
    <form onSubmit={submitHandler}>
      <input
        id="message"
        name="message"
        placeholder="Message..."
        value={message}
        onChange={inputHandler}
```

```
      />
      <input type="submit" />
    </form>
  );
};

export default function App() {
  return (
    <div>
      <MessageList />
      <MessageForm />
    </div>
  );
}
```

Au plus simple on a une liste des messages et une form pour en envoyer de nouveaux.

Plusieurs users peuvent se connecter à l'url du client http://localhost:3000 dans l'exemple

- New user joined javascript room!
- New user joined javascript room!
- Hi
- Hello, how are you ?

Message... Envoyer

Tests

Avec Mocha

Assez ancien mais convient bien pour Node.js : Mocha + chai

Mocha est le test runner.

Chai fournit un ensemble d'assertions « mieux » que celles de mocha :

- « assert »
- « expect » ou « should »

Installation

```
npm i mocha chai -D
```

On peut installer Mocha en global si on veut pouvoir entrer directement la commande « mocha » dans la console.

Script (ou lancement de tests en ligne de commande)

Par défaut Mocha va chercher les fichiers de tests dans un **dossier** « **test** » (avec la commande « mocha », sinon il faut définir pattern)…

Ajout d'un **script** dans « package.json »

```
  "scripts": {
    "test": "mocha"
  },
```

Ou avec le pattern

Chemin entre apostrophes

```
  "scripts": {
    "test": "mocha './test/**/*.spec.js'"
  },
```

Watch : les tests sont réexécutés aux changements dans le code

```
  "scripts": {
    "test": "mocha --watch './test/**/*.spec.js'"
  },
```

Que l'on lance avec « **npm test** »

Création du dossier de tests nommé « test »

Même si les fichiers ne finissent pas par « *.spec.js » ou « *.test.js »

Attention à git qui peut ignorer le dossier « test ».

```
// const assert = require("assert"); // mocha

const assert = require("chai").assert; // chai

describe("First test suite", () => {
  // message commençant par should avec le resultat attendu
  it("should return result", () => {
    assert.equal(1 + 1, 2);
  });
  it("should throw errors", () => {
    throw { message: "my error message" };

    //on peut afficher dans la console l'erreur si besoin. Exemple
    // try {
    //   assert.equal(1 + 1, 3);
    // } catch (err) {
    //   console.log(err);
    // }
  });
});
```

Exemple de résultats

```
First test suite
  ✓ should return result (38ms)
  1) should throw errors


1 passing (119ms)
1 failing

1) First test suite
     should throw errors:
   my error message
```

Afficher le message d'erreur

```
  First test suite
    ✓ should return result
AssertionError: expected 2 to equal 3
    at Context.<anonymous> (C:\Users\romag\Documents\Node projects\node-mocha-test\test\first.js:14:14)
    at callFn (C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runnable.js:366:21)
    at Test.Runnable.run (C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runnable.js:354:5)
    at Runner.runTest (C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runner.js:680:10)
    at C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runner.js:803:12
    at next (C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runner.js:595:14)
    at C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runner.js:605:7
    at next (C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runner.js:488:14)
    at Immediate._onImmediate (C:\Users\romag\Documents\Node projects\node-mocha-test\node_modules\mocha\lib\runner.js:573:5)
    at processImmediate (internal/timers.js:464:21) {
  showDiff: true,
  actual: 2,
  expected: 3,
  operator: 'strictEqual'
}
    ✓ should throw errors


  2 passing (48ms)
```

Assert, expect, should

Chai API

Le choix dépend de ses préférences. Venant du monde .NET j'ai tendance à préférer assert.

Avec assert

```
const assert = require("chai").assert; // chai

describe("First test suite", () => {
  it("should return result", () => {
    let operation = 1 + 1;
    assert.equal(operation, 2);
  });
});
```

Ne pas confondre assert de chai et assert de Mocha

```
// const assert = require("assert"); // mocha
const assert = require("chai").assert; // chai
```

Avec expect

expect(operation).to….

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

describe("First test suite", () => {
  it("should return result", () => {
    let operation = 1 + 1;
    expect(operation).to.equal(2);
  });
});
```

Avec should

variable.should…

```
const should = require("chai").should(); // méthod

describe("First test suite", () => {
  it("should return result", () => {
    let operation = 1 + 1;
    operation.should.equal(1 + 1, 2);
  });
});
```

chai-as-promised

```
npm i chai-as-promised -D
```

Plugin permettant de tester les méthodes retournant une promise avec. Je ne vois pas trop l'intérêt personnellement puisque l'on peut utiliser async await

```
const should = require("chai").should();
const chai = require("chai");
const chaiAsPromised = require("chai-as-promised");
chai.use(chaiAsPromised);

describe("First test suite", () => {

  function getMessage() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve("Ok");
      }, 1500);
    });
  }

  it("should execute async function", () => {
    return getMessage().should.eventually.equal("Ok");
  });
});
```

Before, beforeEach, after, afterEach

Mocha dispose de hooks qui sont exécutés respectivement :

- before : 1 fois au lancement des tests
- beforeEach : avant chaque test (définit avec « it »)
- after : 1 fois à la fin des tests
- afterEach : 1 fois après chaque test

Exemple

```
const assert = require("chai").assert; // chai

describe("First test suite", () => {
  before(() => {
    console.log("before");
  });
  beforeEach(() => {
    console.log("before each");
  });
  after(() => {
    console.log("after");
  });
  afterEach(() => {
    console.log("after each");
  });
  it("should return result", () => {
    assert.equal(1 + 1, 2);
  });
  it("should throw errors", () => {
    throw { message: "my error message" };
  });
});
```

Async test

Avec le callback « done »

```
  it("should execute async test", (done) => {
    setTimeout(() => {
      assert.equal(1 + 1, 2);
      done();
    }, 1500);
  });
```

Avec async await

```
  it("should execute async test", async () => {
    await setTimeout(() => {
      assert.equal(1 + 1, 2);
    }, 1500);
  });
```

Function retournant une promise

Exemple de function testée

```
  function getMessage() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve("Ok");
      }, 1500);
    });
  }
```

… et le test

```
  it("should execute async function", async () => {
    const result = await getMessage();
```

```
    assert.equal(result, "Ok");
  });
```

« only »

Pour n'exécuter qu'un test sur « it » ou un groupe de tests sur « describe »

```
const assert = require("chai").assert; // chai

describe("First test suite", () => {
  it.only("should return result", () => {
    assert.equal(1 + 1, 2);
  });
  it("should throw errors", () => {
    throw { message: "my error message" };
  });
});
```

Que ce test exécuté

… ou qu'un groupe de tests exécutés

```
const assert = require("chai").assert; // chai

describe("First test suite", () => {
  describe.only('my group', () => {
    it("should return result", () => {
      assert.equal(1 + 1, 2);
    });
    it("should throw errors", () => {
      throw { message: "my error message" };
    });
  })

 // ... autres tests
```

Que ces tests exécutés

Spy, stubs, mocks

Installation de sinon

```
npm i sinon -D
```

Spy pour « espionner » une function

Exemple tester la function « render » de « res » . Dans un controller :

```
const getIndex = (req, res) => {
  res.render("index");
};
exports.getIndex = getIndex;
```

On vérifie que la function render est appelée (1 fois) et que l'argument est bien « index »

```
const assert = require("chai").assert;
const sinon = require("sinon");
const myControllers = require("../../controllers/my-controllers");

describe("my controllers test", () => {
  it("should render index", () => {
    const req = {};
    const res = {
      render: sinon.spy(),
    };
    myControllers.getIndex(req, res);
```

On peut avoir divers informations : le nombre de fois que la function a été appelée, la valeur de ses arguments, etc.

```
    assert.isTrue(res.render.calledOnce);
    assert.equal(res.render.args[0], "index");
  });
});
```

Avec mock

C'est une alternative à « spy ». On définit ce que l'attend (expects) et on vérifie ensuite.

```
  it("should render index", () => {
    const req = {};
    const res = {
      render: function () {},
    };
    const mock = sinon.mock(res);
    mock.expects("render").once().withExactArgs("index");

    myControllers.getIndex(req, res);

    mock.verify();
  });
```

Observer une function existante

Avec

sinon.spy(<object>, "function_name")

Exemple

```
const assert = require("chai").assert;
const sinon = require("sinon");
const myControllers = require("../../controllers/my-controllers");

const sample = {
  getMessage: (name) => `hello ${name}!`,
};

describe("my controllers test", () => {
  it("should spy existing function", () => {
    sinon.spy(sample, "getMessage");

    sample.getMessage("world");

    assert.isTrue(sample.getMessage.calledOnce);
    assert.equal(sample.getMessage.args[0], "world");
  });
});
```

Remplacer une function avec stub()

On peut imaginer par exemple un service qui effectue une requête http. Ce n'est pas trop testable.

```
const assert = require("chai").assert;
const sinon = require("sinon");

const sample = {
  getMessage: (name) => `hello ${name}!`,
};

describe("my controllers test", () => {
  it("should spy existing function", () => {
    const fakeFn = function (name) {
      return `Fake hello ${name}!`;
    };
    sinon.stub(sample, "getMessage").callsFake(fakeFn);

    const message = sample.getMessage("world");
    assert.equal(message, "Fake hello world!");
    // plus
    assert.isTrue(sample.getMessage.calledOnce);
    assert.equal(sample.getMessage.args[0], "world");
  });
});
```

Ici on utilise une fake function pour le retour

On pourrait aussi utiliser « returns » pour une function qui retourne un boolean par exemple

```
sinon.stub(user, "isAuthorized").returns(true);
```

Astuce pour débugger ses tests avec VS Code

Création d'un fichier « launch.json ». Onglet DEBUG … add configuration … node et modifier par :

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=83
0387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "pwa-node",
            "request": "launch",
            "name": "Mocha",
```

```
            "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/mocha",
            "runtimeArgs": [
                "--colors",
                "--recursive",
                "${workspaceRoot}/test/**/*.js"
            ],
            "internalConsoleOptions": "openOnSessionStart"
        }
    ]
}
```

Avec Jest

Installation

En globale si on désire pouvoir entrer la commande jest depuis une invite de commande ou en local

```
npm i jest -D
```

Par défaut jest cherche les tests dans un dossier « __tests__ » ou les fichiers qui contiennent « spec » ou « test » (exemple : « mycontroller.spec.js »)

```
PS C:\Users\romag\Documents\Node projects\node-jest-sample> npm test

> node-jest-sample@1.0.0 test C:\Users\romag\Documents\Node projects\node-jest-sample
> jest

No tests found, exiting with code 1
Run with `--passWithNoTests` to exit with code 0
In C:\Users\romag\Documents\Node projects\node-jest-sample
  3 files checked.
  testMatch: **/__tests__/**/*.[jt]s?(x), **/?(*.)+(spec|test).[tj]s?(x) - 0 matches
  testPathIgnorePatterns: \\node_modules\\ - 3 matches
  testRegex:  - 0 matches
Pattern:  - 0 matches
npm ERR! Test failed.  See above for more details.
```

Pour executer les tests commande « jest » ou ajout d'un script à « package.json »

```
  "scripts": {
    "test": "jest"
  },
```

Puis « npm test »

Ecriture de test

expect

expect(operation).to….

Exemple

```
describe("Sample test", () => {
```

Pas besoin d'import pour utiliser expect

```
  it("should returns the result", () => {
    expect(1 + 1).toEqual(2);
  });
});
```

Exemple de résultats de tests

```
> node-jest-sample@1.0.0 test C:\Users\romag\Documents\Node projects\node-jest-sample
> jest

 PASS  ./sample.test.js
  Sample test
    √ should returns the result (3 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.961 s, estimated 1 s
Ran all test suites.
```

Test asynchrone

Avec le callback **done**

```
  it("should wait timeout", (done) => {
    setTimeout(() => {
      expect(1 + 1).toEqual(2);
      done();
    }, 1500);
  });
```

Avec **Promise** et **async await**

```
  function fn(name) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve(`Hello ${name}!`);
      }, 1500);
    });
  }

  it("should work with promise", async () => {
    const result = await fn("world");
    expect(result).toEqual("Hello world!")
  });
```

only

Jest supporte également only sur « it » (un seul test exécuté) et « describe » (groupe de tests exécutés)

```
  it.only("should returns the result", () => {
```

```
    expect(1 + 1).toEqual(2);
  });
```

Mocks

https://jestjs.io/fr/docs/mock-functions

https://jestjs.io/docs/mock-function-api

Exemple

Code testé

```
const getIndex = (req, res) => {
  res.render("index");
};
exports.getIndex = getIndex;
```

Exemple tester la function « render » de « res »

```
const myControllers = require("./controllers/mycontrollers");

describe("my controllers test", () => {
  it("should render index", () => {
    const req = {};
    const mockCallback = jest.fn();
    const res = {
      render: mockCallback
    };
    myControllers.getIndex(req, res);
    expect(mockCallback.mock.calls.length).toBe(1);
    expect(mockCallback.mock.calls[0][0]).toBe("index");
  });
});
```

Plus

Création d'un serveur sans Express

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

const PORT = 5000;

const server = http.createServer((req, res) => {
  const url = req.url;
  const method = req.method;

  console.log(url, method);
  // / GET
  // /api/test GET

  // on peut tester l'url et la méthode pour définir la réponse à retourner
  if (url == "/" && method == "GET") {
    res.setHeader("Content-Type", "text/html");
    return res.end("<h1>Home</h1>");
  }
  // json
  if(url=="/api/test" && method == "GET"){
```

```
    res.setHeader("Content-Type", "application/json");
    return res.end(JSON.stringify({message: "test"}));
  }

  // not found
  res.setHeader("Content-Type", "text/html");
  return res.end("<h1>404</h1>");
});

server.listen(PORT, () => {
  console.log(`Backend server is running on port ${PORT}`);
});
```

Création d'une classe d'error personnalisée (HttpError)

```
class HttpError extends Error {
    constructor(message, errorCode) {
      super(message);
      this.code = errorCode;
    }
  }
module.exports = HttpError;
```

Middlewares

```
const express = require("express");

const app = express();

// middlewares
app.use((req, res, next) => {
  console.log("middleware 1");
  next();
});

app.use((req, res, next) => {
  console.log("middleware 2");
  next();
});

app.use((req, res, next) => {
  console.log("middleware 3");
  res.send("It works"); // on doit renvoyer une réponse sinon la page charge indéfiniment
  // dernier middleware, next peut être omis ?
});

app.listen(5000, () => {
  console.log("Backend server is running!");
});
```

Dans la console

```
Backend server is running!
middleware 1
middleware 2
middleware 3
```

Mieux comprendre l'ordre d'exécution avec erreurs, réponses, middlewares

```
const express = require("express");

const app = express();

// toujours appelé en premier
app.use((req, res, next) => {
  console.log("In middleware before");
  next();
});

app.get("/:id", (req, res, next) => {
```

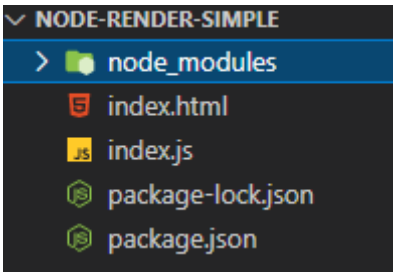
```
  console.log("In route");
  const id = req.params.id;
  if (id === "10") {
    const error = new HttpError(
      "Invalid credentials, could not log you in.",
      403
    );
    return next(error); // execption: pas de middlewares appelé après
  }

  res.json({ message: "Ok !" }); // réponse: pas de middlewares appelé après
});

// appelé seulement si pas de route trouvée
app.use((req, res, next) => {
  const error = new HttpError("Could not find this route.", 404);
  throw error;
});

app.listen(5000, () => {
  console.log("Backend server is running!");
});
```

Création d'un serveur qui affiche une page html

Simple

```
const express = require("express");
const path = require("path");

const app = express();

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "index.html"));
});
// or
// const router = express.Router();
// router.get("/", (eq, res) => {
//   res.sendFile(path.join(__dirname, "index.html"));
// });
// app.use("/", router);

app.listen(5000, () => {
  console.log("Server is running");
});
```

Structure simple



Avec View Engine

Moteurs de vues : swig, ejs, pug, react, … liste

Exemple avec « swig » (documentation)

On installe le moteur de vue « npm i swig »

On crée un dossier « views »

Création d'un layout « layout.html » dans le dossier « views »

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Swig Sample</title>
    </head>
    <body>
        {% block content %}{% endblock %}
    </body>
</html>
```

Et d'une view « index.html » dans le dossier « views »

```
{% extends 'layout.html' %}

{% block content %}
<h1>Hello {{username}}!</h1>
{% endblock %}
```

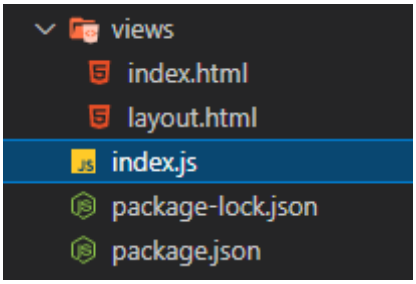
Dans « index.js »

```
const express = require("express");
const path = require("path");
const swig = require('swig');

const app = express();

app.engine("html", swig.renderFile);

app.set("view engine", "html");
app.set("views", path.join(__dirname, "/views"));

app.get("/", (req, res) => {
  res.render("index", { username: "jerome" });
});

app.listen(5000, () => {
  console.log("Server is running");
});
```

Configuration du view engine et du chemin vers les views

Utilisation de la méthode « render » et passage de paramètre par exemple

Structure du projet



Debugger avec VS Code

Menu « run » … « Add configuration » … « Node »

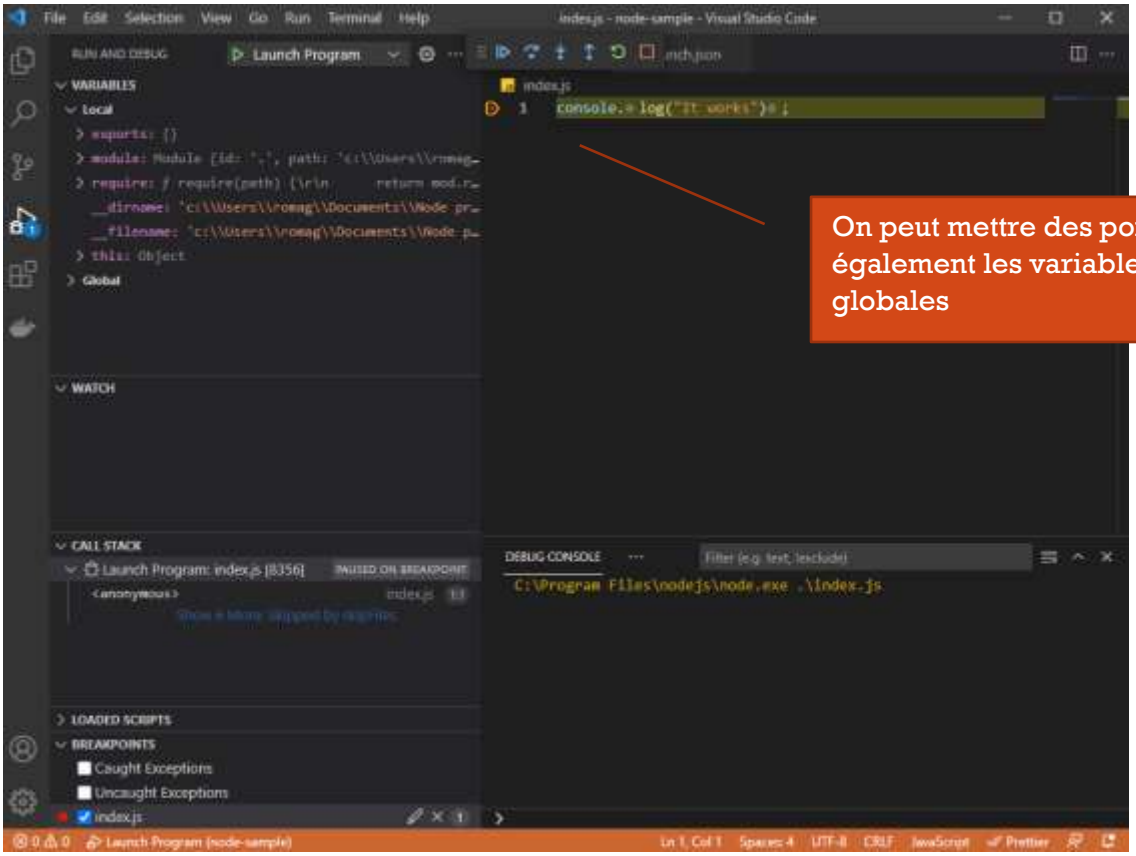
Un fichier « launch.js » est ajouté dans un dossier « .vscode »

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
```

```
    "version": "0.2.0",
    "configurations": [
        {
            "type": "pwa-node",
            "request": "launch",
            "name": "Launch Program",
            "skipFiles": [
                "<node_internals>/**"
            ],
            "program": "${workspaceFolder}\\index.js"
        }
    ]
}
```

Chemin vers le fichier à exécuter (« index.js » par défaut)

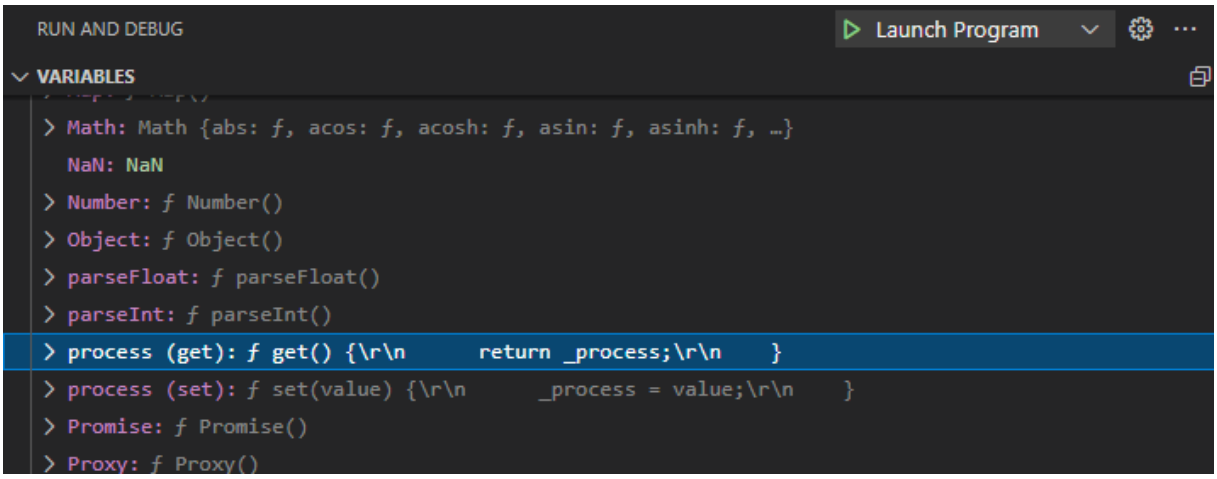
Depuis l'onglet « RUN AND DEBUG » , cliquer sur « launch program »



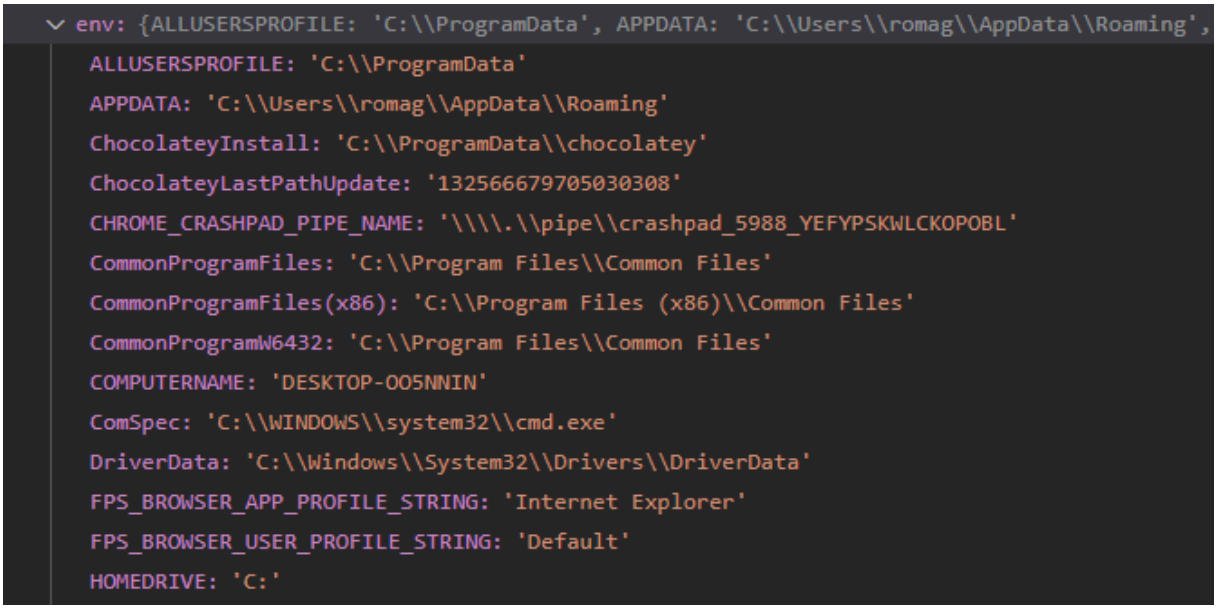
Variables d'environnement

En « debug » on peut voir la variable globale « process.env »

VARIABLES … « global » … « process(get) »



… « env »



On peut ainsi définir des variables dans « process.env » et retrouver/ utiliser leurs valeurs. Exemples « process.env.PORT », « process.env.NODE_ENV »

En ligne de commande

avec Powershell (terminal VS Code)

```
$env:NODE_ENV="dev"
node index.js
```

En ligne de commande ou dans les scripts avec « set …. »

```
"scripts": {
    "start": "set PORT=8000 && set NODE_ENV=dev && node index.js",
  },
```

Sur mac et Linux on peut faire

```
"start":"NODE_ENV=dev PORT=8000 node index.js"
```

Avec « cross-env »

Permet de définir les variables d'environnement dans les scripts de package.json

```
npm i cross-env
```

.. et définition dans le script de package.json des variables

```
"scripts": {
    "start": "cross-env NODE_ENV=production node index.js"
  },
```

On peut imaginer avoir un fichier de config

```
var path = require('path');
var rootPath = path.normalize(__dirname + "/../");

module.exports = {
    development: {
        db: 'mongodb://localhost:27017/local-dev',
        rootPath: rootPath,
        port: process.env.PORT || 3000,
        facebook: {
            clientID: '963837333707758',
            clientSecret: '0e9a6cea4a2a9730c68ecb41db985229',
            callbackURL: '/api/auth/facebook/callback'
        },
        google: {
            clientID: '456807293205-
pf5lsl58nllhmurec0rl1f6fhfrkcnj1.apps.googleusercontent.com',
            clientSecret: '093NEpj1eHEDlVLWzDVO8f0Z',
            callbackURL: 'http://localhost:3000/api/auth/google/callback'
        }
    },
    production: {
        rootPath: rootPath,
        db: 'mongodb://....', // à configurer
        port: process.env.PORT || 80,
        facebook: {
            clientID: '963837333707758',
            clientSecret: '0e9a6cea4a2a9730c68ecb41db985229',
            callbackURL: '/api/auth/facebook/callback'
        },
        google: {
            clientID: '456807293205-
pf5lsl58nllhmurec0rl1f6fhfrkcnj1.apps.googleusercontent.com',
            clientSecret: '093NEpj1eHEDlVLWzDVO8f0Z',
            callbackURL: 'http://localhost:3000/api/auth/google/callback'
        }
    }
};
```

Et récupérer la configuration selon

```
var env = process.env.NODE_ENV || 'development';
var config = require('./config')[env];
```

Dans le code

On peut afficher et changer la valeur de « NODE_ENV »

```
console.log(process.env.NODE_ENV); // undefined
process.env.NODE_ENV="production";
```

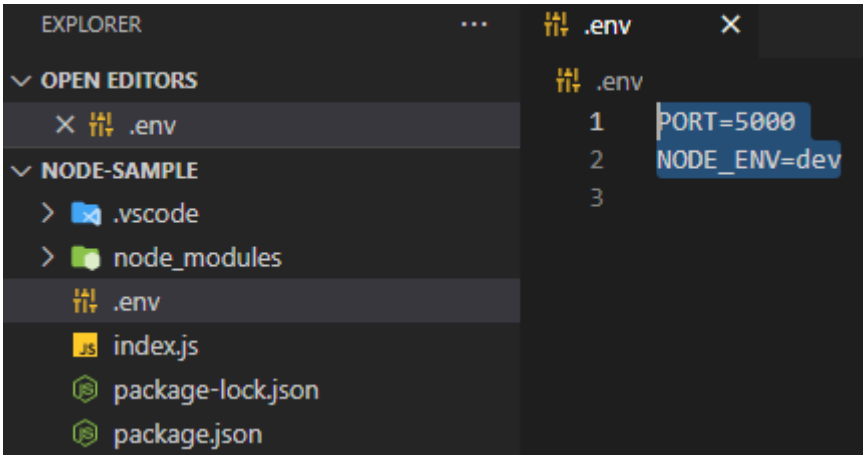
```
console.log(process.env.NODE_ENV); // production
```

Avec « **dotenv** » création d'un fichier « .env »

npm

Permet de définir les variables d'environnement dans un fichier « .env » créé à la racine du projet de backend

```
npm i dotenv
```



Dans « index.js ». « dotenv » va lire les variables définies dans le fichier « .env » et les affecter à la variable globale « process.env »

```
const dotenv = require("dotenv");
dotenv.config();

console.log(process.env.PORT); // 5000
console.log(process.env.NODE_ENV); // dev
process.env.NODE_ENV="production";
console.log(process.env.NODE_ENV); // production
```

Il est possible de faire un fichier de config « dotenv/config.js » et ne plus avoir à définir la config dans « index.js »

```
const dotenv = require("dotenv");

const result = dotenv.config();
if (result.error) throw result.error;
const { parsed: envs } = result;
module.exports = envs;
```

… on indique le fichier de config dans le script

```
"dev":"node -r dotenv/config index.js"
```

On peut importer les variables

```
const envs = require("./dotenv/config");
console.log(envs); // { PORT: '5000', NODE_ENV: 'dev' }
```

Nodemon

Pendant le développement on peut définir ses variables avec nodemon. Dans un fichier « nodemon.json » à la racine du projet de backend

```
{
    "env": {
        "PORT": 1100,
        "NODE_ENV": "nodemon_test"
    }
}
```

Le script ne change pas vraiment avec nodemon

```
"start:dev": "nodemon app.js"
```