

Codelab: Authentification des comptes utilisateurs avec passport-local

Utilisation de Passport avec Sequelize et PostgreSQL/MySQL

Objectifs

Sequelize est un ORM Node.js basé sur des promesses. Il peut être utilisé avec PostgreSQL, MySQL, MariaDB, SQLite et MSSQL. Dans ce tutoriel, nous allons implémenter l'authentification pour les utilisateurs d'une application Web. Et nous utiliserons ***Passport***, le middleware d'authentification populaire pour Node, avec Sequelize et MySQL pour mettre en œuvre l'enregistrement et la connexion des utilisateurs.

Base de données

Ce codelab utilisera MySQL. Vous pouvez utiliser PostgreSQL ou une autre base de données. Assurez-vous simplement de mettre à jour le fichier **.sql** si nécessaire et les paramètres de configuration (port, dialect, etc.).

Nous allons utiliser la base de données : **bdd_node_1** (consultez Moodle / cours-info).

Étape 1: Générez un fichier package.json

Créez un répertoire pour votre application. Dans ce répertoire, exécutez ceci depuis votre terminal:

```
$ npm init --yes
```

Cela initialise le gestionnaire de dépendances npm (npm Dependency Manager).

Étape 2: installer les dépendances

Les principales dépendances de ce tutoriel sont:

- Express
- Sequelize
- MySQL
- Passport
- Passport Local Strategy
- Body Parser

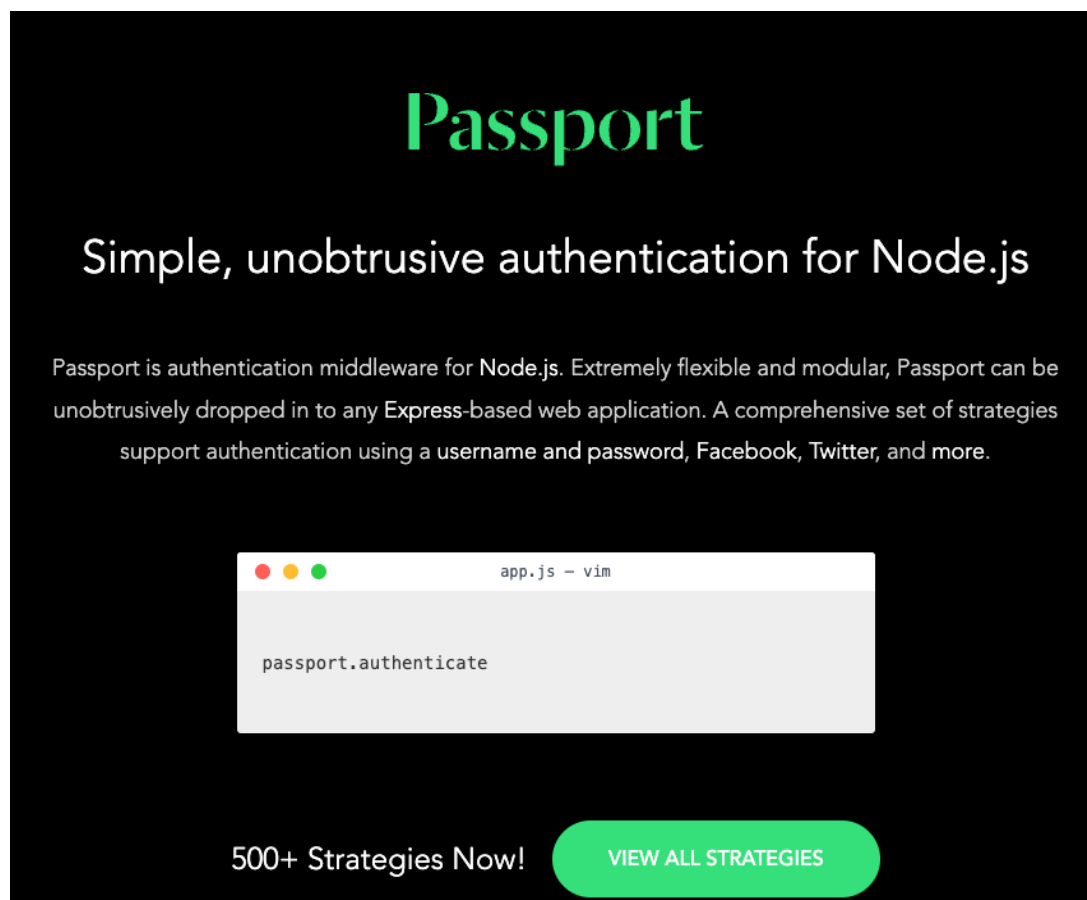
- Express Session
- Bcrypt Nodejs
- Express Handlebars pour les vues

Pour les installer, depuis votre terminal, exécutez les commandes suivantes l'une après l'autre:

- `npm install express --save`
- `npm install sequelize --save`
- `npm install mysql mysql2 --save`
- `npm install passport --save`
- `npm install passport-local --save`
- `npm install body-parser --save`
- `npm install express-session --save`
- `npm install bcrypt --save`
- `npm install express-handlebars --save`

Qu'est-ce que Passport?

Passport est un middleware d'authentification pour Node.js. Extrêmement flexible et modulaire, Passport peut être déposé discrètement dans n'importe quelle application Web basée sur Express. Un ensemble complet de stratégies prend en charge l'authentification à l'aide d'un nom d'utilisateur et d'un mot de passe, Facebook, Twitter, etc.



The image is a promotional banner for Passport.js. It features a dark background with the word "Passport" in a large, green, serif font at the top. Below it, the text "Simple, unobtrusive authentication for Node.js" is written in a white, sans-serif font. A paragraph of text describes Passport as an authentication middleware for Node.js that is flexible, modular, and can be easily integrated into Express-based web applications. Below this text is a screenshot of a code editor window titled "app.js - vim" showing the code `passport.authenticate`. At the bottom left, it says "500+ Strategies Now!" and at the bottom right, there is a green button with the text "VIEW ALL STRATEGIES".

Passport

Simple, unobtrusive authentication for Node.js

Passport is authentication middleware for Node.js. Extremely flexible and modular, Passport can be unobtrusively dropped in to any Express-based web application. A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, and more.

```
app.js - vim
```

```
passport.authenticate
```

500+ Strategies Now! [VIEW ALL STRATEGIES](#)

Étape 3: Configurer l'application

Maintenant, nous créons un fichier serveur:

```
$ touch server.js
```

Dans le fichier **server.js**, nous collons ce qui suit:

```
const express = require('express');
const app = express();
app.get('/', function(req, res) {
  res.send('Welcome to Passport with Sequelize');
});
app.listen(3000, function(err) {
  if (!err)
    console.log("Le serveur écoute sur le port 3000");
  else console.log(err)
});
```

Ensuite, nous importons certains modules dont nous avons besoin, comme **passport**, **express-session**, et **body-parser**. Après **const app = express()** nous ajoutons les lignes suivantes:

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

Dans les deux premières lignes, nous importons les modules **passport** et **session express**, dont nous avons tous deux besoin pour gérer l'authentification.

Ensuite, nous importons le module **body-parser**. Cela extrait toute la partie du corps d'une requête entrante et l'expose dans un format plus facile à utiliser. Dans ce cas, nous utiliserons le format JSON.

Pour permettre à notre application d'utiliser le **body-parser**, nous ajoutons ces lignes sous les lignes d'importation:

```
//Pour bodyParser
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Ensuite, nous initialisons **passport**, **session express**, et **session passport** et les ajoutons tous les deux en tant que middleware. Pour ce faire, nous ajoutons ces lignes après la ligne **app.use(bodyParser.json());**

```
// Pour Passport
app.use(session({ secret: 'butinfo', resave: true, saveUninitialized: true }));
app.use(passport.initialize());
app.use(passport.session());
```

Nous allons maintenant commencer à travailler sur l'authentification. Nous allons procéder en quatre étapes:

- Configurez Sequelize avec MySQL
- Créez le modèle utilisateur (user)
- Configurer des vues
- Ecrire une stratégie de passeport

Étape 4: Configurer Sequelize avec MySQL

Dans ce tutoriel, nous utiliserons la base de données **bdd_node_1** qui contient les tables: users, posts, et comments. Importons le module dot-env pour gérer les variables d'environnement. Exécutez ceci dans votre dossier de projet racine:

```
$ npm install --save dotenv
```

Dotenv est un module sans dépendance qui charge les variables d'environnement d'un fichier **.env** dans **process.env**.

Ensuite, nous l'importons dans le fichier du serveur principal, **server.js**, juste en dessous des autres importations.

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

Ensuite, nous créons un fichier dans notre dossier de projet et le nommons **.env**.

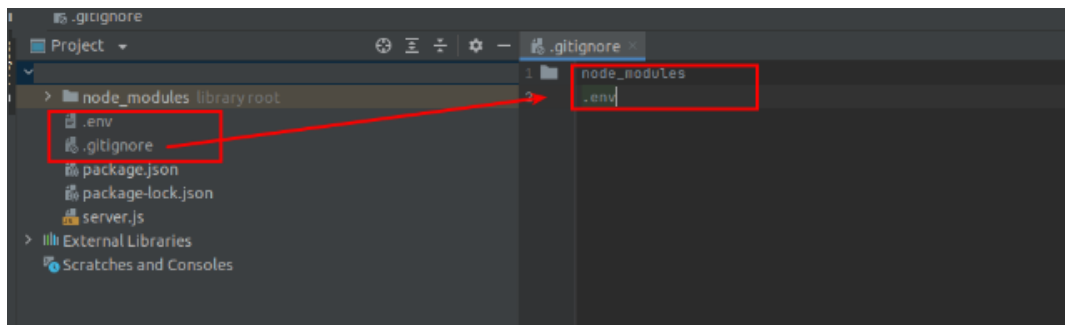
```
$ touch .env
```

Cette prochaine étape à suivre est facultative si vous n'utilisez pas **Git**:

Nous ajouterons le fichier **.env** à votre fichier **.gitignore**. La raison en est que nous ne voulons pas pousser notre fichier **.env** lors du **push**.

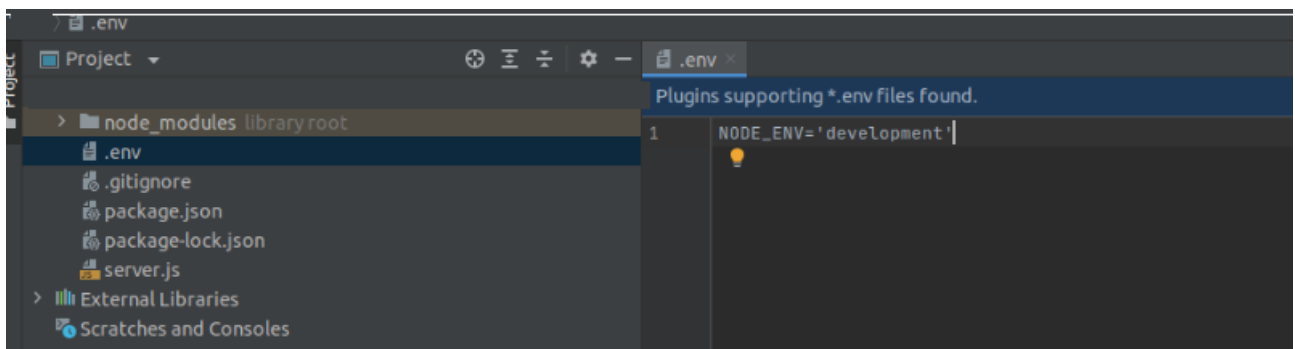
```
$ touch .gitignore
```

Votre fichier **.gitignore** devrait ressembler à ceci:



Après cela, nous ajoutons notre environnement au fichier **.env** en ajoutant cette ligne:

NODE_ENV= 'development'



Ensuite, nous créons un fichier **config.json** qui sera utilisé par Sequelize pour gérer différents environnements.

La première chose à faire est de créer un dossier nommé **config** dans notre dossier de projet. Dans ce dossier, nous créons un fichier **config.json**.

```
$ mkdir config
```

```
$ touch config/config.json
```

Ce fichier doit être ignoré si vous poussez vers un référentiel. Pour ce faire, ajoutez le code suivant à votre **.gitignore**:

config/config.json

Ensuite, nous collons le code suivant dans notre fichier **config.json**.

```
{
  "development": {
    "username": "root",
    "password": "root",
    "database": "bdd_node_1",
    "host": "localhost",
    "port" : 3306,
    "dialect": "mysql",
    "define": {
      "timestamps": false
    }
  },
  "test": {
    "username": "",
    "password": null,
    "database": "",
    "host": "",
    "dialect": "mysql",
    "define": {
      "timestamps": false
    }
  },
  "production": {
    "username": "",
    "password": null,
    "database": "",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "define": {
      "timestamps": false
    }
  }
}
```

Si vous utilisez **Postgres** :

```
... {} config.json ×
config > {} config.json > ...
1  {
2    "development": {
3      "username": "josephazar",
4      "password": null,
5      "database": "bd_passportlocal",
6      "host": "localhost",
7      "port": 5432,
8      "dialect": "postgres"
9    },
10   "test": {},
11   "production": {}
12 }
```

Configuration Postgres

N'oubliez pas de remplacer les valeurs du bloc de développement ci-dessus par les détails d'authentification de votre base de données. Si vous utilisez PostgreSQL, remplacez également le paramètre "dialect".

Il est maintenant temps de créer le dossier **models** et de créer un nouveau fichier nommé **index.js** dans le dossier **models**.

```
$ mkdir models
```

```
$ touch models/index.js
```

Dans le fichier **index.js**, nous collons le code ci-dessous.

```
"use strict";

const fs      = require("fs");
const path    = require("path");
const Sequelize = require("sequelize");
const env     = process.env.NODE_ENV || "development";
const config  = require(path.join(__dirname, '..', 'config', 'config.json'))[env];
const sequelize = new Sequelize(config.database, config.username, config.password, config);
const db      = {};

fs.readdirSync(__dirname)
  .filter(function(file) {
    return (file.indexOf(".") !== 0) && (file !== "index.js");
```

```

    })

    .forEach(function(file) {

        let model = require(path.join(__dirname, file))(sequelize)

        db[model.name] = model;

    });

Object.keys(db).forEach(function(modelName) {

    if ("associate" in db[modelName]) {

        db[modelName].associate(db);

    }

});

db.sequelize = sequelize;

db.Sequelize = Sequelize;

module.exports = db;

```

Vous pouvez utiliser le mode strict (use strict) dans tous vos programmes. Il vous aide à écrire un code plus propre, en vous empêchant par exemple d'utiliser des variables non déclarées.

Le code ci-dessus est utilisé pour importer tous les modèles que nous plaçons dans le dossier **models** et les exporter.

Pour tester que tout va bien, nous ajoutons ce qui suit dans notre fichier **server.js**.

```

//Models
const models = require("./models");
//Sync Database
models.sequelize.sync().then(function() {
    console.log('La base de données fonctionne bien')
}).catch(function(err) {
    console.log(err, "Quelque chose s'est mal passé avec la mise à jour de la base de données!")
});

```

Ici, nous importons les modèles, puis appelons la fonction de synchronisation (Sequelize).

Exécutez ceci pour voir si tout va bien:

```
$ node server.js
```

Si vous obtenez le message "La base de données fonctionne bien", alors vous avez configuré Sequelize avec succès. Si ce n'est pas le cas, veuillez suivre attentivement les étapes ci-dessus et essayez de déboguer le problème.

Étape 5: Créer le modèle utilisateur

La prochaine chose que nous allons faire est de créer le modèle utilisateur, qui est essentiellement la table **"users"**. Celui-ci contiendra des informations de base sur l'utilisateur. Dans notre dossier **models**, nous créons un fichier et le nommons **users.js**.

```
$ touch models/users.js
```

Ouvrez le fichier **users.js** et ajoutez le code suivant:

```
const { DataTypes } = require('sequelize');

module.exports = function(sequelize) {
  const User = sequelize.define('user', {
    id: { autoIncrement: true, primaryKey: true, type: DataTypes.INTEGER, allowNull: false },
    firstName: { type: DataTypes.STRING, notEmpty: true },
    lastName: { type: DataTypes.STRING, notEmpty: true },
    emailId: { type: DataTypes.STRING, validate: { isEmail: true } },
    password: { type: DataTypes.STRING, allowNull: false },
  }, {
    tableName: 'users'
  });

  return User;
}
```

Exécutez maintenant:

```
$ node server.js
```

Vous devriez voir le message «La base de données fonctionne bien». Cela signifie que nos modèles Sequelize ont été synchronisés avec succès.

Étape 6: Configurer les vues

La première chose à faire est d'importer le module **express handlebars** que nous utilisons pour les vues dans ce tutoriel. Ajoutez cette ligne à **server.js**.

```
const expghbs = require('express-handlebars')
```

Ensuite, nous ajoutons les lignes suivantes dans notre fichier **server.js**.

```
//Pour Handlebars
app.set('views', './views')
app.set('view engine', '.hbs');
app.engine(
  'hbs',
  exphbs.engine({
    extname: ".hbs",
    defaultLayout: "",
    layoutsDir: "",
  })
);
```

Créez trois dossiers nommés **views**, **controllers** et **routes**.

```
$ mkdir views
```

```
$ mkdir controllers
```

```
$ mkdir routes
```

Dans le dossier **views**, nous créons un fichier nommé **signup.hbs** (\$ touch **views/signup.hbs**) et y collons le code ci-dessous.

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
<form id="signup" name="signup" method="post" action="/signup">
  <label for="email">Email Address</label>
  <input class="text" name="email" type="email" />
  <label for="firstname">Firstname</label>
  <input name="firstname" type="text" />
  <label for="lastname">Lastname</label>
  <input name="lastname" type="text" />
  <label for="password">Password</label>
  <input name="password" type="password" />
  <input class="btn" type="submit" value="Sign Up" />
</form>
</body>
</html>
```

Ensuite, dans notre dossier **controllers**, nous créons un nouveau fichier et le nommons **authcontroller.js**.

```
$ touch controllers/authcontroller.js
```

Dans ce fichier, nous collons le code suivant pour la route d'inscription que nous allons créer dans un instant.

```
exports.signup = function(req, res) {  
  res.render('signup');  
}
```

Ensuite, nous créons une route pour l'inscription. Dans le dossier **routes**, nous créons un nouveau fichier nommé **auth.js** (\$ touch routes/auth.js) puis, dans ce fichier, nous importons le contrôleur d'authentification et définissons la route d'inscription.

```
const authController = require('../controllers/authcontroller.js');  
module.exports = function(app,passport){  
  app.get('/signup', authController.signup);  
}
```

Maintenant, nous allons importer cette route dans notre fichier **server.js**.

```
//Routes  
const authRoute = require('./routes/auth.js')(app,passport);
```

```
//Models  
const models = require("./models");  
//Routes  
const authRoute = require('./routes/auth.js')(app,passport);  
//Sync Database  
models.sequelize.sync().then(function() {  
  console.log('La base de données fonctionne bien')  
}).catch(function(err) {  
  console.log(err, "Quelque chose s'est mal passé avec la mise à jour de la base de données!")  
});  
app.listen( port: 3000, hostname: function(err) {  
  if (!err)  
    console.log("Le serveur écoute sur le port 3000");  
  else console.log(err)  
});
```

Notez que nous passons **passport** au routeur. Nous l'utiliserons plus tard.

Exécutez cette commande:

```
$ node server.js
```

Maintenant, visitez <http://localhost:3000/signup> et vous verrez le formulaire d'inscription.

Répetons les étapes du formulaire d'inscription. Comme précédemment, nous allons créer un fichier nommé **signin.hbs** (\$ touch views/signin.hbs) dans notre dossier **views** et y coller le code HTML suivant:

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
<form id="signin" name="signin" method="post" action="signin">
  <label for="email">Email Address</label>
  <input class="text" name="email" type="text" />
  <label for="password">Password</label>
  <input name="password" type="password" />
  <input class="btn" type="submit" value="Sign In" />
</form>
</body>
</html>
```

Ensuite, ajoutez un contrôleur pour la connexion dans **controllers/authcontroller.js**.

```
exports.signin = function(req, res) {
  res.render('signin');
}
```

Ensuite, dans **routes/auth.js**, nous ajoutons une route pour la connexion comme celle-ci:

```
app.get('/signin', authController.signin);
```

```
const authController = require('../controllers/authcontroller.js');
module.exports = function(app, passport){
  app.get('/signup', authController.signup);
  app.get('/signin', authController.signin);
}
```

Maintenant, lorsque vous exécutez:

```
$ node server.js
```

et visitez <http://localhost:3000/signin/>, vous devriez voir le formulaire de connexion.

L'étape finale et majeure consiste à écrire nos stratégies **passport**.

Étape 7: Écrire une stratégie passport

Dans le dossier **config**, nous créons un nouveau dossier nommé **passport**.

```
$ mkdir config/passport
```

Ensuite, dans notre nouveau dossier **config/passport**, nous créons un nouveau fichier et le nommons **passport.js**. Ce fichier contiendra nos stratégies de passeport.

```
$ touch config/passport/passport.js
```

Copiez ce qui suit dans le fichier **passport.js**:

```
//load bcrypt
const bcrypt = require('bcrypt');
module.exports = function(passport,user){
  const User = user;
  const LocalStrategy = require('passport-local').Strategy;
  passport.serializeUser(function(user, done) {
    done(null, user.id);
  });
  // used to deserialize the user
  passport.deserializeUser(function(id, done) {
    // findbyPk ==> trouver l'utilisateur avec la clé primaire
    User.findById(id).then(function(user) {
      if(user){
        done(null, user.get());
      }
      else{
        done(user.errors,null);
      }
    });
  });
  //Nous déclarons ici quels champs de requête (req) sont nos usernameField et
  passwordField (variables de passeport)
  passport.use('local-signup', new LocalStrategy(
    {
      usernameField : 'email',
      passwordField : 'password',
      passReqToCallback : true // allows us to pass back the entire request to the
      callback
    },
```

```

// Dans cette fonction, nous allons gérer le stockage des détails d'un utilisateur
function(req, email, password, done){
  // nous ajoutons notre fonction de génération de mot de passe haché à l'intérieur de
  la fonction de rappel
  let generateHash = function(password) {
    return bcrypt.hashSync(password, bcrypt.genSaltSync(8), null);
  };
  // en utilisant le modèle utilisateur Sequelize que nous avons initialisé plus tôt,
  // nous vérifions si l'utilisateur existe déjà, et sinon nous l'ajoutons
  User.findOne({where: {emailId:email}}).then(function(user){
    if(user)
    {
      return done(null, false, {message : 'Cette email est déjà pris'} );
    }
    else
    {
      let userPassword = generateHash(password);
      let data =
      { emailId:email,
        password:userPassword,
        firstName: req.body.firstname,
        lastName: req.body.lastname
      };
      // User.create() est une méthode Sequelize pour ajouter de nouvelles entrées à
      la base de données.
      // Notez que les valeurs de l'objet de données sont obtenues à partir de
      l'objet req.body qui
      // contient l'entrée de notre formulaire d'inscription

      User.create(data).then(function(newUser,created){
        if(!newUser){
          return done(null,false);
        }
        if(newUser){
          return done(null,newUser);
        }
      });
    }
  });
}
});
// LOCAL SIGNIN
passport.use('local-signin', new LocalStrategy(
{
  // par défaut, la stratégie locale utilise le nom d'utilisateur et le mot de passe,
  // nous remplacerons le nom d'utilisateur par l'e-mail
  usernameField : 'email',
  passwordField : 'password',
  passReqToCallback : true // nous permet de renvoyer l'ensemble de la requête au
  callback
},
function(req, email, password, done) {
  let User = user;
  let isValidPassword = function(userpass,password){
    return bcrypt.compareSync(password, userpass);
  }
  User.findOne({ where : { emailId: email}}).then(function (user) {
    if (!user) {

```

```

        return done(null, false, { message: 'L\'e-mail n\'existe pas' });
    }
    if (!isValidPassword(user.password,password)) {
        return done(null, false, { message: 'Mot de passe incorrect.' });
    }
    let userinfo = user.get();
    return done(null,userinfo);
}).catch(function(err){
    console.log("Error:",err);
    return done(null, false, { message: 'Une erreur s\'est produite lors de votre
connexion' });
});
}
));
}

```

Tout d'abord, nous importons **bcrypt** dont nous avons besoin pour sécuriser les mots de passe. Ensuite, nous ajoutons un bloc **module.exports** qui prend l'objet **passport** et l'objet **user** comme argument. A l'intérieur de ce bloc, nous initialisons la stratégie **passport-local**, et le modèle utilisateur (**user**), qui seront passés en argument.

La stratégie **passport-local** est utilisée pour l'authentification avec un nom d'utilisateur et un mot de passe. Ce module vous permet de vous authentifier à l'aide d'un nom d'utilisateur et d'un mot de passe dans vos applications Node.js. En se connectant à **Passport**, l'authentification locale peut être intégrée facilement et discrètement dans votre application. La stratégie nécessite un rappel de vérification (**verify**), qui accepte ces informations d'identification et les appels effectués (**done**) en fournissant un utilisateur.

Q Search for Strategies

21,074

passport-local

build passing coverage 98% maintainability C 404 badge not found

Passport strategy for authenticating with a username and password.

This module lets you authenticate using a username and password in your Node.js applications. By plugging into Passport, local authentication can be easily and unobtrusively integrated into any application or framework that supports [Connect](#)-style middleware, including [Express](#).

Install

\$ npm install passport-local

Usage

Configure Strategy

The local authentication strategy authenticates users using a username and password. The strategy requires a `verify` callback, which accepts these credentials and calls `done` providing a user.

npm install passport-local

1.0.0 published a month ago

112,273 downloads in the last day

603,610 downloads in the last week

2,645,228 downloads in the last month

Comprendre la sérialisation et la désérialisation “passport”

L'identifiant utilisateur – user id - (que vous fournissez comme deuxième argument de la fonction **done**) est enregistré dans la session et est ensuite utilisé pour récupérer l'ensemble de l'objet via la fonction **deserializeUser**.

serializeUser détermine quelles données de l'objet utilisateur doivent être stockées dans la **session**. Le résultat de la méthode **serializeUser** est attaché à la **session** en tant que **req.session.passport.user = {}**. Ici, par exemple, ce serait (car nous fournissons l'identifiant de l'utilisateur comme clé) **req.session.passport.user = {id: 'xyz'}**

Le premier argument de **deserializeUser** correspond à la clé de l'objet utilisateur qui a été donnée à la fonction **done**. Ainsi, tout votre objet est récupéré à l'aide de cette clé. Cette clé ici est l'identifiant de l'utilisateur (la clé peut être n'importe quelle clé de l'objet utilisateur, c'est-à-dire le nom, l'e-mail, etc.). Dans **deserializeUser**, cette clé est mise en correspondance avec la base de données ou toute ressource de données. L'objet récupéré est attaché à l'objet de requête en tant que **req.user**.

```

passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, function(err, user) {
    done(err, user);
  });
});

```

Diagram illustrating the flow of user data between Passport.js functions and the request object:

- The `serializeUser` function is called, passing the user object and a `done` callback.
- The `done` callback is called with `user.id`, which is then saved to the session: `req.session.passport.user = {id: '...'}`.
- The `deserializeUser` function is called, passing the session ID and a `done` callback.
- The `done` callback is called with the user object, which is then attached to the request: `req.user`.

Nous avons ajouté deux stratégies dans **passport.js**. La première stratégie est **"local-signup"** pour créer un nouvel utilisateur au cas où l'e-mail n'existerait pas déjà.

La deuxième stratégie est **"local-signin"**. Dans cette stratégie, la fonction **isValidPassword** compare le mot de passe saisi avec la méthode de comparaison **bcrypt** puisque nous avons stocké notre mot de passe avec **bcrypt**.

Nous devons maintenant créer une page d'accueil qui apparaît lorsque l'utilisateur se connecte avec succès.

\$ touch views/home.hbs

Ajoutez ce qui suit à **home.hbs**:

```

<!DOCTYPE html>
<html>
<head>
  <title>Passport avec Sequelize</title>
</head>
<body>
<h2>Home</h2>
<h5>Vous êtes connecté.</h5>
</body>
</html>

```

Nous devons maintenant mettre à jour le routeur (**auth.js**). Remplacez le contenu de **auth.js** par ce qui suit:

```
const authController = require('../controllers/authcontroller.js');
module.exports = function(app,passport){
  app.get('/signup', authController.signup);
  app.get('/signin', authController.signin);
  app.post('/signup', passport.authenticate('local-signup', { successRedirect: '/home',
    failureRedirect: '/signup'}
  ));
  app.get('/home',isLoggedIn, authController.home);
  app.get('/logout',authController.logout);
  app.post('/signin', passport.authenticate('local-signin', { successRedirect: '/home',
    failureRedirect: '/signin'}
  ));
  function isLoggedIn(req, res, next) {
    if (req.isAuthenticated())
      return next();
    res.redirect('/signin');
  }
}
```

- successRedirect ==> Si l'inscription ou la connexion a réussi, le contenu de la page "successRedirect" apparaîtra
- failureRedirect ==> En cas d'échec de l'inscription ou de la connexion, le contenu de la page «failureRedirect» apparaîtra
- isLoggedIn ==> Cette fonction vérifiera si l'utilisateur est déjà connecté en utilisant la session

Maintenant, nous devons mettre à jour le contrôleur. Remplacez le contenu de **authcontroller.js** par ce qui suit:

```
exports.signup = function(req, res) {
  res.render('signup');
}

exports.signin = function(req, res) {
  res.render('signin');
}

exports.home = function(req,res){
  res.render('home');
}

exports.logout = function(req,res){
  req.session.destroy(function(err) {
    res.redirect('/signin');
  });
}
```

Nous allons maintenant importer la stratégie dans **server.js**:

```
server.js x
32
33 app.get('/', function(req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) {
34   res.send( body: 'Welcome to Passport with Sequelize');
35 });
36
37 //Models
38 const models = require("./models");
39 //Routes
40 const authRoute = require('./routes/auth.js')(app,passport);
41
42 //charger les stratégies passport
43 require('./config/passport/passport.js')(passport,models.user);
44
45
46 //Sync Database
47 models.sequelize.sync().then(function() {
48   console.log('La base de données fonctionne bien')
49 }).catch(function(err) {
50   console.log(err, "Quelque chose s'est mal passé avec la mise à jour de la base de données!")
51 });
52
53 app.listen( port: 3000, hostname: function(err) {
54   if (!err)
55     console.log("Le serveur écoute sur le port 3000");
56   else console.log(err)
57 });
```

```
//charger les stratégies passport
require('./config/passport/passport.js')(passport,models.user);
```

Maintenant, lancez l'application (\$ **node server.js**) et essayez de vous inscrire, puis connectez-vous. Vous devriez pouvoir vous connecter avec l'un des détails que vous avez utilisés lors de votre inscription, et vous serez dirigé vers <http://localhost:3000/home/>.

Lorsque vous souscrivez un nouvel utilisateur, vous remarquerez que le mot de passe dans la table users est maintenant haché et non affiché en texte brut.

```
mysql> select * from users;
+-----+-----+-----+-----+-----+
| id | firstName | lastName | emailId | password |
+-----+-----+-----+-----+-----+
| 142 | Bill | Gates | billgates@microsoft.com | $2b$08$b10ft7eWaqUhj5YOrfkINe7IRJs91wmn0SdNhifI7qUtCqDeNaMoq |
+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

localhost:3000/signup

Email Address Firstname Lastname Password

MySQL:

Showing rows 0 - 0 (1 total, Query took 0.0002 seconds.)

```
SELECT * FROM `users`
```

☐ Profiling [[Edit inline](#)] [[Edit](#)] [[Explain SQL](#)] [[Create PHP code](#)] [[Refresh](#)]

☐ Show all | Number of rows: Filter rows:

+ Options

	id	firstName	lastName	emailId	password
☐ Edit Copy Delete	1	Joseph	Azar	joseph.azar@univ-fcomte.fr	\$2b\$08\$IIIRSpJD5/TNj0J4rwB1xF.oRDEv0mQaYdS1uygKbq8g...

PostgreSQL:

Object Explorer | Dashboard | Properties | SQL | Statistics | Dependencies | Dependents | Processes | bd_passportlocal/josephazar@localhost*

bd_passportlocal/josephazar@localhost

Query | Query History

```
1 select * from users;
```

Data Output | Messages | Notifications

	id	firstName	lastName	emailId	password
	[PK] integer	character varying (255)	character varying (255)	character varying (255)	character varying (255)
1	1	Joseph	Azar	joseph.azar@univ-fcomte.fr	\$2b\$08\$ZzbaCUDe1LP5pmiADX5peOx.eowOaxRLEaztixb
2					

localhost:3000/signin

Email Address Password

Home

Vous êtes connecté.