**PUBLISHED: NOVEMBER 28 2018**
**LAST UPDATED: JULY 02 2020**

# Node.js - Role Based Authorization Tutorial with Example API

Tutorial built with **Node.js**

Other versions available:

- **ASP.NET:** ASP.NET Core 3.1 (/post/2019/10/16/aspnet-core-3-role-based-authorization-tutorial-with-example-api), ASP.NET Core 2.2 (/post/2019/01/08/aspnet-core-22-role-based-authorization-tutorial-with-example-api)

In this tutorial we'll go through a simple example of how to implement role based authorization / access control in a Node.js API with JavaScript. The example builds on another tutorial I posted recently which focuses on JWT authentication in Node.js (/post/2018/08/06/nodejs-jwt-authentication-tutorial-with-example-api), this version has been extended to include role based authorization / access control on top of the JWT authentication.

The example API has just three endpoints / routes to demonstrate authentication and role based authorization:

- `/users/authenticate` - public route that accepts HTTP POST requests with username and password in the body. If the username and password are correct then a JWT authentication token is returned.
- `/users` - secure route restricted to "Admin" users only, it accepts HTTP GET requests and returns a list of all users if the HTTP Authorization header contains a valid JWT token and the user is in the "Admin" role. If there is no auth token, the token is invalid or the user is not in the "Admin" role then a 401 Unauthorized response is returned.
- `/users/:id` - secure route restricted to authenticated users in any role, it accepts HTTP GET requests and returns the user record for the specified "id" parameter if authorization is successful. Note that "Admin" users can access all user records, while other roles (e.g. "User") can only access their own user record.

The tutorial project is available on GitHub at https://github.com/cornflourblue/node-role-based-authorization-api (https://github.com/cornflourblue/node-role-based-authorization-api).

Update History:

- 02 Jul 2020 - Updated to express-jwt version 6.0.0 to fix security vulnerability
- 15 May 2020 - Added instructions to run the Node.js api with an Angular 9 client app
- 28 Nov 2018 - Built with Node.js

## Running the Node.js Role Based Authorization API Locally

1. Download or clone the tutorial project code from https://github.com/cornflourblue/node-role-based-authorization-api (https://github.com/cornflourblue/node-role-based-authorization-api)
2. Install all required npm packages by running `npm install` from the command line in the project root folder (where the package.json is located).
3. Start the api by running `npm start` from the command line in the project root folder, you should see the message `Server listening on port 4000`. You can test the api directly using an application such as Postman (https://www.getpostman.com/) or you can test it with one of the single page example applications below.

## Running an Angular 9 client app with the Node.js Role Based Auth API

For full details about the example Angular 9 application see the post Angular 9 - Role Based Authorization Tutorial with Example (/post/2020/05/15/angular-9-role-based-authorization-tutorial-with-example). But to get up and running quickly just follow the below steps.

1. Download or clone the Angular 9 tutorial code from https://github.com/cornflourblue/angular-9-role-based-authorization-example (https://github.com/cornflourblue/angular-9-role-based-authorization-example)
2. Install all required npm packages by running `npm install` from the command line in the project root folder (where the package.json is located).
3. Remove or comment out the line below the comment `// provider used to create fake backend` located in the `/src/app/app.module.ts` file.
4. Start the application by running `npm start` from the command line in the project root folder, this will launch a browser displaying the Angular example application and it should be hooked up with the Node.js Role Based Authorization API that you already have running.

## Running a React client app with the Node.js Role Based Auth API

For full details about the example React application see the post React - Role Based Authorization Tutorial with Example (/post/2019/02/01/react-role-based-authorization-tutorial-with-example). But to get up and running quickly just follow the below steps.

1. Download or clone the React tutorial code from https://github.com/cornflourblue/react-role-based-authorization-example (https://github.com/cornflourblue/react-role-based-authorization-example)
2. Install all required npm packages by running `npm install` from the command line in the project root folder (where the package.json is located).
3. Remove or comment out the 2 lines below the comment `// setup fake backend` located in the `/src/index.jsx` file.
4. Start the application by running `npm start` from the command line in the project root folder, this will launch a browser displaying the React example application and it should be hooked up with the Node.js Role Based Authorization API that you already have running.

## Running a Vue.js client app with the Node.js Role Based Auth API

For full details about the example Vue.js application see the post Vue.js - Role Based Authorization Tutorial with Example (/post/2019/03/08/vuejs-role-based-authorization-tutorial-with-example). But to get up and running quickly just follow the below steps.

1. Download or clone the Vue.js tutorial code from https://github.com/cornflourblue/vue-role-based-authorization-example (https://github.com/cornflourblue/vue-role-based-authorization-example)
2. Install all required npm packages by running `npm install` from the command line in the project root folder (where the package.json is located).
3. Remove or comment out the 2 lines below the comment `// setup fake backend` located in the `/src/index.js` file.
4. Start the application by running `npm start` from the command line in the project root folder, this will launch a browser displaying the Vue.js example application and it should be hooked up with the Node.js Role Based Authorization API that you already have running.

## Node.js Role Based Access Control Project Structure

The project is structured into "feature folders" (users) "non-feature / shared component folders" (_helpers). Shared component folders contain code that can be used by multiple features and other parts of the application, and are prefixed with an underscore to group them together so it's easy to see what's what at a glance.

The example only contains a single *users* feature, but other features can be added pretty easily by copying the users folder and following the same pattern.

- _helpers
  - authorize.js
  - error-handler.js
  - role.js
- users
  - user.service.js
  - users.controller.js
- config.json
- server.js

## Node.js Auth Helpers Folder

**Path: /_helpers**

The helpers folder contains all the bits and pieces that don't fit into other folders but don't justify having a folder of their own.

Back to top

## Node.js Authorize Role Middleware

**Path: /_helpers/authorize.js**

The authorize middleware can be added to any route to restrict access to authenticated users within specified roles. If the roles parameter is left blank then the route will be restricted to any authenticated user regardless of role. It's used in the users controller to restrict access to the "get all users" and "get user by id" routes.

The authorize function actually returns 2 middleware functions, the first ( `jwt({ ... })` ) authenticates the request by validating the JWT token in the *Authorization* http request header. On successful authentication a `user` object is attached to the `req` object that contains the data from the JWT token, which in this case includes the user id ( `req.user.sub` ) and user role ( `req.user.role` ). The `sub` property is short for subject and is the standard JWT property for storing the id of the item in the token.

The second middleware function checks that the authenticated user is authorized to access the requested route based on their role. If either authentication or authorization fails then a 401 Unauthorized response is returned.

```
const jwt = require('express-jwt');
const { secret } = require('config.json');

module.exports = authorize;

function authorize(roles = []) {
    // roles param can be a single role string (e.g. Role.User or 'User')
    // or an array of roles (e.g. [Role.Admin, Role.User] or ['Admin', 'User'])
    if (typeof roles === 'string') {
        roles = [roles];
    }

    return [
        // authenticate JWT token and attach user to request object (req.user)
        jwt({ secret, algorithms: ['HS256'] }),

        // authorize based on user role
        (req, res, next) => {
            if (roles.length && !roles.includes(req.user.role)) {
                // user's role is not authorized
                return res.status(401).json({ message: 'Unauthorized' });
            }

            // authentication and authorization successful
            next();
        }
    ];
}
```

Back to top

## Node.js Auth Global Error Handler Middleware

**Path: /_helpers/error-handler.js**

The global error handler is used catch all errors and remove the need for redundant error handler code throughout the application. It's configured as middleware in the main server.js file.

```
module.exports = errorHandler;

function errorHandler(err, req, res, next) {
    if (typeof (err) === 'string') {
        // custom application error
        return res.status(400).json({ message: err });
    }

    if (err.name === 'UnauthorizedError') {
        // jwt authentication error
        return res.status(401).json({ message: 'Invalid Token' });
    }

    // default to 500 server error
    return res.status(500).json({ message: err.message });
}
```

Back to top

## Node.js Auth Role Object / Enum

**Path: /_helpers/role.js**

The role object defines the all the roles in the example application, I created it to use like an enum (https://en.wikipedia.org/wiki/Enumerated_type) to avoid passing roles around as strings, so instead of `'Admin'` we can use `Role.Admin`.

```
module.exports = {
  Admin: 'Admin',
  User: 'User'
}
```

Back to top

## Node.js Auth Users Folder

**Path: /users**

The users folder contains all code that is specific to the users feature of the role based authorization api.

## Node.js Auth User Service

**Path: /users/user.service.js**

The user service contains a method for authenticating user credentials and returning a JWT token, a method for getting all users in the application, and a method for getting a single user by id.

I hardcoded the array of users in the example to keep it focused on authentication and role based authorization, however in a production application it is recommended to store user records in a database with hashed passwords. I've posted another slightly different example (includes registration but excludes role based authorization) that stores data in MongoDB if you're interested in seeing how that's configured, you can check it out at NodeJS + MongoDB - Simple API for Authentication, Registration and User Management (/post/2018/06/14/nodejs-mongodb-simple-api-for-authentication-registration-and-user-management).

Near the top of the file (below the hardcoded users) I've got the exported service method definitions so it's easy to see all methods at a glance, and below that the rest of the file contains the method implementations.

```
const config = require('config.json');
const jwt = require('jsonwebtoken');
const Role = require('_helpers/role');

// users hardcoded for simplicity, store in a db for production applications
const users = [
    { id: 1, username: 'admin', password: 'admin', firstName: 'Admin', lastName: 'User', role: Role.Admin },
    { id: 2, username: 'user', password: 'user', firstName: 'Normal', lastName: 'User', role: Role.User }
];

module.exports = {
    authenticate,
    getAll,
    getById
};

async function authenticate({ username, password }) {
    const user = users.find(u => u.username === username && u.password === password);
    if (user) {
        const token = jwt.sign({ sub: user.id, role: user.role }, config.secret);
        const { password, ...userWithoutPassword } = user;
        return {
            ...userWithoutPassword,
            token
        };
    }
}

async function getAll() {
    return users.map(u => {
        const { password, ...userWithoutPassword } = u;
        return userWithoutPassword;
    });
}

async function getById(id) {
    const user = users.find(u => u.id === parseInt(id));
    if (!user) return;
    const { password, ...userWithoutPassword } = user;
    return userWithoutPassword;
}
```

Back to top

## Node.js Auth Users Controller

**Path: /users/users.controller.js**

The users controller defines all user routes for the api, the route definitions are grouped together at the top of the file and the route implementations are below.

Routes that use the authorize middleware are restricted to authenticated users, if the role is included (e.g. `authorize(Role.Admin)` ) then the route is restricted to users in the specified role / roles, otherwise if the role is not included (e.g. `authorize()` ) then the route is restricted to all authenticated users regardless of role. Routes that don't use the authorize middleware are publicly accessible.

The `getById` route contains some extra custom authorization logic within the route function. It allows admin users to access any user record, but only allows normal users to access their own record.

Express is the web server used by the api, it's one of the most popular web application frameworks for Node.js.

```
const express = require('express');
const router = express.Router();
const userService = require('./user.service');
const authorize = require('_helpers/authorize')
const Role = require('_helpers/role');

// routes
router.post('/authenticate', authenticate);     // public route
router.get('/', authorize(Role.Admin), getAll); // admin only
router.get('/:id', authorize(), getById);       // all authenticated users
module.exports = router;

function authenticate(req, res, next) {
    userService.authenticate(req.body)
        .then(user => user ? res.json(user) : res.status(400).json({ message: 'Username or password is incorrect' }))
        .catch(err => next(err));
}

function getAll(req, res, next) {
    userService.getAll()
        .then(users => res.json(users))
        .catch(err => next(err));
}

function getById(req, res, next) {
    const currentUser = req.user;
    const id = parseInt(req.params.id);

    // only allow admins to access other user records
    if (id !== currentUser.sub && currentUser.role !== Role.Admin) {
        return res.status(401).json({ message: 'Unauthorized' });
    }

    userService.getById(req.params.id)
        .then(user => user ? res.json(user) : res.sendStatus(404))
        .catch(err => next(err));
}
```

Back to top

## Node.js Auth App Config

**Path: /config.json**

The app config file contains configuration data for the api.

IMPORTANT: The "secret" property is used by the api to sign and verify JWT tokens for authentication, update it with your own random string to ensure nobody else can generate a JWT to gain unauthorised access to your application.

```
{
    "secret": "THIS IS USED TO SIGN AND VERIFY JWT TOKENS, REPLACE IT WITH YOUR OWN SECRET, IT CAN BE ANY STRING"
}
```

Back to top

## Node.js Auth Main Server Entrypoint

**Path: /server.js**

The server.js file is the entry point into the api, it configures application middleware, binds controllers to routes and starts the Express web server for the api.

```
require('rootpath')();
const express = require('express');
const app = express();
const cors = require('cors');
const bodyParser = require('body-parser');
const errorHandler = require('_helpers/error-handler');

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(cors());

// api routes
app.use('/users', require('./users/users.controller'));

// global error handler
app.use(errorHandler);

// start server
const port = process.env.NODE_ENV === 'production' ? 80 : 4000;
const server = app.listen(port, function () {
    console.log('Server listening on port ' + port);
});
```

Back to top