Node.js Server & Authentication Basics: Express, Sessions, Passport, and cURL

Evan Gow · Follow
21 min read · Oct 30, 2017

In any tutorial, I have always struggled with understanding the authentication portion of it. Instead of actually explaining the mechanics and what's going on, I just feel like the author is simply providing a walkthrough of how to copy/paste from the docs. This tutorial/explainer is meant to actually walk you through the authentication process and explain each mechanism.

This tutorial assumes some familiarity with the terminal/command-line interface (CLI) and Javascript / Node.js.

Step 1) Set up the file structure

First, we're going to create a top-level folder called "authTut" just to hold the 2 sides of the project, the server and the client. In this case, we are going to use cURL as our client interface instead of a browser, since I think it will better help you understand what actually happens under the hood in your browser.

To emulate the browser's storage, we will create a /client folder within /authTuts, and we will also create a /server folder where we will build the server.

Run the following commands in your terminal.

```
workspace $ mkdir authTut
workspace $ cd authTut
```

```
authTut $ mkdir server
authTut $ mkdir client
```

Step 2) Initialize npm and install express in the /server folder

First, we change into our /server folder, then initialize npm, so we can keep track of what dependencies our server has. Next, go ahead and install express as a dependency, then create a server.js file.

```
authTut $ cd server
server $ npm init -y
server $ npm install express --save
server $ touch server.js
```

Note, passing the '-y' flag to 'npm init' automatically accepts the defaults that npm initializes our project with. After calling this function, you should see the defaults options logged to the console.

At this point you should have a folder/file structure that looks like the following:

```
- /authTuts
  - /server
    - /node_modules
    - server.js
    - package.json
  - /client
```

Step 3) Create the server and run it

Open the /authTuts folder in your favorite text editor, then open the authTuts/server/server.js file.

First we are going to require the express module, then we call the express() function to create our app, and lastly we tell express which port to run on.

```
//npm modules
const express = require('express');

// create the server
const app = express();

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-1.md hosted with ❤ by GitHub view raw

Next, call 'node server.js' from the terminal. This will start up our server. You should get the response "Listening on localhost:3000". (The 'node' command in our terminal can be used to run Javascript files).

```
server $ node server.js
Listening on localhost:3000
```

Awesome! You've just created a server! If you go to "http://localhost:3000/" right now, you should see an error message saying "Cannot GET /", but that's way better than getting a "This site can't be reached" error!

Step 4) Add our homepage route at '/'

Update the server.js file to add the GET method to our '/' route. When the client (browser or cURL as we will soon see) calls the GET method, our server will respond with data. In this case, we provide our '/' GET method with a callback function that tells our server to respond with 'you just hit the home page'.

```
//npm modules
const express = require('express');

// create the server
const app = express();

// create the homepage route at '/'
app.get('/', (req, res) => {
  res.send('you just hit the home page\n')
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-2.md hosted with ❤ by GitHub view raw

The 'req' and 'res' parameters handed to our app.get('/') callback function are the 'request' and 'response' objects generated from the request headers that came in.

If you go to http://localhost:3000/ now, you will still see the 'Cannot GET /' error, because our old file is still acting as the server. We need to go to ahead and restart the server after saving our changes. We can do this by pressing 'control C' while in the terminal and then running 'node server.js' again.

```
Listening on localhost:3000
^C
server $ node server.js
Listening on localhost:3000
```

Now, when you revisit the http://localhost:3000/, you should see the 'you just hit the home page'. Open up a new tab or window in your terminal and change into the /client folder. Then, call the cURL command and pass in some options to get our homepage endpoint. You should see our response returned.

```
server $ cd ..
authTut $ cd client
client $ cURL -X GET http://localhost:3000/
you just hit the home page
```

The above makes use of the '-X' option we can pass curl to GET or POST to an endpoint. Here, we are 'getting' our '/' endpoint.

Step 5) Add nodemon

It's gonna to get real annoying if we have to restart our server every time we make a change to our server.js file. Let's use the nodemon module, which will automatically restart our server every time we save a change to the server.js file. First, install the nodemon package globally.

```
$ npm install -g nodemon
```

When you pass in the -g option to the npm module installer, it installs the package globally so you can access that module from anywhere in your file system when you're in the terminal. Note, I didn't show which folder you call the above from,

because it doesn't matter. Now, let's shut our server down and start it using nodemon.

```
^C
server $ nodemon server.js
```

Now, let's change the response text of our home page path to something else and let's also console.log() the request object, so we can see what it looks like. After you save the file, you should see the server restart in the server terminal tab.

```
//npm modules
const express = require('express');

// create the server
const app = express();

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log(req)
  res.send('You hit the home page without restarting the server automatically\n')
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-3.md hosted with ❤ by GitHub view raw

Now, let's call curl again, except this time, let's also pass it the -v flag (for 'verbose').

```
client $ curl -X GET http://localhost:3000 -v
Note: Unnecessary use of -X or --request, GET is already inferred.
* Rebuilt URL to: http://localhost:3000/
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/html; charset=utf-8
< Content-Length: 66
< ETag: W/"42-Ybeup68SVZ+Id3fqVh25rCkXfno"
```

```
< Date: Sun, 29 Oct 2017 19:58:38 GMT
< Connection: keep-alive
<
You hit the home page without restarting the server automatically
```

In the 2nd request, we get information on our curl request. Let's walk through it.

1. cURL is tells us we don't need to pass the -X GET as that is the default for cURL. I wanted to be explicit with this tutorial however.
2. "rebuilt URL to..." let's you know cURL added a slash at the end of the URL.
3. "Trying ::1..." is the IP address that the URL resolved to.
4. The next line is the port we connected to, which you notice is the port we specified when we created the server.
5. > indicates data cURL has sent to the server.
6. < indicates data cURL has received from server.
7. Lastly, you see the response text that the server sent

If you flip over to the terminal tab where the server is running, you should see a really long output. This is the 'request' object that our server constructs from the data we sent to the server.

Step 6) Install uuid to automatically generate unique strings

Now, open up a 3rd terminal tab or window and in the server folder and install the uuid module, which helps us to generate random strings. (Opening up the 3rd tab will allow us to install packages for our server without stopping the current server process. When we include new modules in our server.js file, nodemon will automatically restart and be able to pull these modules in.)

```
server $ npm install --save uuid
```

Then we add it to our server file and update our response text to send it to the client. Note, I am using string interpolation below, which requires using 'back-ticks' instead of quote marks. It's these (``) not these ('). (Probably near the top-left on your keyboard.)

```
//npm modules
const express = require('express');
const uuid = require('uuid').v4

// create the server
const app = express();

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log(req)
  const uniqueId = uuid()
  res.send(`Hit home page. Received the unique id: ${uniqueId}\n`)
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-4.md hosted with ❤️ by GitHub view raw

Now call cURL again.

```
client $ curl -X GET http://localhost:3000
Hit home page. Received the unique id: 044e0263-58b7-4c7f-a032-
056cd81069e3
```

Step 7) Add and configure express-session

Install express-session. This middleware handles session generation as express doesn't automatically do this.

```
server $ npm install express-session --save
```

Once it's installed, let's modify our server.js file in the following ways:

1. Include express-session
2. Add/configure our app to use the session middleware with a unique session id we generate. We will log the request.sessionID object before and after the middleware is used.
3. Remove the id we generated/sent to the client

Note, in the session configuration below, I am leaving the 'secret' as 'keyboard cat', but in production you would want to replace this with a randomly generated string

that's pulled from an environment variable.

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')

// create the server
const app = express();

// add & configure middleware
app.use(session({
  genid: (req) => {
    console.log('Inside the session middleware')
    console.log(req.sessionID)
    return uuid() // use UUIDs for session IDs
  },
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log('Inside the homepage callback function')
  console.log(req.sessionID)
  res.send(`You hit home page!\n`)
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-5.md hosted with ❤️ by GitHub view raw

Now, let's call the curl request again with the -v flag.

```
curl -X GET http://localhost:3000 -v
...
< set-cookie: connect.sid=s%3A5199f3ed-3f5a-4478-aed7-
fab9ce6ca378.DjQlJ%2F1t%2F00RAfIs5yW6CEsVUXM25aMclq7VGzxVnoY;
Path=/; HttpOnly
...
```

I've abbreviated the response above, but you can see that in the data being returned from the server (indicated by the < symbol) that we are setting the session ID to a random string. If we flip over to our server logs we should see the following:

```
Inside the session middleware
undefined
```

```
Inside the homepage callback function
5199f3ed-3f5a-4478-aed7-fab9ce6ca378
```

When we logged the req.sessionID inside the session middleware, the session hadn't been instantiated yet, so we hadn't yet added the sessionID to the request object. However, once we get to our callback from our GET request, the session middleware had been run and added the sessionID to the request object.

Try calling the cURL function a few more times. You'll see that a new session id is generated each time. **Browsers will automatically save/send the session id** and send it in each request to the server; however, **cURL doesn't** automatically save our session ID and send it in the request headers. Let's fix that. Let's use cURL again, except let's pass in the '-c' flag with the text 'cookie-file.txt'.

```
client $ curl -X GET http://localhost:3000 -c cookie-file.txt
```

This creates a text file in our /client folder called 'cookie-file.txt'. You should see this text file appear in your project. Now we can call curl again, but this time calling cookie-file.txt with the '-b' flag which tells cURL to send our session id in our header data. Let's also add the '-v' flag to see this.

```
curl -X GET http://localhost:3000 -b cookie-file.txt -v
...
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.54.0
> Accept: */*
> Cookie: connect.sid=s%3A de59a40f-6737-4b8d-98bf-
bf2bb8495118 .e0pWTi2w8%2FAAOKxKgKDBdu99JnspruYSEgLSV3tvxX4
...
```

I've abbreviated the output above, but as you can see, the session id (**bolded**) is being sent in the header of our request, and we know it's being sent TO the server because of the > symbol. Try calling this function as many times as you like. If you flip over to the server logs, you should see that the same session id is being output to the console each time. You may also notice, we don't see the 'Inside the session middleware' log being made. This is because our 'genid' function isn't called since the id is already being taken in.

Alright, I'm expecting one problem now. Let's try restarting our server.

In the terminal tab where the server is running press 'control C' then start it back up with nodemon.

```
Inside the homepage callback function
de59a40f-6737-4b8d-98bf-bf2bb8495118
^C
server $ nodemon server.js
Listening on localhost:3000
```

From the client folder, call the cURL command again.

```
client $ curl -X GET http://localhost:3000 -b cookie-file.txt
```

Then look back at the server logs again.

```
Listening on localhost:3000
Inside the session middleware
de59a40f-6737-4b8d-98bf-bf2bb8495118
Inside the homepage callback function
ac656d2a-9796-4560-9dbf-73996a1853f8
```

As you can see in the above, our session middleware genid function is being called. This is because the session was being stored in the server's memory. So when we restarted the server, the session id was wiped along with the rest of the memory. Here's the breakdown.

1. Server is restarted and session memory is wiped.
2. We send our cURL request to the server along with our session id
3. The server receives the requests, and the session middleware can't find the session id in memory, so it call the genid function
4. The genid function logs that we are inside the session middleware and it logs the request object's session id. Since we sent the session id in our cURL request, the

request object was actually instantiated with that session id. However, this session id is overwritten by the return value of the genid function.

5. When the session middleware is done overwriting the session id we sent, control is handed over to the callback function within app.get(), where we log that we are inside the hompage callback function and log the new id.

Let's make that curl request one more time from our client folder.

```
client $ curl -X GET http://localhost:3000 -b cookie-file.txt
```

Look at the server logs again.

```
Inside the session middleware
de59a40f-6737-4b8d-98bf-bf2bb8495118
Inside the homepage callback function
b02aa920-7031-427f-ac2e-b82f21140002
```

Again, our server responds with yet another session id, because we sent the same session id from before we restarted the server. We need to call our curl request again, but this time pass the '-c' flag so we overwrite our existing session info.

```
client $ curl -X GET http://localhost:3000 -c cookie-file.txt
```

Back to the server logs.

```
Inside the session middleware
undefined
Inside the homepage callback function
74f37795-6fcf-4300-beb9-3de41395eafe
```

The req.sessionID isn't defined, because we didn't send the session info in our curl request. Let's take a look at our cookie-file.txt. Sure enough, there's the session id that was generated and sent back.

```
...
#HttpOnly_localhost FALSE / FALSE 0 connect.sid s%3A74f37795-6fcf-
4300-beb9-3de41395eafe.5mblOCOvpwAMh7bNuTZ9qyloG5UOcIczep5GjMnVEi8
```

Now, if we call our curl request with the '-b' flag again. We won't see the 'Inside the session middle' log being called, because genid isn't being called. The session id is being matched with the session id in memory.

We still haven't solved the problem though. If we restart our server again, the memory will be wiped again. So we need to have some way of making sure that we can save our session id even if the server shuts down. That's where the 'session store' comes in. Normally, your database would act as a session store, but since we're trying to keep things as simple as possible, let's just store our session info in text files.

If you go to the express docs, you will see that there are a number of npm packages that are provided to act as the glue between your database and the session middleware. We're going to use the one called 'session-file-store.' As usual, let's install it.

```
server $ npm install session-file-store --save
```

Now, let's add it to our server.js file.

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')
const FileStore = require('session-file-store')(session);

// create the server
const app = express();

// add & configure middleware
app.use(session({
  genid: (req) => {
    console.log('Inside the session middleware')
    console.log(req.sessionID)
    return uuid() // use UUIDs for session IDs
  },
  store: new FileStore(),
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log('Inside the homepage callback function')
  console.log(req.sessionID)
  res.send(`You hit home page!\n`)
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-6.md hosted with ❤️ by GitHub view raw

Notice in the above, that we are calling the session variable when we require the FileStore. Then, we add an instance to the FileStore to our session configuration.

We also need to do one other thing. When we use the the ‘session-file-store’ module, by default, it creates a new ‘/sessions’ directory when it is first called. The first time and each subsequent time that we create a new session, the module creates a new file for the session info in the /sessions folder. Since we import the session-file-store in server.js and the session-file-store depends on the /sessions folder, nodemon will restart the server each time we create a new session.

We can tell nodemon to ignore a file or directory by calling ‘— ignore’ and passing it the file or directory name.

```
server $ nodemon --ignore sessions/ server.js
```

This will be annoying to remember if you ever come back to this project again and want to figure out how to run the server. Let's make it easy on ourselves by adding it to our npm scripts in the package.json file.

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev:server": "nodemon --ignore sessions/ server.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.2",
    "express-session": "^1.15.6",
    "session-file-store": "^1.1.2",
    "uuid": "^3.1.0"
  }
}
```

package-json-1.md hosted with ❤️ by GitHub view raw

If you haven't already, go ahead and kill the current nodemon process and call the dev:server script we just added.

```
^C
server $ npm run dev:server
```

Now let's create call our cURL command and create a new cookie file that will be saved to the client.

```
client $ curl -X GET http://localhost:3000 -c cookie-file.txt
```

Your cookie-file.txt should now have a new session id saved in it. If you ignore the 's%3A' bit of it, the rest before the '.' should match the name of the new file saved in the /sessions folder.

Now restart the server again.

```
^C
server $ npm run dev:server
```

Then call the cURL command passing in the cookie-file.txt with the '-b' flag this time, so that it sends the session id we created before we restarted the server.

```
client $ curl -X GET http://localhost:3000 -b cookie-file.txt
```

Try calling it as much as you like. You should get the same session id output by server every time. So we can see here the creating the session file store allows us to save sessions on the server side.

If you've got this far, congrats! We're done with the sessions piece of this tutorial! On to authentication!

Authentication

Step 1) Add login endpoints

Alright! Let's get to it! First, we're going to add a login route to our application with both a GET and POST method. Note that in the post method, we are calling 'req.body'. This should log the data that we send to the server in our POST request.

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')
const FileStore = require('session-file-store')(session);

// create the server
const app = express();

// add & configure middleware
app.use(session({
  genid: (req) => {
    console.log('Inside the session middleware')
    console.log(req.sessionID)
    return uuid() // use UUIDs for session IDs
  },
  store: new FileStore(),
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log('Inside the homepage callback function')
  console.log(req.sessionID)
  res.send(`You got home page!\n`)
})

// create the login get and post routes
app.get('/login', (req, res) => {
  console.log('Inside GET /login callback function')
  console.log(req.sessionID)
  res.send(`You got the login page!\n`)
})

app.post('/login', (req, res) => {
  console.log('Inside POST /login callback function')
  console.log(req.body)
  res.send(`You posted to the login page!\n`)
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-7.md hosted with ❤️ by GitHub view raw

Step 2) Configure Express to be able to read the POST data

In our client, let's write a new cURL command.

```
curl -X POST http://localhost:3000/login -b cookie-file.txt -H
'Content-Type: application/json' -d '{"email":"test@test.com",
"password":"password"}'
```

Quick note, if you're using a Windows machine, you will need to use double quotes and escape them with a back slash, like so:

```
curl -X POST  http://localhost:3000/login -b cookie-file.txt -H
"Content-Type: application/json" -d "{\"email\":\"test@test.com\",
\"password\":\"password\"}"
```

I will just be using single quotes for the rest of this article since it's easier to read. Just remember on Windows, you need to use double quotes and escape them.

Okay, let's get back to it. In the above, we have changed a few things.

1. We're now using -X POST instead of -X GET
2. We've added the -H flag to set the header content-type to application/json
3. We pass the -d flag in along with the data that we want to send. Note that it is wrapping in single quotes

Looking at our server output we find:

```
Inside POST /login callback function
undefined
```

It looks like the req.body is 'undefined'. The problem here is that Express doesn't actually know how to read the JSON content-type, so we need to add another middleware to do this. We can use the body-parser middleware to body parse the data and add it to the req.body property.

```
server $ npm install body-parser --save
```

Then require it in server.js and configure express to use it.

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')
const FileStore = require('session-file-store')(session);
const bodyParser = require('body-parser');

// create the server
const app = express();

// add & configure middleware
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
app.use(session({
  genid: (req) => {
    console.log('Inside the session middleware')
    console.log(req.sessionID)
    return uuid() // use UUIDs for session IDs
  },
  store: new FileStore(),
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log('Inside the homepage callback function')
  console.log(req.sessionID)
  res.send(`You got home page!\n`)
})

// create the login get and post routes
app.get('/login', (req, res) => {
  console.log('Inside GET /login callback function')
  console.log(req.sessionID)
  res.send(`You got the login page!\n`)
})

app.post('/login', (req, res) => {
  console.log('Inside POST /login callback function')
  console.log(req.body)
  res.send(`You posted to the login page!\n`)
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-8.md hosted with ❤️ by GitHub view raw

You'll notice in the above that when we configure our app to use the body-parser middleware, bodyParser.json() and bodyParser.urlencoded(). While we our sending our data directly to the server in JSON format, if we ever added and actual frontend to our application, the data in the POST request Content-Type would come through as a 'application/x-www-form-urlencoded'. Here, we're including it just in case you ever want to use this file as boilerplate for a new project.

Step 3) Add and configure Passport.js

Install the passport.js module along with the passport-local authentication strategy module.

```
server $ npm install passport passport-local --save
```

Before we get into the code, let's talk about the authentication flow.

1. The user is going to POST their login information to the /login route
2. We need to do something with that data. This is where passport comes in. We can call passport.authenticate('login strategy', callback(err, user, info)). This method takes 2 parameters. Our 'login strategy' which is 'local' in this case, since we will be authenticating with email and password (you can find a list of other login strategies using passport though. These include Facebook, Twitter, etc.) and a callback function giving us access to the user object if authentication is successful and an error object if not.
3. passport.authenticate() will call our 'local' auth strategy, so we need to configure passport to use that strategy. We can configure passport with passport.use(new strategyClass). Here we tell passport how the local strategy can be used to authenticate the user.
4. Inside the strategyClass declaration, we will take in the data from our POST request, use that to find the matching user in the database and check that the credentials match. If they do match, passport will add a login() method to our request object, and we return to our passport.authenticate() callback function.
5. Inside the passport.authenticate() callback function, we now call the req.login() method.
6. The req.login(user, callback()) method takes in the user object we just returned from our local strategy and calls passport.serializeUser(callback()). It takes that user object and 1) saves the *user id* to the **session file store** 2) saves the *user id* in the **request object as request.session.passport** and 3) adds the *user object* to the **request object as request.user**. Now, on subsequent requests to authorized routes, we can retrieve the user object without requiring the user to login again (by getting the id from the session file store and using that to get the user object from the database and adding it to our request object).

Alright! That probably seemed like a lot! Things should be more clear after looking at the code and the server logs we generate.

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')
const FileStore = require('session-file-store')(session);
const bodyParser = require('body-parser');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

const users = [
  {id: '2f24vvg', email: 'test@test.com', password: 'password'}
]

// configure passport.js to use the local strategy
passport.use(new LocalStrategy(
  { usernameField: 'email' },
  (email, password, done) => {
    console.log('Inside local strategy callback')
    // here is where you make a call to the database
    // to find the user based on their username or email address
    // for now, we'll just pretend we found that it was users[0]
    const user = users[0]
    if(email === user.email && password === user.password) {
      console.log('Local strategy returned true')
      return done(null, user)
    }
  }
));

// tell passport how to serialize the user
passport.serializeUser((user, done) => {
  console.log('Inside serializeUser callback. User id is save to the session file store here')
  done(null, user.id);
});

// create the server
const app = express();

// add & configure middleware
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
app.use(session({
  genid: (req) => {
    console.log('Inside session middleware genid function')
    console.log(`Request object sessionID from client: ${req.sessionID}`)
    return uuid() // use UUIDs for session IDs
  },
  store: new FileStore(),
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))
app.use(passport.initialize());
app.use(passport.session());

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log('Inside the homepage callback')
  console.log(req.sessionID)
  res.send(`You got home page!\n`)
})

// create the login get and post routes
app.get('/login', (req, res) => {
  console.log('Inside GET /login callback')
  console.log(req.sessionID)
  res.send(`You got the login page!\n`)
})
```

```
app.post('/login', (req, res, next) => {
  console.log('Inside POST /login callback')
  passport.authenticate('local', (err, user, info) => {
    console.log('Inside passport.authenticate() callback');
    console.log(`req.session.passport: ${JSON.stringify(req.session.passport)}`)
    console.log(`req.user: ${JSON.stringify(req.user)}`)
    req.login(user, (err) => {
      console.log('Inside req.login() callback')
      console.log(`req.session.passport: ${JSON.stringify(req.session.passport)}`)
      console.log(`req.user: ${JSON.stringify(req.user)}`)
      return res.send('You were authenticated & logged in!\n');
    })
  })(req, res, next);
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server.9.md hosted with ❤ by GitHub view raw

Okay. So it probably looks like we've added a bunch of code here, but if you take out the console.log()s and the additional comments, it's really much. Let's walk through it.

1. At the top of the file we are requiring passport and the passport-local strategy.
2. Going down to the middle of the file, we can see that we configure our application to use passport as a middleware with the calls to app.use(passport.initialize()) and app.use(passport.session()). Note, that we call this after we configure our app to use express-session and the session-file-store. This is because passport rides on top of these.
3. Going further down, we see our app.post('login') method immediately calls passport.authenticate() with the local strategy.
4. The local strategy is configured at the top of the file with passport.use(new LocalStrategy()). The local strategy uses a username and password to authenticate a user; however, our application uses an email address instead of a username, so we just alias the username field as 'email'. Then we tell the local strategy how to find the user in the database. Here, you would normally see something like 'DB.findById()' but for now we're just going to ignore that and assume the correct user is returned to us by calling our users array containing our single user object. Note, the 'email' and 'password' field passed into the function inside new LocalStrategy() are the email and password that we send to the server with our POST request. If the data we receive from the POST request matches the data we find in our database, we call the done(error object, user

object) method and pass in null and the user object returned from the database. (We will make sure to handle cases where the credential don't match shortly.)

5. After the done() method is called, we hop into to the passport.authenticate() callback function, where we pass the user object into the req.login() function (remember, the call to passport.authenticate() added the login() function to our request object). The req.login() function handles serializing the user id to the session store and inside our request object and also adds the user object to our request object.

6. Lastly, we respond to the user and tell them that they've been authenticated!

Let's try it out! Call the cURL request and send our login credentials to the server. Note, before doing the below I have deleted all the files that were stored in my /sessions directory and I am calling the POST request below with the '-c' flag to create/overwrite our cookie-file.txt in our client folder. So we're essentially starting clean.

```
client $ curl -X POST  http://localhost:3000/login -c cookie-
file.txt -H 'Content-Type: application/json' -d
'{"email":"test@test.com", "password":"password"}'

You were authenticated & logged in!
```

In the server logs, you should see something like the following.

```
Inside session middleware genid function
Request object sessionID from client: undefined
Inside POST /login callback
Inside local strategy callback
Local strategy returned true
Inside passport.authenticate() callback
req.session.passport: undefined
req.user: undefined
Inside serializeUser callback. User id is save to the session file
store here
Inside req.login() callback
req.session.passport: {"user":"2f24vvg"}
req.user:
{"id":"2f24vvg","email":"test@test.com","password":"password"}
```

As you can see in the above, before we call req.login(), the req.session.passport object and req.user object are undefined. After the req.login() function is called (i.e. once we're in the req.login() callback function), they are defined!

Step 4) Add a route that requires authorization

Let's add a route to our app that requires authorization. Note, now that the user is 'authenticated' (i.e. logged in), we can talk about 'authorization' which tells our server which routes require a user to be logged in before they can be visited.

Let's code it up!

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')
const FileStore = require('session-file-store')(session);
const bodyParser = require('body-parser');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

const users = [
  {id: '2f24vvg', email: 'test@test.com', password: 'password'}
]

// configure passport.js to use the local strategy
passport.use(new LocalStrategy(
  { usernameField: 'email' },
  (email, password, done) => {
    console.log('Inside local strategy callback')
    // here is where you make a call to the database
    // to find the user based on their username or email address
    // for now, we'll just pretend we found that it was users[0]
    const user = users[0]
    if(email === user.email && password === user.password) {
      console.log('Local strategy returned true')
      return done(null, user)
    }
  }
));

// tell passport how to serialize the user
passport.serializeUser((user, done) => {
  console.log('Inside serializeUser callback. User id is save to the session file store here')
  done(null, user.id);
});

passport.deserializeUser((id, done) => {
  console.log('Inside deserializeUser callback')
  console.log(`The user id passport saved in the session file store is: ${id}`)
  const user = users[0].id === id ? users[0] : false;
  done(null, user);
});

// create the server
const app = express();

// add & configure middleware
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
app.use(session({
  genid: (req) => {
    console.log('Inside session middleware genid function')
    console.log(`Request object sessionID from client: ${req.sessionID}`)
    return uuid() // use UUIDs for session IDs
  },
  store: new FileStore(),
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))
app.use(passport.initialize());
app.use(passport.session());

// create the homepage route at '/'
app.get('/', (req, res) => {
  console.log('Inside the homepage callback')
  console.log(req.sessionID)
  res.send(`You got home page!\n`)
})

// create the login get and post routes
```

```
// create the login get and post routes
app.get('/login', (req, res) => {
  console.log('Inside GET /login callback')
  console.log(req.sessionID)
  res.send(`You got the login page!\n`)
})

app.post('/login', (req, res, next) => {
  console.log('Inside POST /login callback')
  passport.authenticate('local', (err, user, info) => {
    console.log('Inside passport.authenticate() callback');
    console.log(`req.session.passport: ${JSON.stringify(req.session.passport)}`)
    console.log(`req.user: ${JSON.stringify(req.user)}`)
    req.login(user, (err) => {
      console.log('Inside req.login() callback')
      console.log(`req.session.passport: ${JSON.stringify(req.session.passport)}`)
      console.log(`req.user: ${JSON.stringify(req.user)}`)
      return res.send('You were authenticated & logged in!\n');
    })
  })(req, res, next);
})

app.get('/authrequired', (req, res) => {
  console.log('Inside GET /authrequired callback')
  console.log(`User authenticated? ${req.isAuthenticated()}`)
  if(req.isAuthenticated()) {
    res.send('you hit the authentication endpoint\n')
  } else {
    res.redirect('/')
  }
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

You can see in the above we've add a '/authrequired' route available via the get method which checks our request object to see if req.isAuthenticated() is true. This is function added to our request object by passport. Let's create a new session by going to the homepage, then let's use that new session to try going to the /authrequired route.

Note, in our second request below, we are passing curl the '-L' flag, which tell cURL to follow redirects.

```
client $ curl -X GET http://localhost:3000 -c cookie-file.txt
You got home page!

client $ curl -X GET http://localhost:3000/authrequired -b cookie-file.txt -L
You got home page!
```

Now, let's check the server logs.

```
#first request to the home page
Inside session middleware genid function
Request object sessionID from client: undefined
Inside the homepage callback
e6388389-0248-4c69-96d1-fda44fbc8839

#second request to the /authrequired route
Inside GET /authrequired callback
User authenticated? false
```

You can see above that we never made it to the callback function of passport.deserializeUser(), because we have not authenticated. Now, let's hit our login route again, and using our existing cookie-file.txt then hit the /authrequired route.

```
curl -X POST http://localhost:3000/login -b cookie-file.txt -H
'Content-Type: application/json' -d '{"email":"test@test.com",
"password":"password"}'
You were authenticated & logged in!

curl -X GET http://localhost:3000/authrequired -b cookie-file.txt -L
you hit the authentication endpoint
```

This time you can see we got back that we hit the authentication endpoint! Awesome! In the server logs we see:

```
Inside POST /login callback
Inside local strategy callback
Local strategy returned true
Inside passport.authenticate() callback
req.session.passport: undefined
req.user: undefined
Inside serializeUser callback. User id is save to the session file
store here
Inside req.login() callback
req.session.passport: {"user":"2f24vvg"}
req.user:
{"id":"2f24vvg","email":"test@test.com","password":"password"}
Inside deserializeUser callback
The user id passport saved in the session file store is: 2f24vvg
Inside GET /authrequired callback
User authenticated? true
```

The one new thing to point out here is that we got to the deserializeUser callback function, which matched our session id to the session-file-store and retrieved our user id.

Step 5) Hook up a database and handle incorrect credentials

This is gonna be another big step! First, let's create a another folder inside of authTuts called 'db', initialize npm, install json-server, and create a new db.json file.

```
authTuts $ mkdir db
authTuts $ cd db
db $ npm init -y
db $ npm install json-server --save
db $ touch db.json
```

Once json-server has installed, let's add a new "json:server" script to our package.json

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "json:server": "json-server --watch ./db.json --port 5000"
},
```

So what was the point of all that? json-server is a package that automatically sets up RESTful routes for data in the db.json file. Let's try it out. Copy/paste the following into your db.json file.

```
{
  "users": [
    {
      "id":"2f24vvg",
      "email": "test@test.com",
      "password": "password"
    },
    {
      "id":"d1u9nq",
      "email": "user2@example.com",
      "password": "password"
    }
  ]
}
```

db-1.md hosted with ❤ by GitHub view raw

Then call 'npm run json:server' from the /db folder.

```
db $ npm run json:server
```

Then open http://localhost:5000/users in your browser. You should see the JSON from our db.json file being output. Try hit a specific /users route: http://localhost:5000/users/2f24vvg. You should just see the id, email, and password for that one user. Let's try that again, but instead of passing the user id directly into the URL, let's pass an email address in as a query parameter to the URL: http://localhost:5000/users?email=user2@example.com. This time you should get our 2nd user's JSON object. Pretty cool, right?!

Leave the json-server running in it's own tab in the terminal, and let's flip back over to our terminal tab in the server folder (or open a new one if you need to), and let's install axios, a package that helps fetch data.

```
server $ npm install axios --save
```

In our LocalStrategy configuration, we're now going to fetch our user object from the /users REST endpoint using the email address as a query parameter (like we manually did before). While we're at it, let's also update our configuration to handle invalid user credentials or any errors that are returned by axios from the json-server.

In our passport.deserializeUser() function, let's return the user object by calling axios to retrieve the /users endpoint and passing in the user id in the path (i.e. /users/:id).

Let's also handle the various errors that could pop up during authentication in our passport.authenticate() callback function, and instead of simple telling the user that they have logged in, let's redirect the user to the /authrequired path.

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')
const FileStore = require('session-file-store')(session);
const bodyParser = require('body-parser');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const axios = require('axios');

// configure passport.js to use the local strategy
passport.use(new LocalStrategy(
  { usernameField: 'email' },
  (email, password, done) => {
    axios.get(`http://localhost:5000/users?email=${email}`)
    .then(res => {
      const user = res.data[0]
      if (!user) {
        return done(null, false, { message: 'Invalid credentials.\n' });
      }
      if (password != user.password) {
        return done(null, false, { message: 'Invalid credentials.\n' });
      }
      return done(null, user);
    })
    .catch(error => done(error));
  }
));

// tell passport how to serialize the user
passport.serializeUser((user, done) => {
  done(null, user.id);
});

passport.deserializeUser((id, done) => {
  axios.get(`http://localhost:5000/users/${id}`)
  .then(res => done(null, res.data) )
  .catch(error => done(error, false))
});

// create the server
const app = express();

// add & configure middleware
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
app.use(session({
  genid: (req) => {
    return uuid() // use UUIDs for session IDs
  },
  store: new FileStore(),
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))
app.use(passport.initialize());
app.use(passport.session());

// create the homepage route at '/'
app.get('/', (req, res) => {
  res.send(`You got home page!\n`)
})

// create the login get and post routes
app.get('/login', (req, res) => {
  res.send(`You got the login page!\n`)
})

app.post('/login', (req, res, next) => {
```

```
  passport.authenticate('local', (err, user, info) => {
    if(info) {return res.send(info.message)}
    if (err) { return next(err); }
    if (!user) { return res.redirect('/login'); }
    req.login(user, (err) => {
      if (err) { return next(err); }
      return res.redirect('/authrequired');
    })
  })(req, res, next);
})

app.get('/authrequired', (req, res) => {
  if(req.isAuthenticated()) {
    res.send('you hit the authentication endpoint\n')
  } else {
    res.redirect('/')
  }
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server_11.md hosted with ❤ by GitHub view raw

As you can see above, I've removed all of our server logging. Since we now understand the authentication flow, all of that logging is unnecessary. There's quite a bit of new code above, but I think it should be fairly easy to understand what's happening. We're most just adding 'if' statements to handle any errors.

Try hitting the login endpoint with a cURL POST request. Note, I've excluded the '-X POST' flag as we want cURL to follow the redirect from the /login route to the /authrequired route, which we GET. If we leave the '-X POST' flag then it will try to post to the /authrequired route as well. Instead, we'll just let cURL infer what it should do on each route.

```
client $ curl http://localhost:3000/login -c cookie-file.txt -H
'Content-Type: application/json' -d '{"email":"test@test.com",
"password":"password"}' -L
you hit the authentication endpoint
```

Step 6) Handling encrypted passwords

First, let's install bcrypt on our server.

```
server $ npm install bcrypt-nodejs --save
```

Now we require it in our server.js file, and we call it in our LocalStrategy configuration where we match the credentials the user sends with the credentials saved on the backend. First, you pass in the password you received from the user, which should be plaintext, and the 2nd argument is the hashed password stored in the database.

```
//npm modules
const express = require('express');
const uuid = require('uuid/v4')
const session = require('express-session')
const FileStore = require('session-file-store')(session);
const bodyParser = require('body-parser');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const axios = require('axios');
const bcrypt = require('bcrypt-nodejs');

// configure passport.js to use the local strategy
passport.use(new LocalStrategy(
  { usernameField: 'email' },
  (email, password, done) => {
    axios.get(`http://localhost:5000/users?email=${email}`)
    .then(res => {
      const user = res.data[0]
      if (!user) {
        return done(null, false, { message: 'Invalid credentials.\n' });
      }
      if (!bcrypt.compareSync(password, user.password)) {
        return done(null, false, { message: 'Invalid credentials.\n' });
      }
      return done(null, user);
    })
    .catch(error => done(error));
  }
));

// tell passport how to serialize the user
passport.serializeUser((user, done) => {
  done(null, user.id);
});

passport.deserializeUser((id, done) => {
  axios.get(`http://localhost:5000/users/${id}`)
  .then(res => done(null, res.data) )
  .catch(error => done(error, false))
});

// create the server
const app = express();

// add & configure middleware
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
app.use(session({
  genid: (req) => {
    return uuid() // use UUIDs for session IDs
  },
  store: new FileStore(),
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true
}))
app.use(passport.initialize());
app.use(passport.session());

// create the homepage route at '/'
app.get('/', (req, res) => {
  res.send(`You got home page!\n`)
})

// create the login get and post routes
app.get('/login', (req, res) => {
  res.send(`You got the login page!\n`)
})
```

```
app.post('/login', (req, res, next) => {
  passport.authenticate('local', (err, user, info) => {
    if(info) {return res.send(info.message)}
    if (err) { return next(err); }
    if (!user) { return res.redirect('/login'); }
    req.login(user, (err) => {
      if (err) { return next(err); }
      return res.redirect('/authrequired');
    })
  })(req, res, next);
})

app.get('/authrequired', (req, res) => {
  if(req.isAuthenticated()) {
    res.send('you hit the authentication endpoint\n')
  } else {
    res.redirect('/')
  }
})

// tell the server what port to listen on
app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

server-12.md hosted with ❤ by GitHub view raw

Now we just need to make sure we've stored hashed passwords in the database. You can use this tool to hash the word 'password' and store replace the existing password values in the 'db.json' file.

```
{
  "users": [
    {
      "id":"2f24vvg",
      "email": "test@test.com",
      "password": "$2a$12$nv9iV5/UNuV4Mdj1Jf8zfuUraqboSRtSQqCmtOc4F7rdwmOb9IzNu"
    },
    {
      "id":"d1u9nq",
      "email": "user2@example.com",
      "password": "$2a$12$VHZ9aJ5A87YeFop4xVW.aOMm95ClU.EviyT9o0i8HYLdG6w6ctMfW"
    }
  ]
}
```

db-2.md hosted with ❤ by GitHub view raw

Lastly, let's try logging in again.

```
client $ curl http://localhost:3000/login -c cookie-file.txt -H
'Content-Type: application/json' -d '{"email":"test@test.com",
"password":"password"}' -L
you hit the authentication endpoint
```

You did it!

Hopefully, you've learned a bit more about the following things:

- Express and how it uses middleware
- How session data is stored and retrieved both on the server and client
- Passport's authentication flow and how to use it for authorization as well
- How to use bcrypt to check plaintext against hashed passwords

I will leave adding the signup flow as an exercise to you. Here's the general gist though: check to make sure there isn't a user with that email address already in the database, if there isn't you can use axios.post() to store data in the db.json (make sure to hash the password with bcrypt), then call req.login() with the user object you've created.

And lastly, always refer to the docs if you're looking for more information!

JavaScript Authentication Nodejs

Follow

Written by Evan Gow

323 Followers

Building tools for writers

More from Evan Gow

Evan Gow

How to Build your MVP Using the Facebook Groups API & Deploy as a Browser Extension

The story of how and why I built the Facebook Writing Prompts Group to power the StoryOrigin Chrome Extension.

5 min read · Sep 29, 2017

Evan Gow

The Indie Author Book Launch Checklist

How to launch a new pen name and book series in 90 days

2 min read · May 11, 2018

91 1

Evan Gow

It's okay. Use the wrong tool for this job.

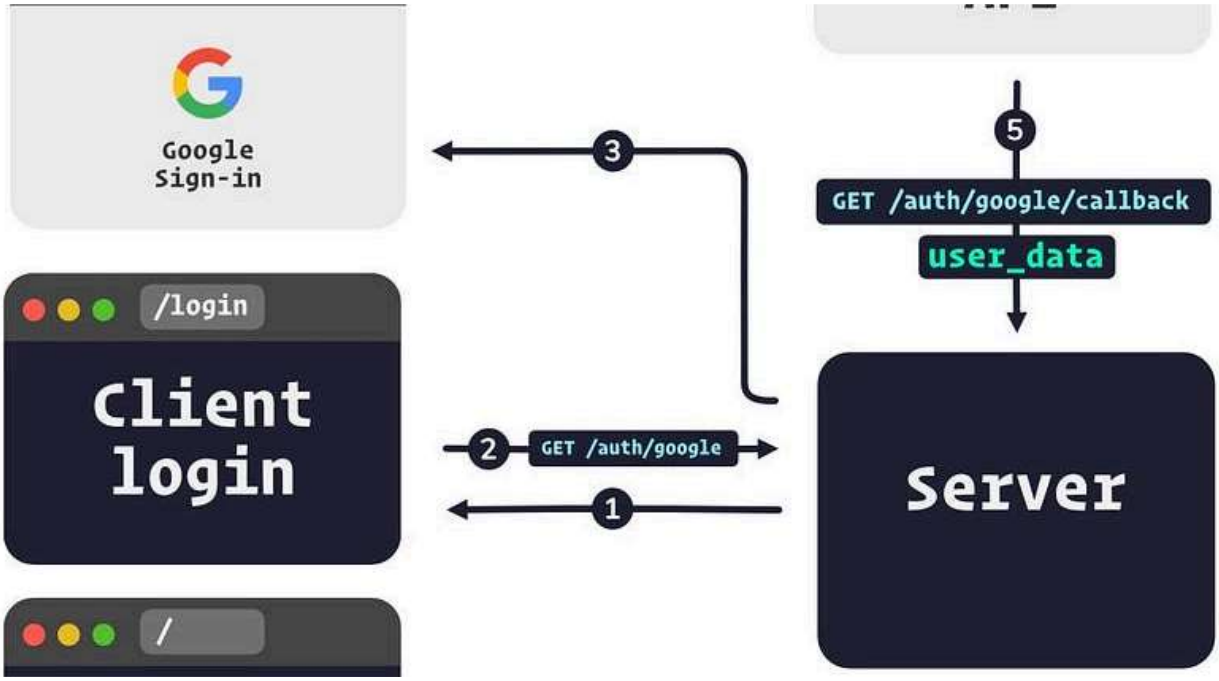
Can we stop all of this “use-the-right-tool-for-the-right-job,” nonsense?

2 min read · Nov 27, 2018

8

See all from Evan Gow

Recommended from Medium



Venkata Naveen Varma V

Google Login (Authentication using Oauth2, Passport, NodeJS)

We go through how to login/authenticate users using there google account with the help of passport, Oauth2.

2 min read · Nov 21, 2023

53

Dev Balaji

JWT Authentication in Node.js: A Practical Guide

Implementing authentication for a web application using Node.js typically involves several steps. “Token-based authentication” is a common...

3 min read · Sep 7, 2023

182 1

Lists

Stories to Help You Grow as a Software Developer

19 stories · 835 saves

General Coding Knowledge

20 stories · 935 saves

data science and AI

40 stories · 81 saves

Generative AI Recommended Reading

52 stories · 743 saves

MVP Catalyst

Building a Secure REST API in Node.js: Best Practices for Web Developers

Node.js + REST

4 min read · Nov 6, 2023

113 1

Mohit Gadhavi

Creating and Managing Cookies in Node.Js and React: A Comprehensive Guide

Cookies are small pieces of data stored by the user's web browser on the user's device. They serve a variety of purposes, and some of the...

3 min read · Nov 28, 2023

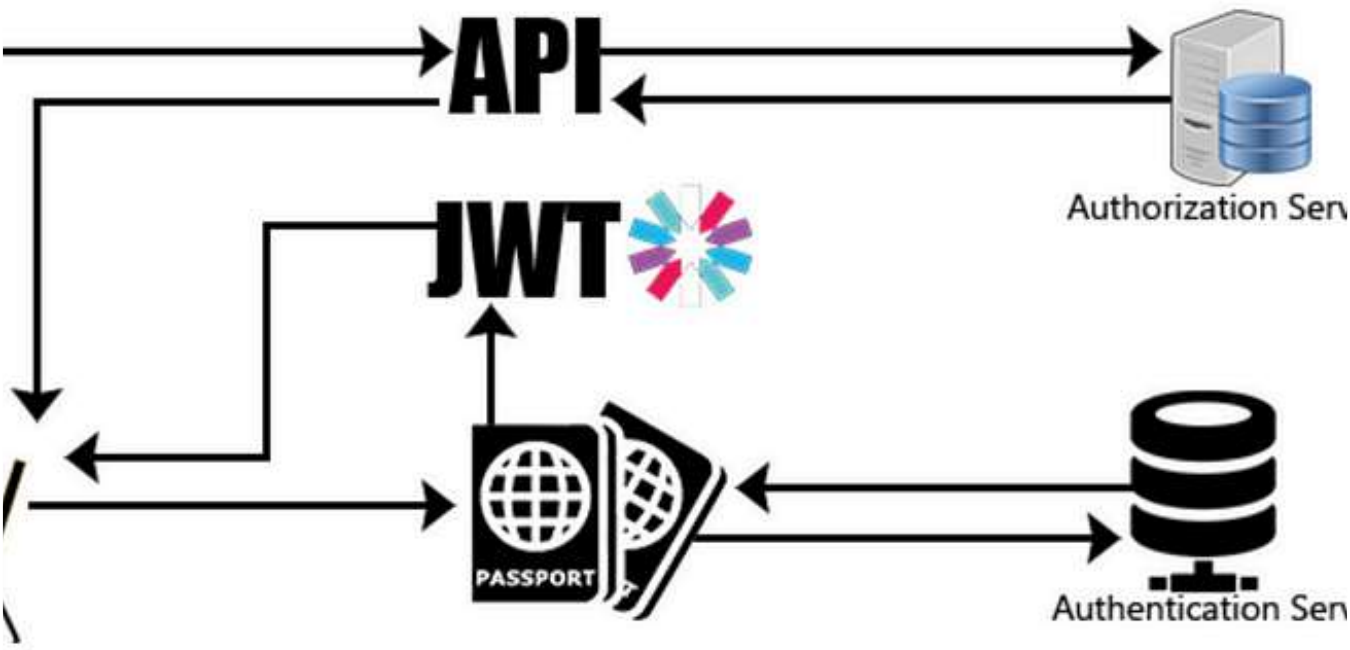
11 1

Mangesh Pal

Node.js JWT Authentication With HTTP Only Cookie (by mangesh Pal)

3 min read · Sep 5, 2023

8



Pawan Kumar

Authentication and Authorization in Node.js: A Comprehensive Guide

Authentication and authorization are two fundamental concepts in web application security. They ensure that users have the right level of...

3 min read · Oct 4, 2023

115 1