How to make an OAuth 2 server with Node.js

~~Now this is a story all about how My life got flipped-turned upside down~~ If you found yourself here hoping to find a clear and granular explanation of how to make an oAuth 2 Node.js server then you, my friend, are in luck.

Throughout this tutorial I will explain both the how, what and why of how to setup the oAuth 2 server and the components it relies upon.

Lets begin by detailing the ingredients of the programmatic recipe we're going to cook-up in order to serve *clients* with our *server*.

Here are the main components that we require and what they will be broadly used for:

- **Express**: We will use this to create the Node.js server and route the server's endpoints. By the end of this tutorial we will have infused our oAuth 2 components into our express server as middleware.
- **MySql** — This will be used to store the bearer tokens (don't worry, I'll explain these later) and the users' details. I will provide the implementation of the database layer of this project on GitHub with a generous scoop of comments to show you what's going on. But, I won't explicitly go into how it's implemented here to make the tutorial more database agnostic and focused on the oAuth implementation.
- **node-oauth2-server** — This is the library that we will be using to piece together the oAuth 2 system. Unfortunately, the official documentation currently lacks any decent official examples. Luckily the good folks dwelling in the issues section gave me some that I used when I put together the server for the first time. This is my attempt to lower the bar of knowledge required to clearly understand how the library works.
- **A model/object** that contains the methods that the **node-oauth2-server** library requires to create the oAuth 2 system.
- **body-parser** — A library that can be used as middleware to help parse JSON in the body of post requests.

A Warning

Before we get any further into this tutorial let me make one warning: the oAuth 2 server that we're going to make is intended for client facing applications and will only handle authentication using the password grant type. This means it will only be useful if you want to authenticate users by means of a username and password. We also won't be using client secrets because they are very vulnerable in client facing applications due to sneaky hackers willing to reverse engineer your software and take those kinds of credentials.

A Broad Overview

I'm going to give you a brief outline of what oAuth 2 is and how the system is going to work so you have a sense where we're going as we're building it.

oAuth2 authentication is a token based authentication system whereby clients initially trade a set of valid credentials for a **bearer token**. Then, when the client makes a request to an endpoint, it can provide this token to the server in the request's header to prove that they have been authorised to use the endpoint in question.

For the system to function we need:

1. An endpoint to register users.
2. An endpoint to provide registered users with a bearer token in exchange for valid credentials.
3. A way of validating bearer tokens.
4. A way of assigning this validation system to endpoints.

We will cover all 4 of those things.

Server & Database Setup

First, we're going to setup the server using express, give express the body-parser to use as middleware, and instantiate MySql database related helpers. If you want to follow along with the tutorial then create a file called **index.js** in a folder of your choice.

I'll guide you through what's going on in the code below:

Login & Registration Part 1

Ok, now we've completed our initial setup. Good job. You did great. Now we'll create the routes to the register and login endpoints.

Next, create a folder called '**authorisation**' in the same directory as the index.js file, that is, if you want to keep up. Now create a file called **'authRouter.js'** in that folder.

At this point your folder structure should look like this:
```
rootFolder/->index.js
          /authorisation
                        ->authRouter.js
```

In this file we're going to put the register and login routes. There are a few objects in it which I haven't explained yet so don't be alarmed when you see them.

Here's the code for the file:

The router object is created from the express object that we required in the index.js file and we will inject it into this module from there. We'll update index.js to setup the injection of this, the expressApp and the authRoutesMethods variable later in this tutorial.

We haven't made the file from which the authRoutesMethods variable is spawned yet, but, don't worry, we're going to craft it now. So toward that end create a file in the authorisation folder called **'authRoutesMethods.js'.** The projects structure should now looks like this:
```
rootFolder/->index.js
          /authorisation
                        ->authRouter.js
                        ->authRoutesMethods.js
```

Before you read the next script and get angry that the promises haven't been split into separate methods you should know that I kept them together so you could read through them without scrolling up and down the page.

You're welcome.

Now, this file will contain the register method that will be used by the above router's register endpoint. You can see this method being accessed in the authRouter.js file we just made.

Now, let me walk you through the code using comments:

Whoah. I'm seriously ~~drunk~~ impressed that you made it this far into the article. You're doing great. Next we're going to return to the index.js and add the authRoutesMethods module that we just made, and the authRoutes module that we made right before that.

Creating The Model

Now we're going to pour the gravy over this dish; it's time to create the module that contains all of the functions that the node-oauth2-server library requires. **The object in this module which contains all of the functions that the node-oauth2-server library needs is called a model**. So, lets stroll onto our binary based catwalk and let our model pout unashamedly.

Firstly, create a file called '**accessTokenModel.js'** in the authorisation folder. Your project structure should now looks like this:

```
rootFolder/->index.js
          /authorisation
                        ->authRouter.js
                        ->accessTokenModel.js
```

To make things clearer as I walk you through the code in accessTokenModel.js just bear in mind that all this module does is compose how authorisation requests should be handled by the framework. The framework itself is only interested in seeing if a user's bearer token is valid, and providing users with valid credentials bearer tokens. Everything this model and the library does is centred around achieving this, for those ends are the core of what oAuth 2 is about: token based authorisation.

Here is a basic representation of how the client gets its hands on a bearer token:

1. The client sends up a username, password and password grantType in a request.
2. The framework attempts to get retrieve the client application that is trying to authenticate a user. This is handled by *getClient().*

3. Then the library makes sure the client is allowed to use the grantType it's requestint to use. This is handled by *grantTypeAllowed().*

4. The library then tries to get the user from the database layer you chose. This is handled by *getUser().*

5. The user we tried to retrieve is then passed to the *saveAccessToken() where the bearerToken is saved for that user.*

And here's how it works when you're making a request that requires you to be authenticated:

1. The client makes a request to use an endpoint which is auth protected (you'll see how to do this later) and sends its bearer token in the request's post body.
2. The model's *getAccessToken() method is called and validates the bearer token. If it's valid then the user can access the authorisation protected endpoint. If it's not valid then, I'm sorry, but you have to stop programming now. Sorry.*

Anyways, here is the code dump of the file with a plentiful helping of comments to explain everything in more detail — enjoy:

Well done if you made it this far! There's not much left do now.

Next, we're going jump back into the index.js file and instantiate this model and setup the **node-oauth2-server** library**.** In order to use the oAuth 2 library we have to assign an instance of it to the oauth property of the expressApp. This will allow us to use the oath property to auth protect endpoints of our choosing.

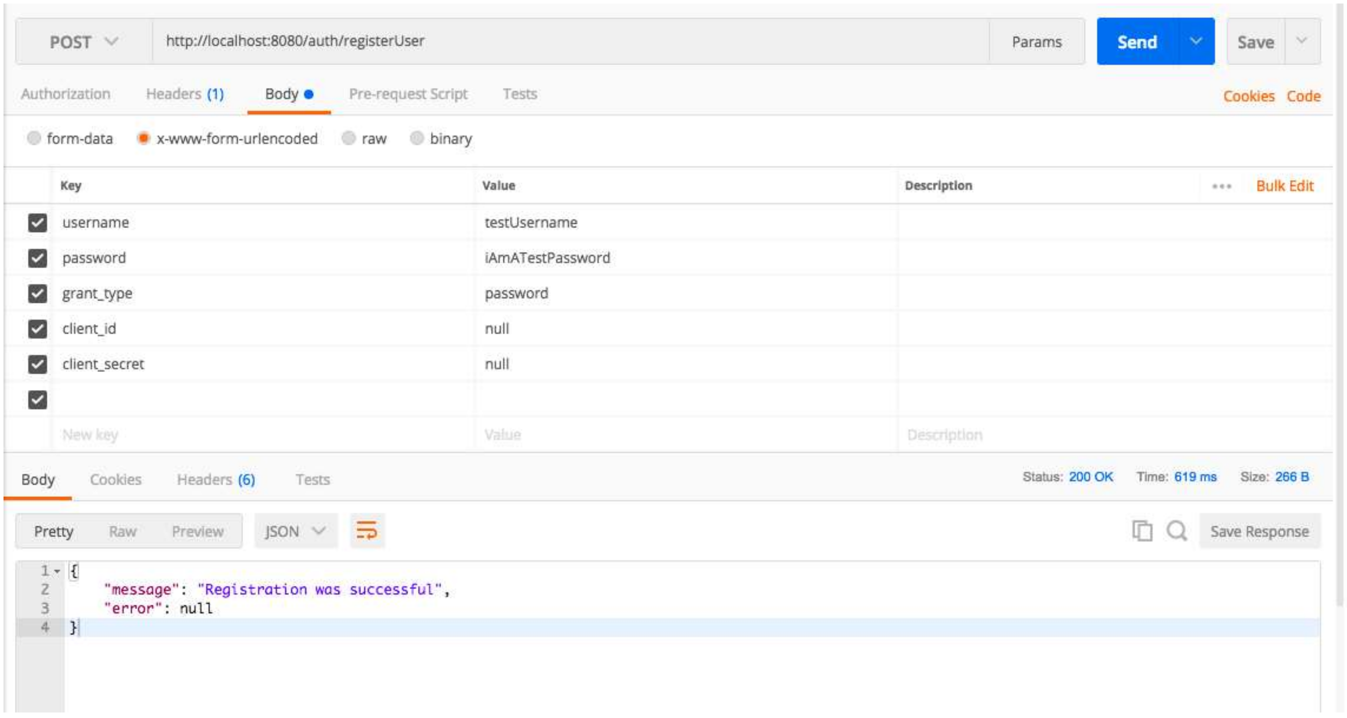
Here is some code:

Login & Registration Part 2

Now we've required the oAuth library, created an instance of it and assigned our model to it we can actually start to: auth protect some endpoints, register users and get them bearer tokens when they log in.

Here's an example of how to register and login a user:

1. Register them using the registerUser endpoint we made:



POST http://localhost:8080/auth/registerUser

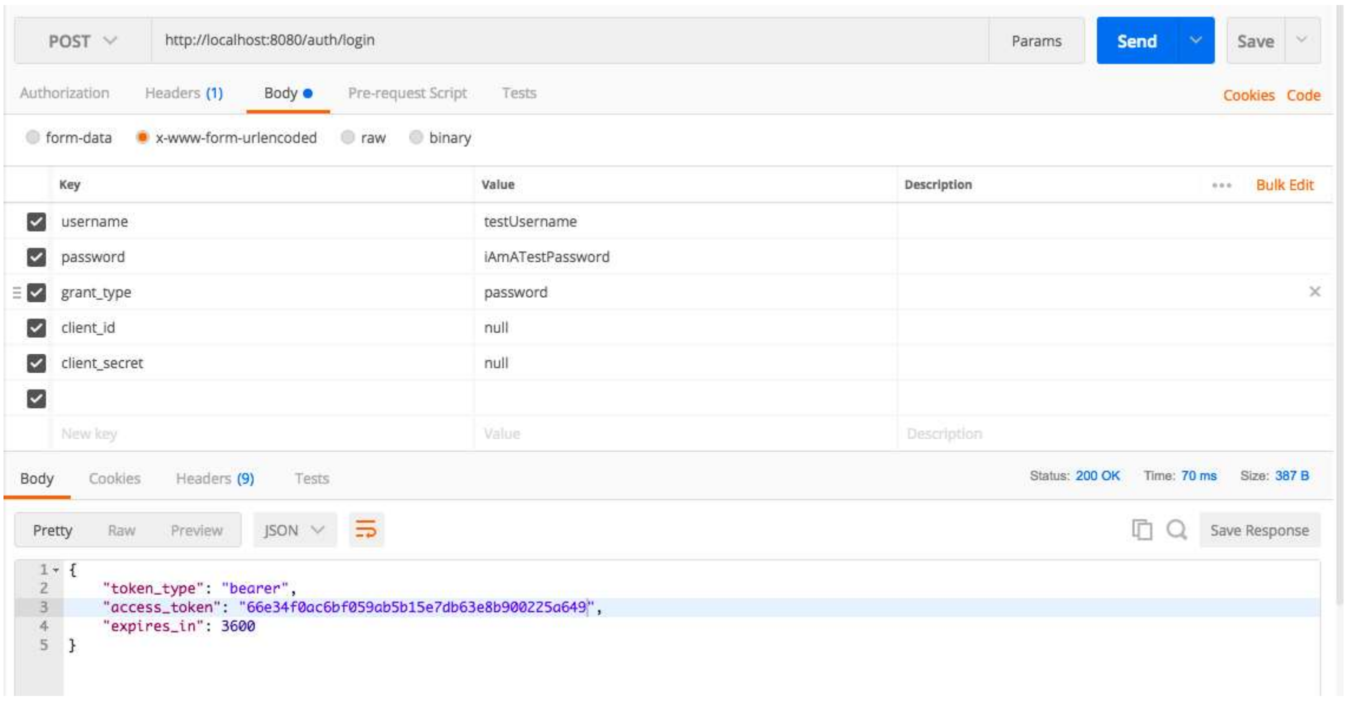
Body: x-www-form-urlencoded

Key	Value	Description
username	testUsername	
password	iAmATestPassword	
grant_type	password	
client_id	null	
client_secret	null	

Status: 200 OK Time: 619 ms Size: 266 B

```
{
    "message": "Registration was successful",
    "error": null
}
```

2. Log them in using the login endpoint we created:



http://localhost:8080/auth/login

Key	Value	Description
username	testUsername	
password	iAmATestPassword	
grant_type	password	
client_id	null	
client_secret	null	

Status: 200 OK Time: 70 ms Size: 387 B

```
{
    "token_type": "bearer",
    "access_token": "66e34f0ac6bf059ab5b15e7db63e8b900225a649",
    "expires_in": 3600
}
```

It’s important to emphasise the whole response here is generated by the node-oauth2-server as a result of:

1. The model we implemented successfully validating the user’s credentials. 2. The expressApp.**oauth**.grant**()** being infused in the login endpoint and both generating and sending back the bearer token upon the client being successfully validated.

Applying Auth Protection

In order to auth protect an endpoint all you have to do is create the endpoint using the expressApp or the express.Router, and then call expressApp.**oauth**.authorise**()** as the second parameter in the route declaration. The second parameters is between the route and endpoint function you pass in.

Applying it looks like this:

```
router.post(‘/myEndpoint’, expressApp.oauth.authorise(),   myEndpointFunction)
```

Whenever someone tries to call the ‘/myEndpoint\ endpoint they will need to provide a valid bearer token.

To provide credentials to an auth protected endpoint you must:

1. Add ‘Authorization’ as a key to your requests header.
2. Add the bearer token as the value to the Authorzation key. The bearer token must be preceded by the word ‘Bearer’. It looks like this:

```
Bearer 1e1b22895b2516dfbc0a7434ee4f5678297eb353
```

3. ~~Take a couple of hits of crack (optional).~~

And that’s it! All you need is that header and your request will be authenticated as long as the token is valid.

The End?

So, at this point we can say bon voy’age as you’ve got all you need to know to make your own username name and password based oAuth 2 server.

However, if you’d like to see a concrete example of how to auth protect an endpoint then you may read on and I will outline one.

A Farewell Example

Firstly create a new folder in the route directory of the project called ‘**restrictedArea**’. Now create one file in there called ‘**restrictedAreaRoutes.js**’. In this file I’m just going to define a route to an auth protected endpoint.

Here’s the code:

Awesome job! Now we’re going to create the restrictedAreaRoutesMethods object you saw being used in the gist above. All this object does is provide a method that will handle the requests made to the ‘/enter’ endpoint.

At this point you’re project structure should look like this:

```
rootFolder/->index.js
          /authorisation
                        ->authRouter.js
                        ->accessTokenModel.js/restrictedArea
                       ->restrictedAreaRoutes.js
                       ->restrictedAreaRoutesMethods.js
```

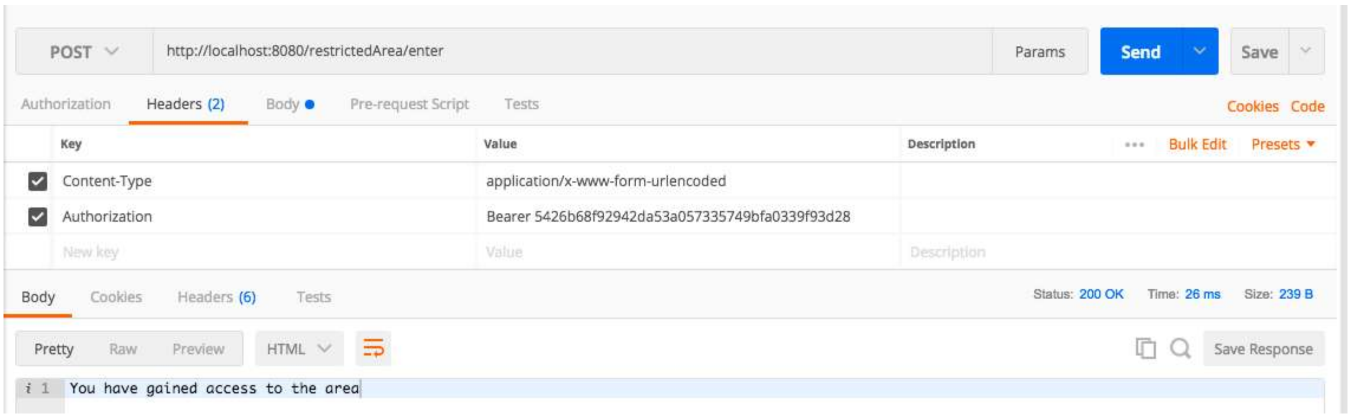
If it does look like that then you’ve done a pretty fantastic job. Well done. You’re amazing etc.

The final step before testing the endpoint is to assign the router in the restrictedAreaRoutes.js module to the expressApp in the index.js file. What we’re going to do is let that router handle all the requests whose route url path starts with /**restrictedArea.** To understand this more look up the express.Router object, but basically all it means all requests whose route path begins with /restrictedArea/ will be handled by that router.

Here’s the code:

You did it! We finished writing all of the code for this tutorial. All there’s left to do is test it out.

Here’s a screenshot of how your header should look and the response you should get when you call the **restrictedArea/enter/** endpoint. Of course, you should be using your own bearer token — but you already knew that!



http://localhost:8080/restrictedArea/enter

Key	Value	Description
Content-Type	application/x-www-form-urlencoded	
Authorization	Bearer 5426b68f92942da53a057335749bfa0339f93d28	

Status: 200 OK Time: 26 ms Size: 239 B

```
You have gained access to the area
```

The End!

Ok, so, that’s it! You’ve reached the end of this very long tutorial. Congratulations for making it this far.

Now go forth and auth protect all of your security conscious endpoints!

And feel free to leave a comment below if you have any questions or suggestions.

P.S. you can find all of the code including the database layer I used in the tutorial here: https://github.com/Meeks91/nodeJS_OAuth2Example