Get My Latest Articles

Email address
Subscribe

Beer Locker: Building a RESTful API With Node - OAuth2 Server

Jul 2nd, 2014

Welcome to part 4 of the Beer Locker series

1. Getting started
2. CRUD
3. Passport
4. OAuth2 Server
5. Digest
6. Username & Password

In our previous article we ended wtih a functional API capable of creating user accounts, locking down API endpoints, and only allowing access to a user's own beer locker.

In this part we will dive into creating an OAuth2 server and allowing access to API endpoints for the authorized user or authorized applications. We will do this by integrating OAuth2orize into our application.

Security

I realized I wasn't explicitly clear about what steps ones should take in regards to security. This article was meant more on how to get an OAuth2 server up and running. When implementing an OAuth2 server you MUST make sure to secure your application. This means running all OAuth2 endpoints over HTTPS and hashing the client secret, authorization code, and access token. All three of those values should be treated the same way you would a password for a user account. If you are unsure about how best to secure your applications, you should seek out the assistance of someone who does.

Application Client

The first thing we need to do is add a new model, controller, and endpoints to allow us to create new application clients. An application client is what would request access to a user account. Perhaps something like a service that wants to help manage your beer collection to notify you when you are running low.

Create a new file called `client.js` in the `models` directory and add the following code to it.

```
1  // Load required packages
2  var mongoose = require('mongoose');
3
4  // Define our client schema
5  var ClientSchema = new mongoose.Schema({
6    name: { type: String, unique: true, required: true },
7    id: { type: String, required: true },
8    secret: { type: String, required: true },
9    userId: { type: String, required: true }
10 });
11
12 // Export the Mongoose model
13 module.exports = mongoose.model('Client', ClientSchema);
```

There isn't too much going on here that differs from what we already did in previous articles. We have a name to help identify the application client. The id and secret are used as part of the OAuth2 flow and should always be kept secret. In this post we aren't adding any encryption, but it would be a good practice to hash the secret at the very least. Finally we have a userId field to identify which user owns this application client.

You could also consider auto generating the client id and secret in order to enforce uniqueness, randomness, and strength.

The next thing we will add is the controller to facilitate adding and viewing application clients. Create a new file called `client.js` in the `controllers` directory and add the following code to it.

```
1  // Load required packages
2  var Client = require('../models/client');
3
4  // Create endpoint /api/client for POST
5  exports.postClients = function(req, res) {
6    // Create a new instance of the Client model
7    var client = new Client();
8
9    // Set the client properties that came from the POST data
10   client.name = req.body.name;
11   client.id = req.body.id;
```

```
  client.secret = req.body.secret;
  client.userId = req.user._id;

  // Save the client and check for errors
  client.save(function(err) {
    if (err)
      res.send(err);

    res.json({ message: 'Client added to the locker!', data: client });
  });
};

// Create endpoint /api/clients for GET
exports.getClients = function(req, res) {
  // Use the Client model to find all clients
  Client.find({ userId: req.user._id }, function(err, clients) {
    if (err)
      res.send(err);

    res.json(clients);
  });
};
```


These two methods will allow us to create new application clients and get all existing ones for the authenticated user.

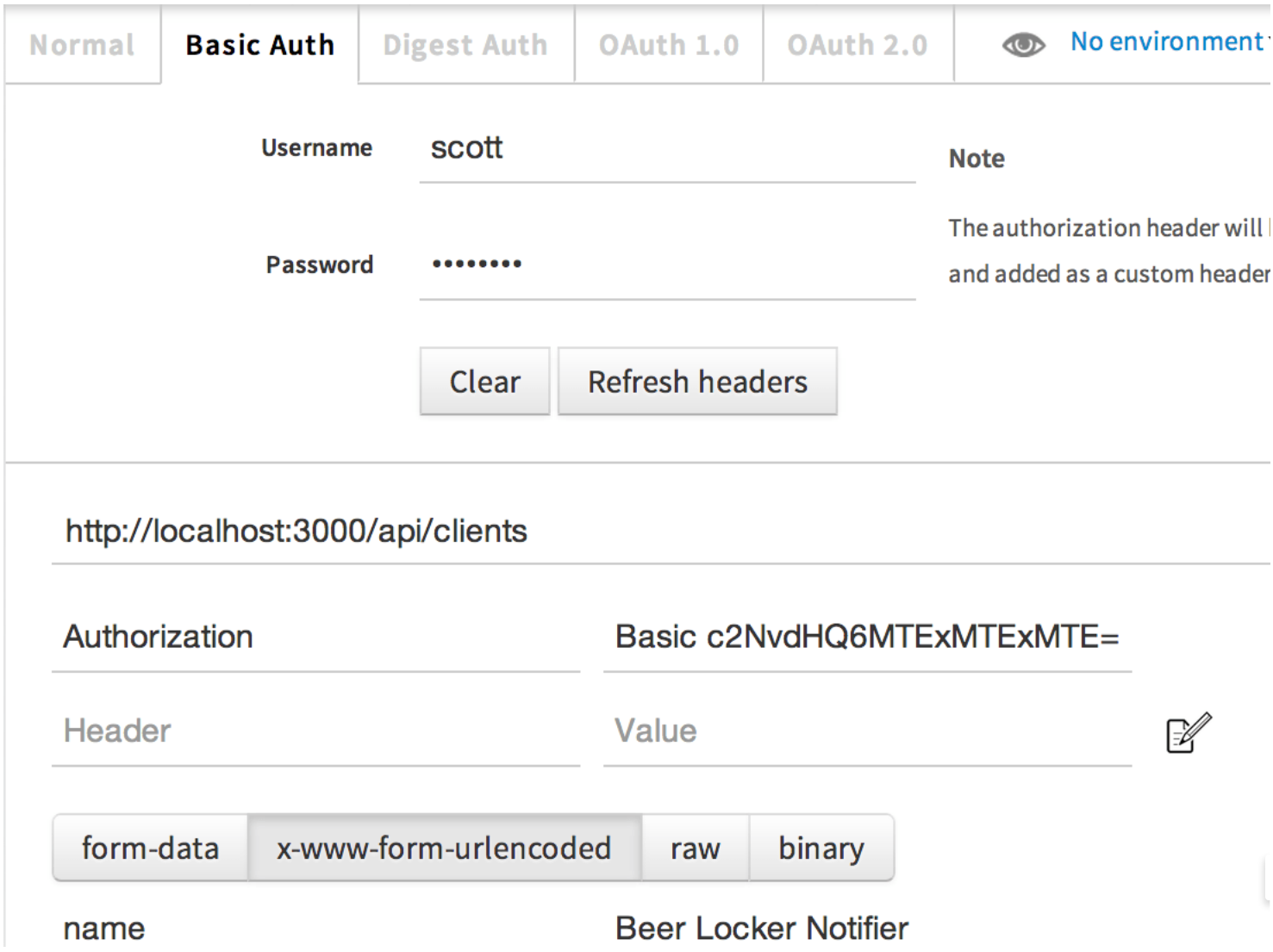
Finally, in the `server.js` file we need to require the new controller and add some new routes for the two endpoints. The new route can be added just after the `/users` route.

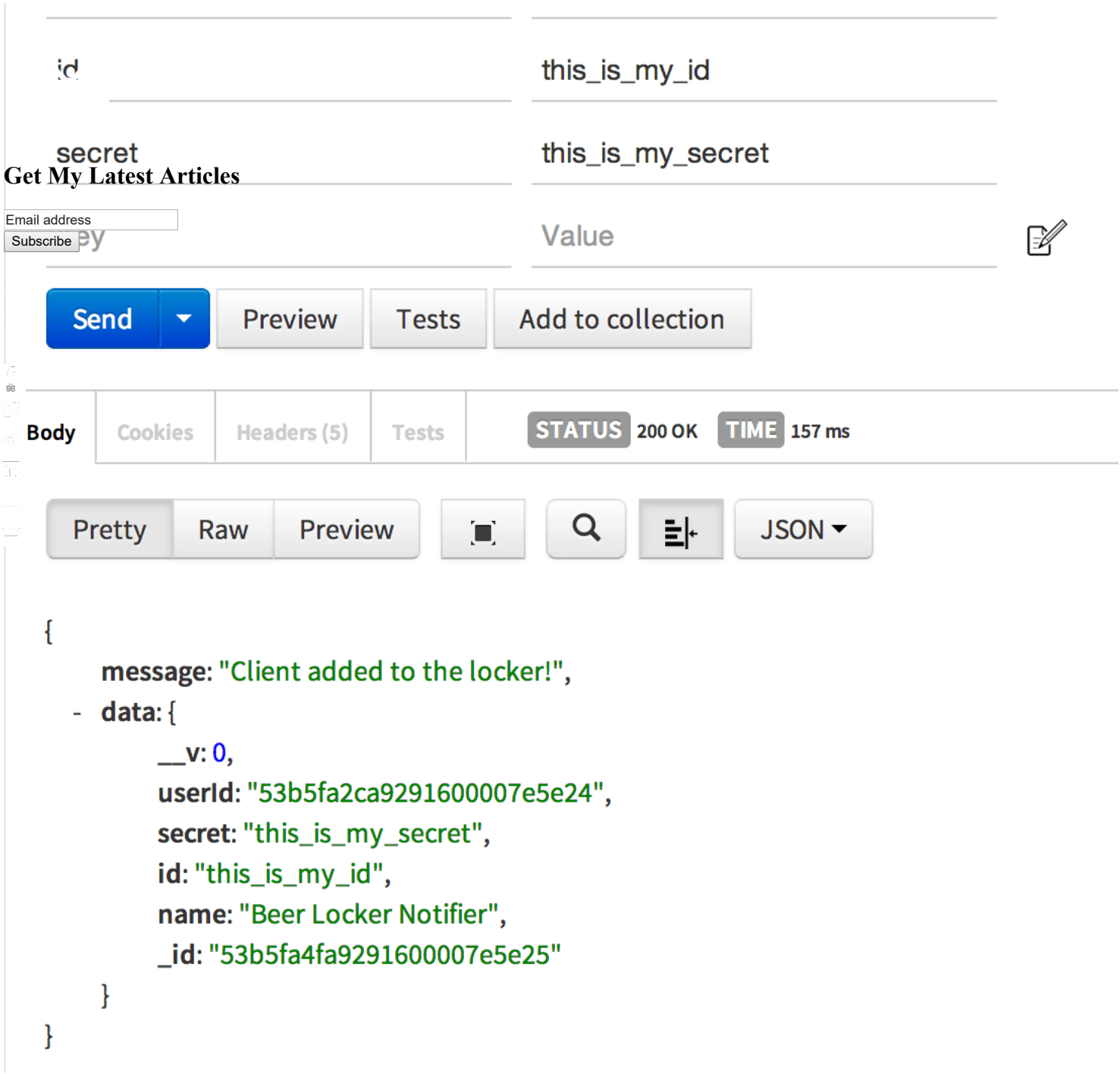
```
var clientController = require('./controllers/client');

...

// Create endpoint handlers for /clients
router.route('/clients')
  .post(authController.isAuthenticated, clientController.postClients)
  .get(authController.isAuthenticated, clientController.getClients);
```

Using Postman, let's go ahead and create a new application client. If for some reason you forgot your password for your user, you should make a new one by posting to the /users endpoint with username and password.





Authenticate our application client

We already created the ability to authenticate a user in our previous article using the BasicStrategy. We need to do the same here so we can lock down our token exchange endpoint which we will implement later.

Update the `controllers/auth.js` file to require the Client model, add a new BasicStrategy to passport, and setup an export that can be used to verify the client is authenticated.

```
1  var Client = require('../models/client');
2
3  ...
4
5  passport.use('client-basic', new BasicStrategy(
6    function(username, password, callback) {
7      Client.findOne({ id: username }, function (err, client) {
8        if (err) { return callback(err); }
9
10       // No client found with that id or bad password
11       if (!client || client.secret !== password) { return callback(null, false); }
12
13       // Success
14       return callback(null, client);
```

```
15    });
16  }
17 \\:
18
19 . .
20
21 exports.isClientAuthenticated = passport.authenticate('client-basic', { session : false });
```

The one thing to note here is that when we call `passport.use()` we are not just supplying a BasicStrategy object. Instead we are also giving it the name `client-basic`. Without this, we would not be able to have two BasicStragies running at the same time.

Authorization Codes

We need to create another model that will store our authorization codes. These are the codes generated in the first part of the OAuth2 flow. These codes are then used in later steps by getting exchanged for access tokens.

Create a new file called `code.js` in the `models` directory and add the following code to it.

```
1  // Load required packages
2  var mongoose = require('mongoose');
3
4  // Define our token schema
5  var CodeSchema   = new mongoose.Schema({
6    value: { type: String, required: true },
7    redirectUri: { type: String, required: true },
8    userId: { type: String, required: true },
9    clientId: { type: String, required: true }
10 });
11
12 // Export the Mongoose model
13 module.exports = mongoose.model('Code', CodeSchema);
```

It is a pretty simple model with the `value` field used to store our authorization code. `redirectUri` is there to store the redirect uri supplied in the initial authorization process so we can add a bit more security later on to make sure the token exchange is legitimate. The `userId` and `clientId` fields are used to know what user and application client own this code.

It is also worth noting, that to be extra secure, you should consider hashing the authorization code.

Access Tokens

Now we need to create the model that will store our access tokens. Access tokens are the final step in the OAuth2 process. With an access token, an application client is able to make a request on behalf of the user. We will implement the code a little later that creates and validates them.

Create a new file called `token.js` in the `models` directory and add the following code to it.

```
1  // Load required packages
2  var mongoose = require('mongoose');
3
4  // Define our token schema
5  var TokenSchema   = new mongoose.Schema({
6    value: { type: String, required: true },
7    userId: { type: String, required: true },
8    clientId: { type: String, required: true }
9  });
10
11 // Export the Mongoose model
12 module.exports = mongoose.model('Token', TokenSchema);
```

The `value` field will be of the most interest here. It is the actual token value used when accessing the API on behalf of the user. The `userId` and `clientId` fields are used to know what user and application client own this token.

Just like we did for user passwords, you should implement a strong hashing scheme for the access token. Never store them as plain text as we are in this example.

Authentication using access tokens

Earlier, we added a second BasicStrategy so we can authenticate requests from clients. Now we need to setup a BearerStategy which will allow us to authenticate requests made on behalf of users via an OAuth token. This is done via the `Authorization: Bearer <access token>` header.

First we need to install another npm package that will provide us with the BearerStrategy for Passport.

```
1 npm install passport-http-bearer --save
```

Update the `controllers/auth.js` file to require the `passport-http-bearer` package and Token model, add a new BearerStrategy to passport, and setup an export that can be used to verify the application client request is authenticated.

```
1  var BearerStrategy = require('passport-http-bearer').Strategy
2  var Token = require('../models/token');
```

```
3
4 ...
5
6 passport.use(new BearerStrategy(
7   function(accessToken, callback) {
8     Token.findOne({value: accessToken }, function (err, token) {
9       if (err) { return callback(err); }
10
11      // No token found
12      if (!token) { return callback(null, false); }
13
14      User.findOne({ _id: token.userId }, function (err, user) {
15        if (err) { return callback(err); }
16
17        // No user found
18        if (!user) { return callback(null, false); }
19
20        // Simple example with no scope
21        callback(null, user, { scope: '*' });
22      });
23    });
24  }
25 ));
26
27 ...
28
29 exports.isBearerAuthenticated = passport.authenticate('bearer', { session: false });
```


s new strategy will allow us to accept requests from application clients using OAuth tokens and for us to validate those requests.

mple UI for granting application client access

to this point in our series, we have not added any UI. We need to add a simple page with a form that will allow a user to grant or deny access to their account for any application client requesting access.

There are a lot of template engines to pick from like jade, handlebars, ejs, and more.For this series, I went with ejs.

First, we need to install the ejs npm package.

```
1 npm install ejs --save
```

Next, we need to update our express application to tell it to use ejs as its view engine. Add the following require and app.set statements in `server.js`.

```
1 var ejs = require('ejs');
2
3 ...
4
5 // Create our Express application
6 var app = express();
7
8 // Set view engine to ejs
9 app.set('view engine', 'ejs');
```

Finally, we need to create our view that will let the user grant or deny the application client access to their account.

Create a new directory called `views` and add a file named `dialog.ejs`.

Add the following code to the `dialog.ejs` file.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Beer Locker</title>
5    </head>
6    <body>
7      <p>Hi <%= user.username %>!</p>
8      <p><b><%= client.name %></b> is requesting <b>full access</b> to your account.</p>
9      <p>Do you approve?</p>
10
11     <form action="/api/oauth2/authorize" method="post">
12       <input name="transaction_id" type="hidden" value="<%= transactionID %>">
13       <div>
14       <input type="submit" value="Allow" id="allow">
15       <input type="submit" value="Deny" name="cancel" id="deny">
16       </div>
17     </form>
18
19   </body>
20 </html>
```

We will come back to this page later as we do a full walkthrough of how everything works. For now, we have this in place and can move on to the next piece.

Enable sessions for our express application

OAuth2orize requires session state for the express application in order to properly complete the authorization transaction. In order to do this, we need to install the express-session package.

```
npm install express-session --save
```

Next we need to require the package and use it in our express application.

Update `server.js` with the following code.


```
var session = require('express-session');

// Set view engine to ejs
app.set('view engine', 'ejs');

// Use the body-parser package in our application
app.use(bodyParser.urlencoded({
  extended: true
}));

// Use express session support since OAuth2orize requires it
app.use(session({
  secret: 'Super Secret Session Key',
  saveUninitialized: true,
  resave: true
}));
```

Create our OAuth2 controller

We are finally ready to create our OAuth2 controller that will facilitate the OAuth2 flow.

First, install the oauth2orize package.

```
npm install oauth2orize --save
```

Next, create a new file called `oauth2.js` in the `controllers` directory. We will add the code to this file in steps.

Load required packages

```
// Load required packages
var oauth2orize = require('oauth2orize')
var User = require('../models/user');
var Client = require('../models/client');
var Token = require('../models/token');
var Code = require('../models/code');
```

Create our OAuth2 server

```
// Create OAuth 2.0 server
var server = oauth2orize.createServer();
```

Register serialization and deserialization functions

```
// Register serialialization function
server.serializeClient(function(client, callback) {
  return callback(null, client._id);
});

// Register deserialization function
server.deserializeClient(function(id, callback) {
  Client.findOne({ _id: id }, function (err, client) {
    if (err) { return callback(err); }
    return callback(null, client);
  });
});
```

When a client redirects a user to user authorization endpoint, an authorization transaction is initiated. To complete the transaction, the user must authenticate and approve the authorization request. Because this may involve multiple HTTP request/response exchanges, the transaction is stored in the session.

Register authorization code grant type

```
// Register authorization code grant type
server.grant(oauth2orize.grant.code(function(client, redirectUri, user, ares, callback) {
  // Create a new authorization code
  var code = new Code({
    value: uid(16),
    clientId: client._id,
    redirectUri: redirectUri,
```

```
    userId: user._id
  });

  // save the auth code and check for errors
  code.save(function(err) {
    if (err) { return callback(err); }

    callback(null, code.value);
  });
}));
```

Get My Latest Articles

OAuth 2.0 specifies a framework that allows users to grant client applications limited access to their protected resources. It does this through a process of the user granting [illegible] exchanging the grant for an access token.

We are registering here for an authorization code grant type. We create a new authorization code model for the user and application client. It is then stored in MongoDB so we can access it later when exchanging for an access token.

Exchange authorization codes for access tokens

```
// Exchange authorization codes for access tokens
server.exchange(oauth2orize.exchange.code(function(client, code, redirectUri, callback) {
  Code.findOne({ value: code }, function (err, authCode) {
    if (err) { return callback(err); }
    if (authCode === undefined) { return callback(null, false); }
    if (client._id.toString() !== authCode.clientId) { return callback(null, false); }
    if (redirectUri !== authCode.redirectUri) { return callback(null, false); }

    // Delete auth code now that it has been used
    authCode.remove(function (err) {
      if(err) { return callback(err); }

      // Create a new access token
      var token = new Token({
        value: uid(256),
        clientId: authCode.clientId,
        userId: authCode.userId
      });

      // Save the access token and check for errors
      token.save(function (err) {
        if (err) { return callback(err); }

        callback(null, token);
      });
    });
  });
}));
```

What we are doing here is registering for the exchange of authorization codes for access tokens. We first look up to see if we have an authorization code for the one supplied. If we do we perform validation to make sure everything is as it should be. If all is well, we remove the existing authorization code so it cannot be used again and create a new access token. This token is tied to the application client and user. It is finally saved to MongoDB.

User authorization endpoint

```
// User authorization endpoint
exports.authorization = [
  server.authorization(function(clientId, redirectUri, callback) {

    Client.findOne({ id: clientId }, function (err, client) {
      if (err) { return callback(err); }

      return callback(null, client, redirectUri);
    });
  }),
  function(req, res){
    res.render('dialog', { transactionID: req.oauth2.transactionID, user: req.user, client: req.oauth2.client });
  }
]
```

This endpoint, initializes a new authorization transaction. It finds the client requesting access to the user's account and then renders the `dialog` ejs view we created eariler.

User decision endpoint

```
// User decision endpoint
exports.decision = [
  server.decision()
]
```

This endpoint is setup to handle when the user either grants or denies access to their account to the requesting application client. The `server.decision()` function handles the data submitted by the post and will call the `server.grant()` function we created earlier if the user granted access.

Application client token exchange endpoint

```
// Application client token exchange endpoint
exports.token = [
```

```
3   server.token(),
4   server.errorHandler()
5 ]
```

This endpoint is setup to handle the request made by the application client after they have been granted an authorization code by the user. The `server.token()` function will initiate a call to the `server.exchange()` function we created earlier.

Utility functions to generate unique identifiers


```
1  function uid (len) {
2    var buf = []
3    , chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'
4    , charlen = chars.length;
5
6    for (var i = 0; i < len; ++i) {
7      buf.push(chars[getRandomInt(0, charlen - 1)]);
8    }
9
10   return buf.join('');
11 };
12
13 function getRandomInt(min, max) {
14   return Math.floor(Math.random() * (max - min + 1)) + min;
15 }
```

dd routes to OAuth2 endpoints

w that we have the controller made for our OAuth2 endpoints, we need to update our express application to add the necessary routes to those endpoints.

erver.js require the new oauth2 controller and add a few new routes.

```
1  var oauth2Controller = require('./controllers/oauth2');
2
3  ...
4
5  // Create endpoint handlers for oauth2 authorize
6  router.route('/oauth2/authorize')
7    .get(authController.isAuthenticated, oauth2Controller.authorization)
8    .post(authController.isAuthenticated, oauth2Controller.decision);
9
10 // Create endpoint handlers for oauth2 token
11 router.route('/oauth2/token')
12   .post(authController.isClientAuthenticated, oauth2Controller.token);
```

Access token authorization on API endpoints

At this point we have everything in place for a fully functioal OAuth2 server. The last piece we need is to update our endpoints that require authorization. Currently, we are authorizing with the BasicStrategy which uses username/password. We need to update that to also allow it to use the BearerStrategy which will allow the use of the access token.

Change the `exports.isAuthenticated` call in `controllers/auth.js` to use either basic or bearer strategies.

```
1 exports.isAuthenticated = passport.authenticate(['basic', 'bearer'], { session : false });
```

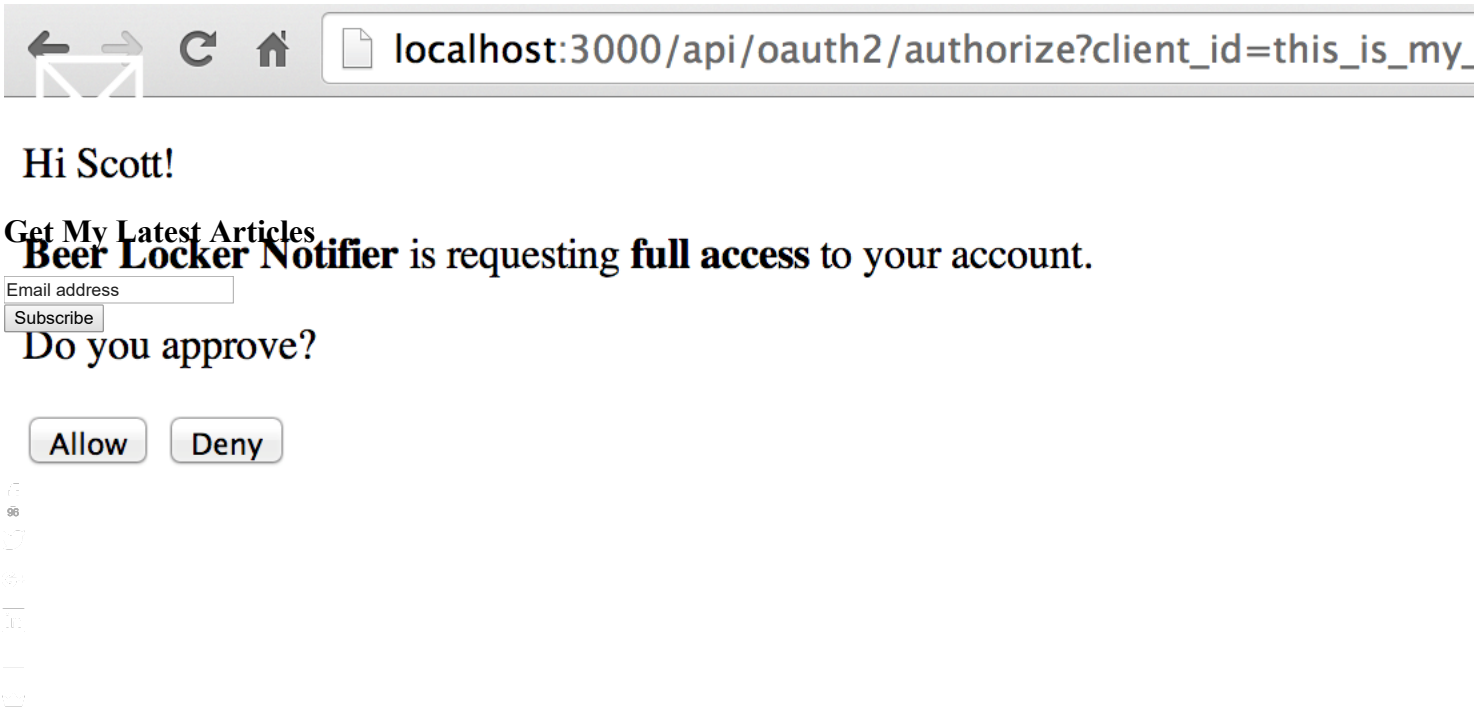
We are already using the `isAuthenticated` function on our endpoints so this change will allow authorization with usernamne/password and access tokens.

Let's use our OAuth2 server!

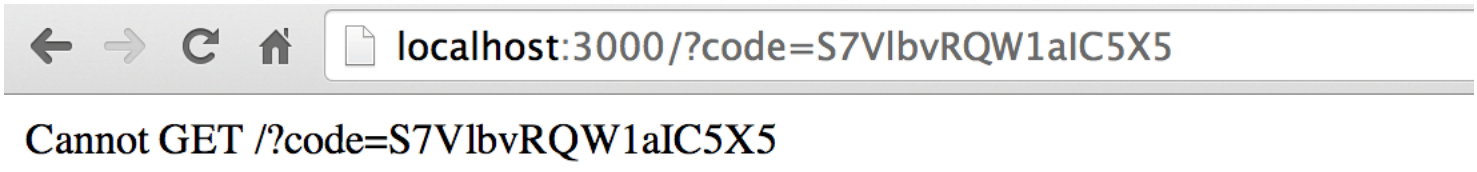
That was a lot of code! Still far less than it would have been had we not used OAuth2orize.

Now it is time to actually try it out.

Open up your favorite web browser and browse to: http://localhost:3000/api/oauth2/authorize?client_id=this_is_my_id&response_type=code&redirect_uri=http://localhost:3000. If you used a different client id, then change it in the query string. Also, if you are running on a different port, be sure to change that in both places. When prompted, enter your username and password.



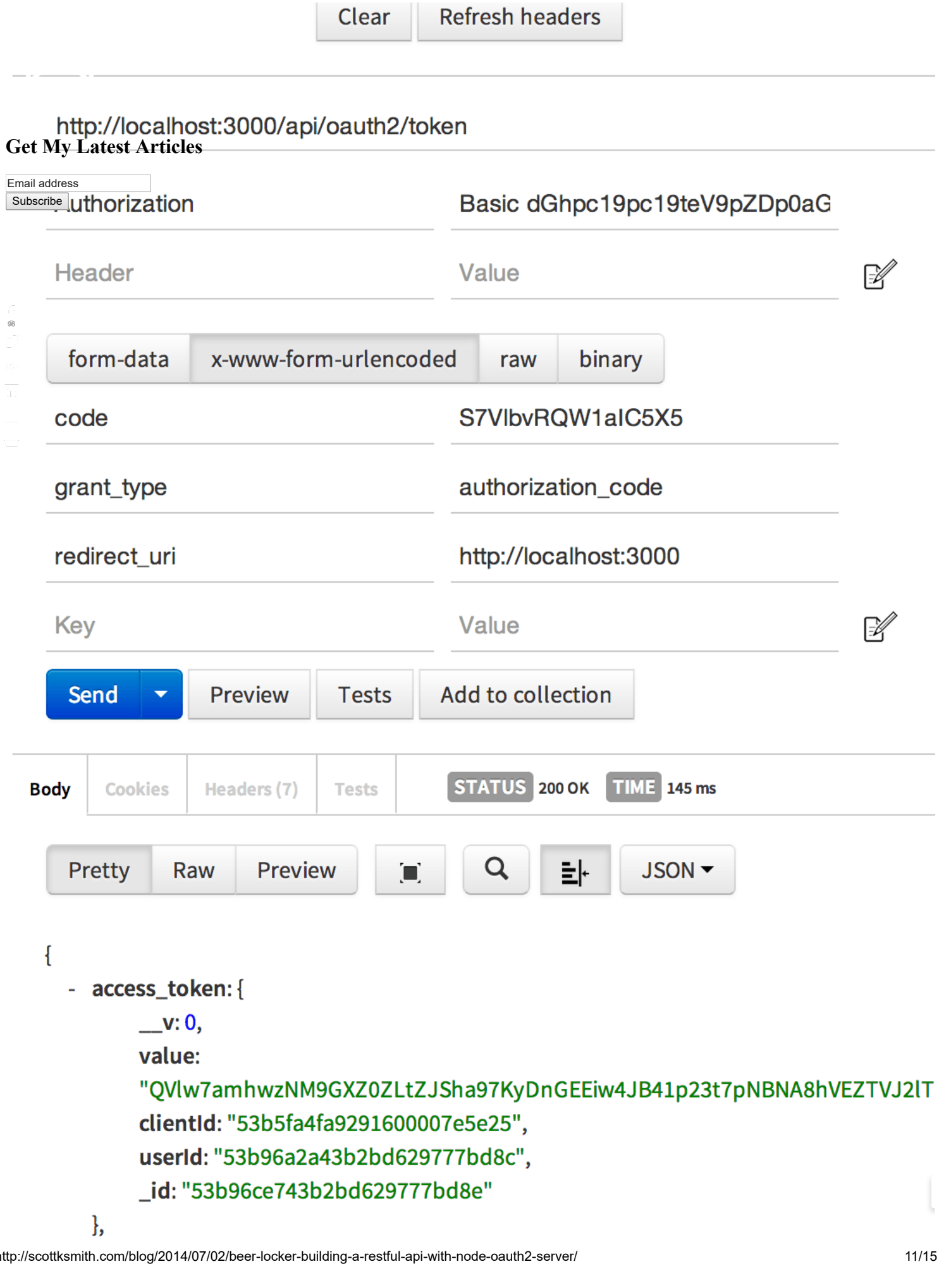
You can test it out by clicking `Deny` if you want and should see it not continue the OAuth2 flow. Go ahead and click `Allow` to continue to the next step.



So why did we get a 404? This is part of the tutorial where we are hacking things together a bit. Normally with OAuth2 you would have an endpoint in the application requesting access to a user's account. That is the query string `redirect_uri` that we supplied. So when a user grants access, that URI is requested and passed the authorization code. This then allows the requesting application to exchange that code for an access token.

To continue this tutorial, we will fake an application server using Postman. Go ahead and copy the authorization code from the query string `code`. Mine in this example would be `S7VlbvRQW1aIC5X5`.

In Postman, we will want to POST to http://localhost:3000/api/oauth2/token, set the Basic Auth username and password to the client id and client secret for your application client, add set post data values code, grant_type, and redirect_uri. Code needs to be set the code you copied from the browser request. Grant_type needs to be set to authorization_code because that is the type we are using. Redirect_uri needs to be set to the same redirect_uri you used in the authorization code request.


Clear
Refresh headers
http://localhost:3000/api/oauth2/token
Get My Latest Articles
Email address
Subscribe
uthorization
Basic dGhpc19pc19teV9pZDp0aG
Header
Value
form-data
x-www-form-urlencoded
raw
binary
code
S7VlbvRQW1aIC5X5
grant_type
authorization_code
redirect_uri
http://localhost:3000
Key
Value
Send
Preview
Tests
Add to collection
Body
Cookies
Headers (7)
Tests
STATUS 200 OK
TIME 145 ms
Pretty
Raw
Preview
JSON
{
- access_token: {
__v: 0,
value:
"QVlw7amhwzNM9GXZ0ZLtZJSha97KyDnGEEiw4JB41p23t7pNBNA8hVEZTVJ2lT
clientId: "53b5fa4fa9291600007e5e25",
userId: "53b96a2a43b2bd629777bd8c",
_id: "53b96ce743b2bd629777bd8e"
},

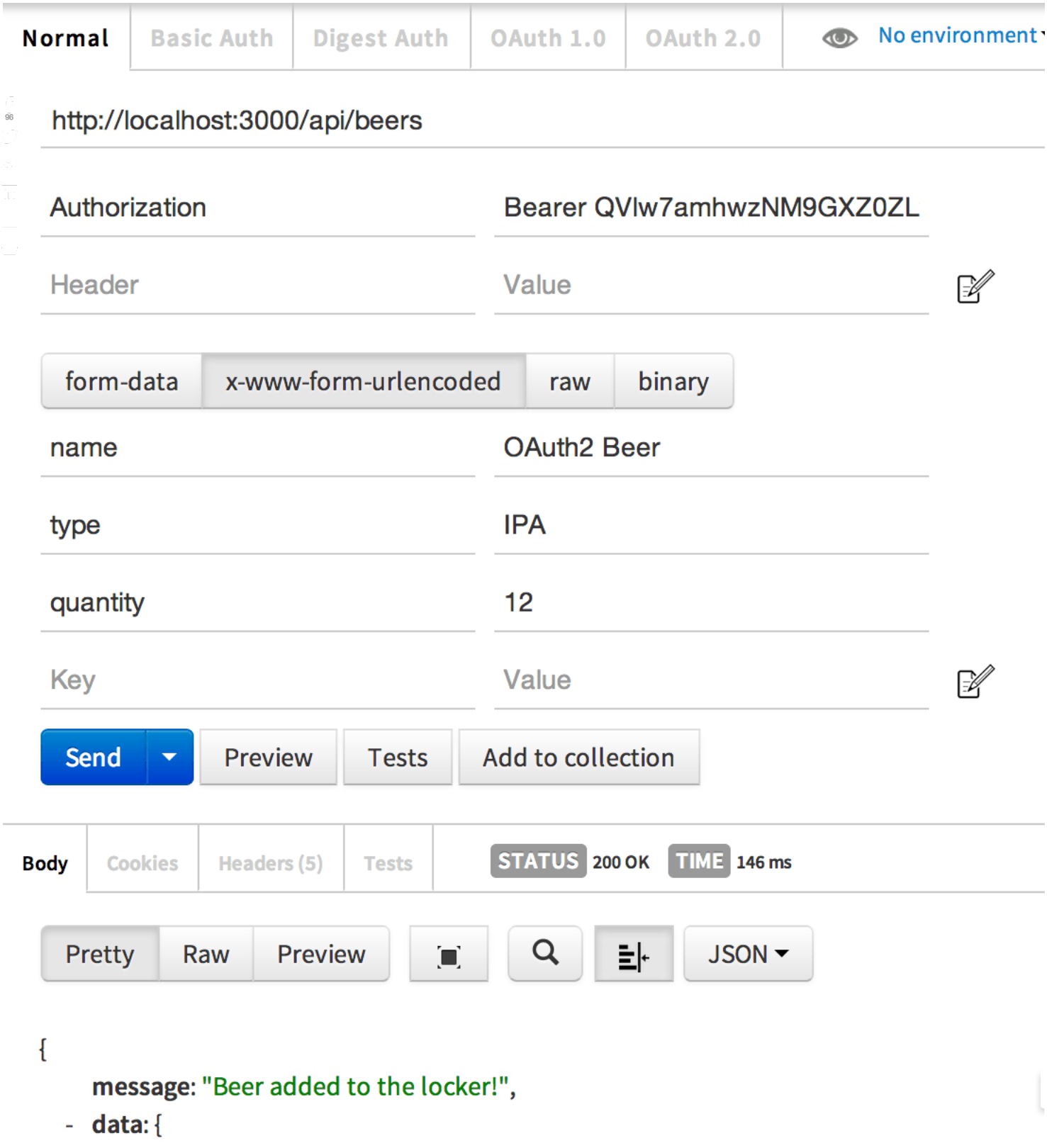
```
    token_type: "Bearer"
}
```


See that `value` field in the response `access_token` object? That is our access token which we can now use to make API requests on behalf of the user!

Let's test our access token by making a request to our API endpoints.

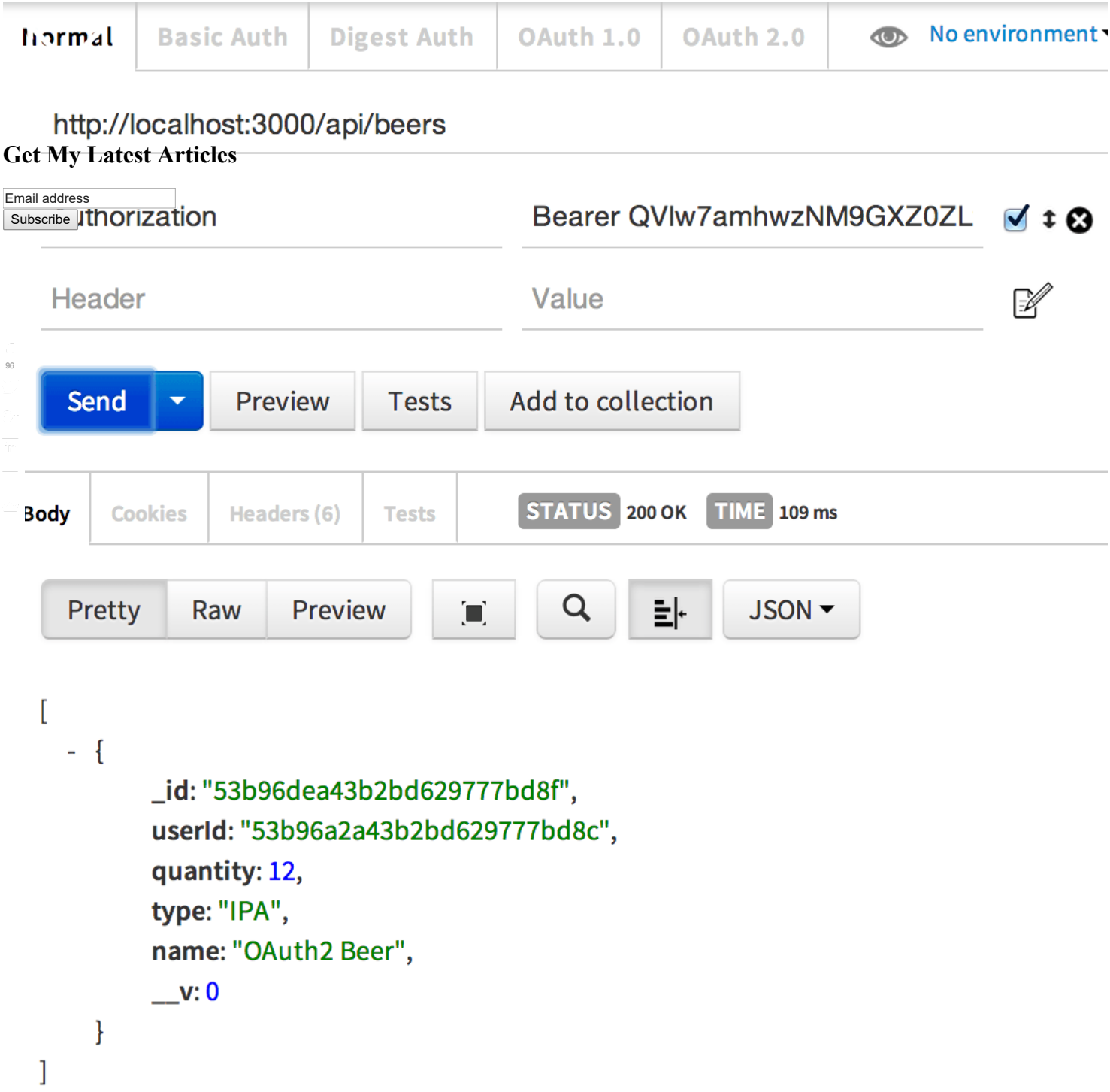
All you have to do is make GET, POST, PUT, or DELETE requests to the API endpoints we made in earlier tutorials. The only difference is you don't have to supply a [illegible]ead, you will add an Authorization header with the value set to `Bearer <access token>`

Add beer to the user's locker



```
__v: 0,
userId: "53b96a2a43b2bd629777bd8c",
quantity: 12,
type: "IPA",
name: "OAuth2 Beer",
_id: "53b96dea43b2bd629777bd8f"
}
}
```

Get beer from the user's locker



Feel free to play around a bit. You should be able to alter the access token and find you are unauthorized. Switch back to username and password to verify the user still has access.

Wrap up

You now have a fully functional OAuth2 server done with just a little bit of work. OAuth2orize is an amazing library that makes building our server very straightfoward.

I have a lot more tutorials coming so be sure to subscribe to my RSS feed or follow me on Twitter. Also, if there are certain topics you would like me to write on, feel free to leave comments and let me know.

Source code for this part can be found here on GitHub.

Posted by Scott Smith Jul 2nd, 2014 Beer, JavaScript, Node.js, Tutorials

« 8 NPM tips for better Node development Using secure cookies in Node on Azure »

Get My Latest Articles

Email address
Subscribe

Software developer, learner, writer, & speaker.

Currently jailbreaking the degree at Degreed and promoting lifelong learning.

Past projects incude coderbits which was sold to Appirio in 2014 and Favatron.