OAuth2.0

---

https://medium.com/@henslejoseph/building-a-restful-api-with-node-oauth2-server-4236c134be4

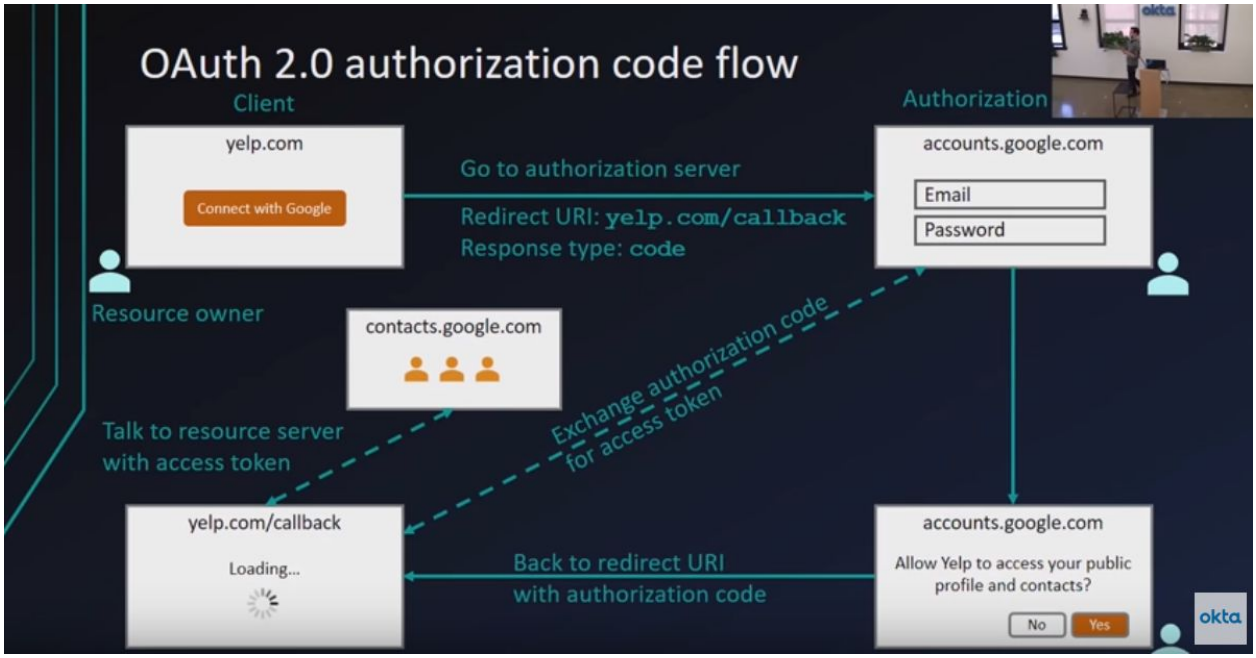
https://www.npmjs.com/package/oauth2orize

http://scottksmith.com/blog/2014/07/02/beer-locker-building-a-restful-api-with-node-oauth2-server/

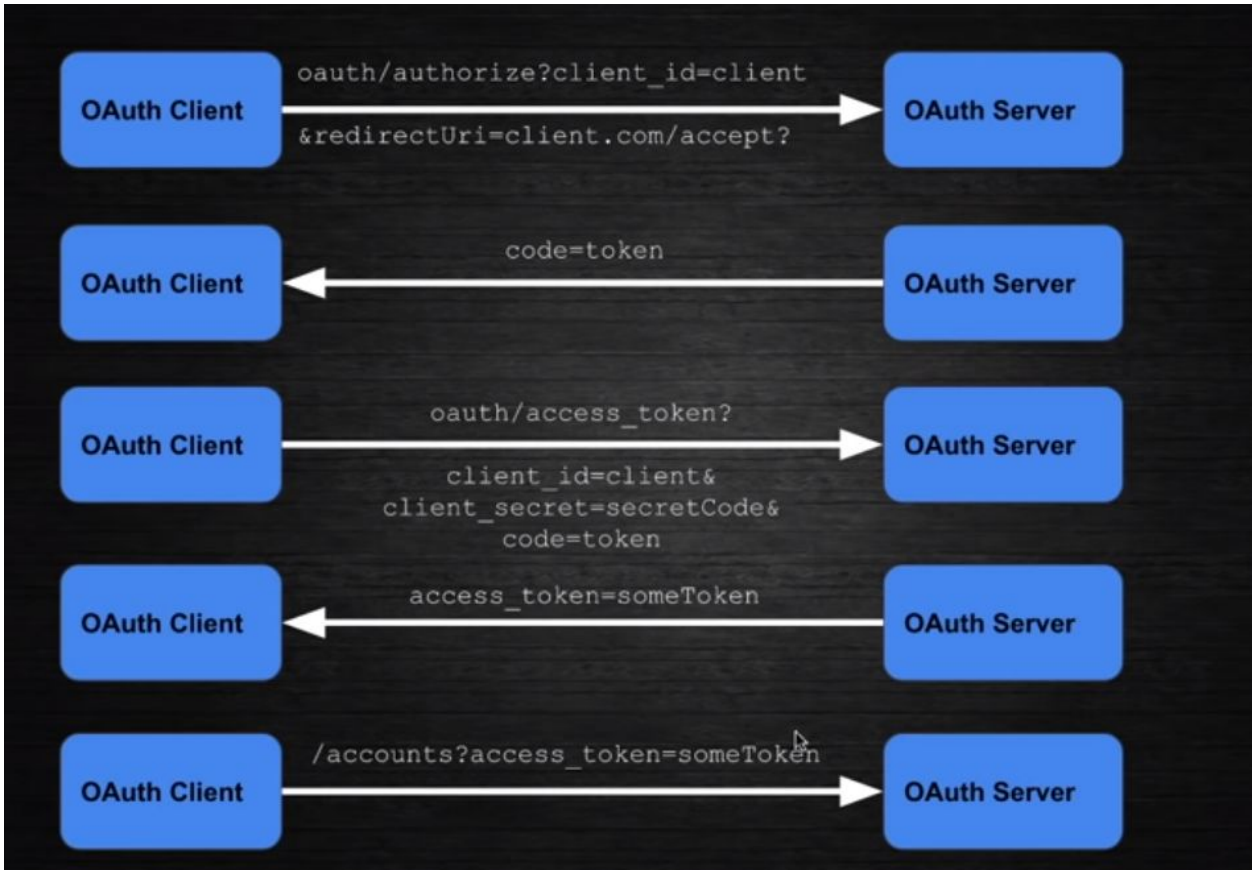
https://www.youtube.com/watch?v=BZwzWgLA0JA&list=PL4cUxeGkcC9jdm7QX143aMLAqyM-jTZ2x&index=13

https://github.com/FrankHassanabad/Oauth2orizeRecipes/wiki/Authorization-code

https://github.com/FrankHassanabad/Oauth2orizeRecipes



OAuth2.0: A protocol outlining how authorization should happen. It is NOT an authentication library.



OAuth 2 Roles

- Resource owner: You, the user that authorizes a client application to access their account
- Client Application: Application (website or app) that wants access to the resource server to obtain information about you
- Resource Server: Server hosting protected data (e.g., your personal information)
- Authorization Server: Server that issues an access token to the client application to request resource from the resource server

OAuth 2 Tokens

- Access token: allows access to user data by the client application
  - Has limited lifetime
  - Need to be kept confidential
  - Scope: parameter used to limit the rights of the access token
- Refresh token: Used to refresh an expired access token

Client Application Registration

- Register the client application on the OAuth service provider:
  - Client App Id
  - Client Secret
  - Redirect URL: URLs for the client for receiving the authorization code and access token

---

Application Client

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

```
// Load required packages
var mongoose = require('mongoose');

// Define our client schema
var ClientSchema = new mongoose.Schema({
  name: { type: String, unique: true, required: true },
  id: { type: String, required: true },
  secret: { type: String, required: true },
  userId: { type: String, required: true }
});

// Export the Mongoose model
module.exports = mongoose.model('Client', ClientSchema);
```

- 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.

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

```
// Load required packages
var Client = require('../models/client');

// Create endpoint /api/client for POST
exports.postClients = function(req, res) {
  // Create a new instance of the Client model
  var client = new Client();

  // Set the client properties that came from the POST data
  client.name = req.body.name;
  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.

form-data | x-www-form-urlencoded | raw | binary

name	Beer Locker Notifier
id	this_is_my_id
secret	this_is_my_secret
Key	Value

Send | Preview | Tests | Add to collection

Body | Cookies | Headers (5) | Tests | STATUS 200 OK | TIME 157 ms

Pretty | Raw | Preview | JSON


```
{
    message: "Client added to the locker!",
  - data: {
        __v: 0,
        userId: "53b5fa2ca9291600007e5e24",
        secret: "this_is_my_secret",
        id: "this_is_my_id",
        name: "Beer Locker Notifier",
        _id: "53b5fa4fa9291600007e5e25"
    }
}
```

Authenticate our application client

- We already created the ability to authenticate a user in our previous article using the BasicStrategy.
- 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.

```
var Client = require('../models/client');

...

passport.use('client-basic', new BasicStrategy(
  function(username, password, callback) {
    Client.findOne({ id: username }, function (err, client) {
      if (err) { return callback(err); }

      // No client found with that id or bad password
      if (!client || client.secret !== password) { return callback(null, false); }

      // Success
      return callback(null, client);
    });
  }
));

...

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.
- The actual implementation for our new BasicStrategy is to lookup a client using the supplied client id and verify the password is correct.

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.

```
// Load required packages
var mongoose = require('mongoose');

// Define our token schema
var CodeSchema   = new mongoose.Schema({
  value: { type: String, required: true },
  redirectUri: { type: String, required: true },
  userId: { type: String, required: true },
  clientId: { type: String, required: true }
});

// Export the Mongoose model
module.exports = mongoose.model('Code', CodeSchema);
```

- It is a pretty simple model with the `value` field used to store our authorization code.
- The `userId` and `clientId` fields are used to know what user and application client own this 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.

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

```
// Load required packages
var mongoose = require('mongoose');

// Define our token schema
var TokenSchema   = new mongoose.Schema({
  value: { type: String, required: true },
  userId: { type: String, required: true },
  clientId: { type: String, required: true }
});

// Export the Mongoose model
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 owns this token.
- Just like we did for user passwords, you should implement a strong hashing scheme for the access token.

Authentication using access tokens

- we added a second BasicStrategy so we can authenticate requests from clients
- Now we need to set up a BearerStategy which will allow us to authenticate requests made on behalf of users via an OAuth token

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

...

passport.use(new BearerStrategy(
  function(accessToken, callback) {
    Token.findOne({value: accessToken }, function (err, token) {
      if (err) { return callback(err); }

      // No token found
      if (!token) { return callback(null, false); }

      User.findOne({ _id: token.userId }, function (err, user) {
        if (err) { return callback(err); }

        // No user found
        if (!user) { return callback(null, false); }

        // Simple example with no scope
        callback(null, user, { scope: '*' });
      });
    });
  }
));

...

exports.isBearerAuthenticated = passport.authenticate('bearer', { session: false });
```

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

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

Simple UI for granting application client access

- 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.
- 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.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Beer Locker</title>
  </head>
  <body>
    <p>Hi <%= user.username %>!</p>
    <p><b><%= client.name %></b> is requesting <b>full access</b> to your account.</p>
    <p>Do you approve?</p>

    <form action="/api/oauth2/authorize" method="post">
      <input name="transaction_id" type="hidden" value="<%= transactionID %>">
      <div>
      <input type="submit" value="Allow" id="allow">
      <input type="submit" value="Deny" name="cancel" id="deny">
      </div>
    </form>

  </body>
</html>
```

Enable sessions for our express application

- OAuth2orize requires session state for the express application in order to properly complete the authorization transaction.
- `npm install express-session --save`
- 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

- First, install the oauth2orize package.

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

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.
- 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);
  });
}));
```

- 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 access, and the client exchanging the grant for an access token.
- 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);
      });
    });
  });
}));
```

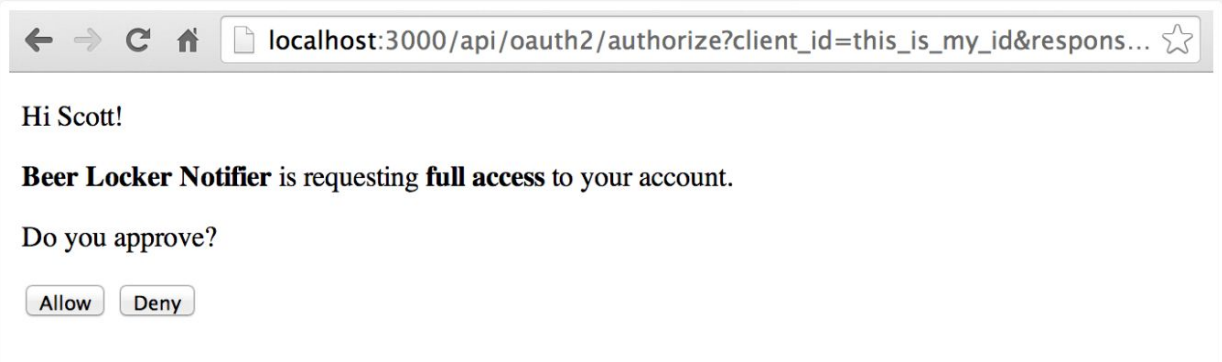
- What we are doing here is registering for the exchange of authorization codes for access tokens.
- we remove the existing authorization code so it cannot be used again and create a new access token.
- This token is tied to the application client and user

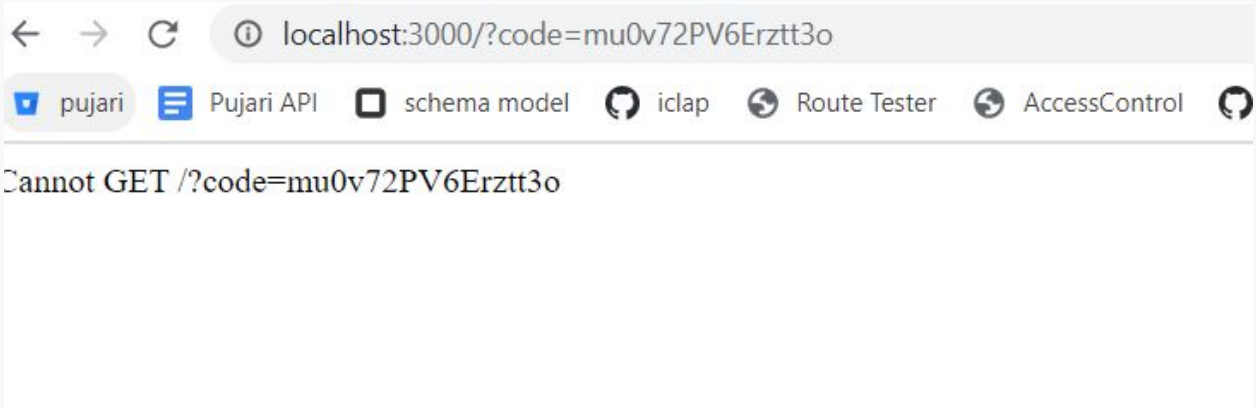
Let's use our OAuth2 server!

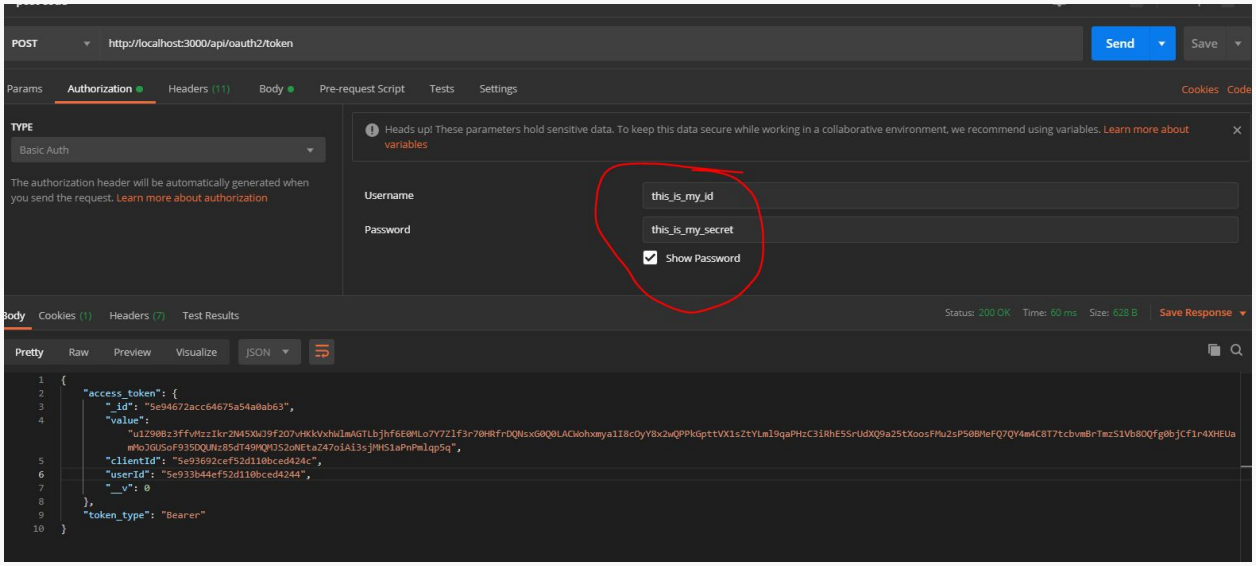
That was a lot of code! Still far less than it would have been had we not used OAuth2orize.

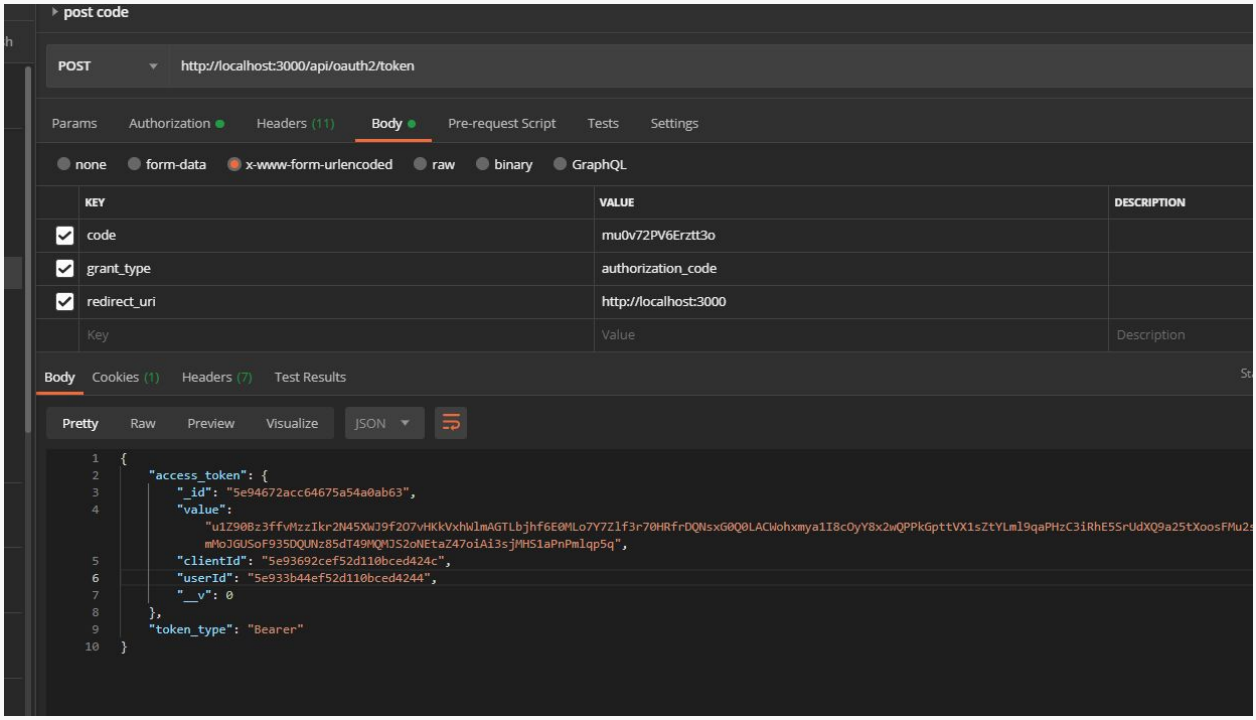
Now it is time to actually try it out.

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.









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 username or password. Instead, you will add an Authorization header with the value set to `Bearer <access token>`

Add beer to the user's locker

add beers

POST http://localhost:3000/api/beers

Params Authorization Headers (11) Body Pre-request Script Tests Settings

Headers 10 hidden

KEY	VALUE	DESCRIPTI
Authorization	Bearer u1Z90Bz3ffvMzzIkr2N45XWJ9f2O7vHKkVxhWlmAGTLbjhf6E0MLo7Y7Zlf3r7 ...	
Key	Value	Descript

Body Cookies (1) Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "message": "Beer added to the locker!",
    "data": {
        "_id": "5e946e83cc64675a54a0ab64",
        "name": "QAuth2 Beer",
        "type": "IPA",
        "quantity": 12,
        "userId": "5e933b44ef52d110bced4244",
        "__v": 0
    }
}
```