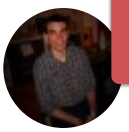


automation | http://bit.ly/2LZoPsI

May 30, 2017 · 17 min read

Node.js REST API Facebook Login

Making user registration and login fast and easy is an important part of developing a new application. Not so long ago, registration forms were mandatory, and, often, users didn't like them much. These forms required that users manually populate data, sometimes 10 or even more input fields, and not every user wanted to spend their precious time on that. Because of that, a lot of web applications lost potential users. Additionally we had login forms, in which users had to enter their usernames/emails and passwords. Often, users would forget their passwords and wouldn't bother going through the process of password recovery, because they would find it tedious. So again, our new and shiny application would lose its users.

Nowadays, we have a much better way to register and login users to our applications. We can now allow users to register using their social network account which they usually have. In this tutorial we will allow users to register and login with their *Facebook* accounts. So, if users want to create a new account in our application, they only need to press a single button provided by our application and then just approve the access to their data on *Facebook*. Similarly, login is just a click of a button away from our users but this time it is not even necessary to allow the access to the data. No password, no username, just a single click of a button :) It's that easy.

In this tutorial we will integrate *Facebook* authentication to a REST API created using *Express.js*. On the backend side we will use *MongoDB* as a database, *Node.js* and *Express.js*. On the frontend side we will implement simple application that will enable us to demonstrate the entire registration and login workflow.

What Is OAuth?

OAuth is an open standard for distributed authentication and authorization. The standard was developed by *Twitter* in 2006, and it is commonly used as a way for Internet users to authorize websites or applications to access their information on other websites without giving them passwords to their accounts. It is used by many well known companies such as *Google*, *Twitter*, *Facebook* and *Microsoft*. A comprehensive list of *OAuth* providers can be found on this link.

What does this means for our application? It means that users can register and use it without entering any data in our registration form. They just need to authorize the application to access their data on a selected *OAuth* provider, and they are ready to use it. Just one click. Isn't that amazing? Additionally, a security benefit of logging in in this manner is that users don't need to remember passwords for our application, and we don't need to store those passwords. This also prevents security holes caused by password reuse and inappropriate storing of passwords.

Authentication Workflow

In this section we will describe the process of the *Facebook* authentication. While in our case we will use a client application written in *Angular 2* and a backend REST API that is written in *Express.js*. Similar process can be applied for any single page application (SPA) and the REST API backend.

When users want to register for our application, they will click the "Signup with *Facebook*". When the button is clicked, our client application will request an access token from *Facebook*. Then, the user will be presented with a dialog to allow the application to access some of their *Facebook* data. If the user gives their permission, our client application will get the *Facebook* access token in response. At this moment we can access user data from the client application, but an account is not yet created at our backend. In order to create new user account, our client application sends a request to our backend with the *Facebook* access token. The backend needs to verify the *Facebook* access token, so it is sends a verification request directly to *Facebook*. If the *Facebook* token is valid, the *Facebook* server will send user data back to our application. Upon receiving this data, the backend server has verified that the user credentials are valid and will create a user profile in our application with data received from *Facebook*. After that, the

backend needs to create a *JSON Web Token (JWT)* which will be used to identify the user. This token is then sent in a response to the client application. The client application will receive *JWT* and save it for further use. Every request that goes to the backend server should contain a *JWT* which uniquely identifies the user. The described flow is shown in figure 1.

The process for user login is similar, so we will not describe it in details. During the login process, users do not need to allow the application to access its *Facebook* data and the backend will not use the *Facebook* data to create a new user, but will instead update the user profile data on our backend.

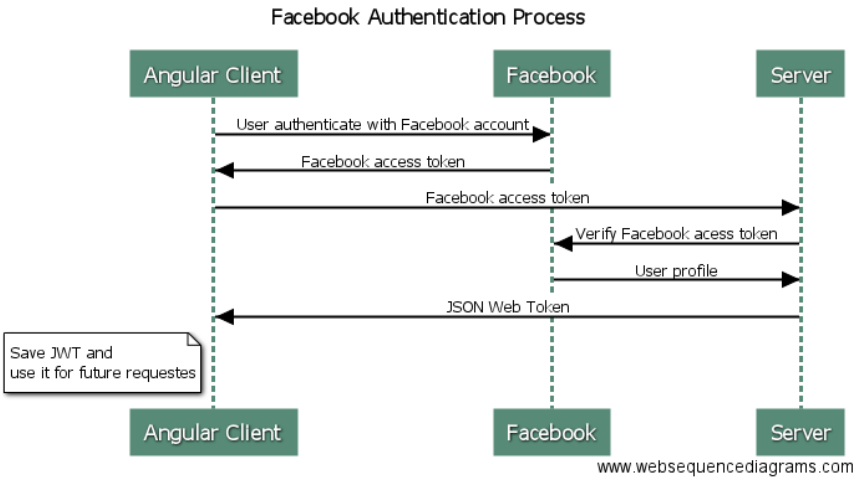


Figure 1. Facebook authentication process

In the following sections we will go through the process of creating a demo application in order to illustrate what we have just described.

Creating a New Facebook Application

In order to make it possible for users to log into our application with their *Facebook* accounts, we need to create an application on the *Facebook* developers page. This is a necessary step for getting an application ID and an application secret code required for all future communication between our application and *Facebook*. *Facebook* provides a well-written tutorial that can be found here. After completing the tutorial we should have an application ID and application secret code like in the figure 2.

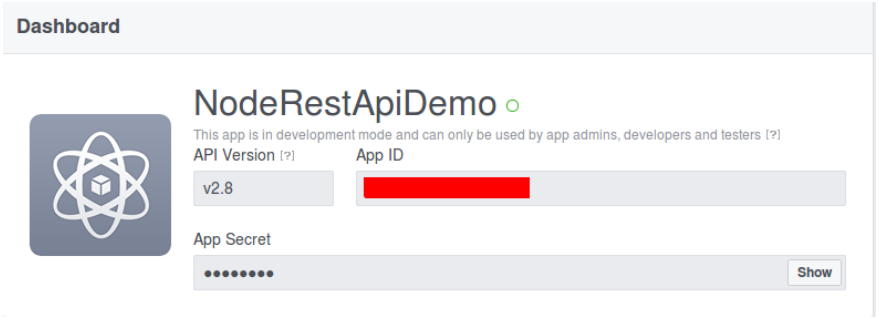


Figure 2. Dashboard picture of created Facebook application

After we have created our application, we need to add some application domains. This can be done on the application page, via the settings menu item. In the App Domains field, just for development purposes, we will add `localhost`. When our application is ready to be released, we will need to replace this with the real application domain.

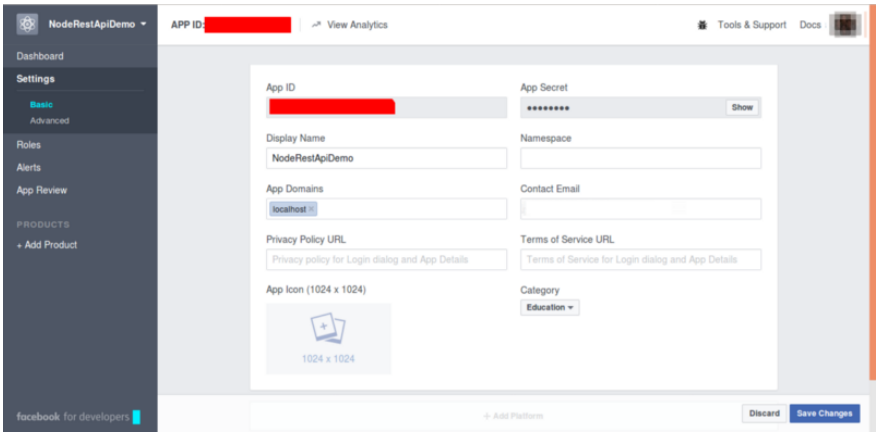


Figure 3. Demo application settings page

Client Setup—Angular 2

In this section we will create a client application in *Angular 2*. The application will demonstrate how to implement *Facebook* authorization in *Angular 2*. As a seed project we will use a project that is explained in detail in our previous blog post.

Code of the client application can be found in frontend directory on *GitHub*.
We will need to add two additional libraries: *bootstrap-social* and *angular2-jwt*:

```
npm install bootstrap-social angular2-jwt — save
```

Bootstrap-social is a small library with well designed social buttons that are made in pure CSS. We will use it to create a good looking *Facebook* sign-in button. The second library, *angular2-jwt*, is used to automatically attach a *JWT* to Authorization header when making HTTP requests from an *Angular 2* application. With it, we will easily send our users' identifiers on every request. For communication with *Facebook* we will use the official *JavaScript SDK*. The *JavaScript SDK* is not available through *npm*, so we will have to add the script in *index.html*:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" >
    <title>AngularBootstrapDemo</title>
    <base href=”/”>
    <meta name=”viewport” content="width=device-width, ini
    <link rel=”icon” type="image/x-icon” href=”favicon.ico
    <script src="//connect.facebook.net/en_US/sdk.js"></sc
  </head>
```

Every time a user opens our application, they will download the *Facebook JavaScript SDK*.

User Service

For communication with *Facebook* and our backend server we will create an *Angular 2* service. The service will be responsible for:

- Using *Facebook* profiles to register users
- Logging users in with *Facebook* profile
- Logging users out
- Checking if users are logged in
- Getting current user

The service can be created with *Angular CLI* command:

```
ng g service user
```

Code of service:

```
@Injectable()
export class UserService {

  constructor(private http: AuthHttp) {
    FB.init({
      appId      : 'YOUR-APP-ID-HERE',
      status     : false, // the SDK will attempt to get i
      cookie     : false,  // enable cookies to allow the
      // the session
      xfbml      : false,  // With xfbml set to true, the
      version    : 'v2.8' // use graph api version 2.5
    });
  }

  fbLogin() {
    return new Promise((resolve, reject) => {
      FB.login(result => {
        if (result.authResponse) {
          return this.http.post(`http://localhost:3000/api
              .toPromise()
              .then(response => {
                var token = response.headers.get('x-auth-t
                if (token) {
                  localStorage.setItem('id_token', token);
                }
                resolve(response.json());
              })
              .catch(() => reject());
        } else {
          reject();
        }
      }, {scope: 'public_profile,email'})
    });
  }
```

In service constructor we are initializing a library that is used to communicate with *Facebook*. The most important part here is to replace string `YOUR-APP-ID-HERE` with the ID of the application that we created on *Facebook*. We need to use same ID on the frontend and on the backend. When we get the access token on the frontend, that access

token will be valid for our application that is registered on *Facebook*. If the backend has a different application ID, and try to use access the token send by the frontend, it will fail.

The next method in the service is `fbLogin`. In `fbLogin` we attempt to get the user's data using `FB.login`. `FB.login` will open a *Facebook* login dialog if the user is not logged in, or a dialog asking to allow the application to use user data if user hasn't used our application before. Response from `FB.login` contains information on whether user is logged in, and whether they have allowed our application to access their data. With that information we are call the backend to log into the application. If we manage to log the user into our backend, we will get a token in response in the `x-auth-token` header. Token will be saved for later use in local storage. The same token has to be sent to the server in every request, and it is used to identify the current user.

The next method, `logout` method, is pretty simple. It deletes the token from the browser's local storage. After that, the user is logged off.

The `getCurrentUser` method is used for getting current user data from the server.

Guards

A common task in the development of any application is deciding what users can and cannot see. *Angular 2* has a built in feature for that purpose—navigation guards. There are four different guard types that we can use:

- ***CanActivate***—Decides if a route can be activated
- ***CanActivateChild***—Decides if children routes of a route can be activated
- ***CanDeactivate***—Decides if a route can be deactivated
- ***CanLoad***—Decides if a module can be loaded lazily

In this example, we want to show the login page when the user is not logged in, and a dashboard page when they are. In order to achieve this we will implement two guards:

- *AuthGuard*—To check if the user is logged in

- *AnonymousGuard*—To check if the user is not logged in

AuthGuard

```
@Injectable()
export class AuthGuard implements CanActivate {

    constructor(private userService: UserService, private

    canActivate(route: ActivatedRouteSnapshot, state: Rout
        return this.checkLogin();
    }

    checkLogin(): Promise<boolean> {
        return new Promise((resolve, reject) => {
            this.userService.isLoggedIn().then(() => {
                resolve(true);
            }).catch(() => {
```

AuthGuard is used to check if the user is logged in. This check will be done by using *UserService* and it's `isLoggedIn` method. If the user is logged in, we will resolve promise, and allow the router to transit to the guarded page. If the user is not logged in, we will navigate them to welcome page.

AnonymousGuard

```
@Injectable()
export class AnonymousGuard implements CanActivate {

    constructor(private userService: UserService, private

    canActivate(route: ActivatedRouteSnapshot, state: Rout
        return this.checkLogin();
    }

    checkLogin(): Promise<boolean> {
        return new Promise((resolve, reject) => {
            this.userService.isLoggedIn().then(() => {
                this.router.navigate(['/dashboard']);
                reject(false);
```

AnonymousGuard is used to check if user is not logged in. This check will, again, be done by *UserService* and it's `isLoggedIn` method. If the user is not logged in, we will resolve promise, and allow the router to transit to the guarded page. If the user is logged in, we will navigate them to dashboard page.

Pages

In this section we will show all the pages that we have created for this demo application. It consists of two pages: welcome and dashboard page. The welcome page is used as a landing page, and contains a *Facebook* login button. If the user is logged in and tries to access the demo page, they will be automatically redirected to the dashboard page. The dashboard page is shown when user is logged in. The page itself shows email information from user details. If an anonymous user tries to access the dashboard page, they will be automatically redirected to the welcome page by guard. So let's get started.

Login Page

Login page can be created with *Angular CLI* command:

```
ng g component login
```

This command will create the template, css and component files for our login page in new folder that is called login. In *login.component.ts* we will add just one method, `fbLogin` , that will be bound to the *Facebook* login button.

```
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.sass']
})
export class LoginComponent implements OnInit {

  constructor(private userService: UserService, private ro

  ngOnInit() {
  }

  fbLogin() {
```

How is *Facebook* authorization done? Very easy, we call the `fbLogin` method from *UserService*. The *UserService* method returns *Promise*. If the user is authenticated with *Facebook*, we will handle that in the method by navigating to dashboard page.

The template for the login page is pretty simple and can be found in *login.component.html*.

```
<div class="navbar navbar-default navbar-fixed-top">
</div>

<div class="page-header"></div>

<div class="container">

  <div class="row">
    <div class="col-lg-8 col-md-7 col-sm-6">
      <div class="panel panel-default">
        <div class="panel-heading text-center">Sign in wit
        <div class="panel-body" align="center">
          <a class="btn btn-social btn-facebook" (click)="
```

The most interesting element of the page is `<a class="btn btn-social btn-facebook" (click)="fbLogin()">` . The `a` element click event is bound to `fbLogin` from *LoginComponent*. So when user clicks the link, the `fbLogin` method will be called.

Dashboard Page

In order to create the dashboard page we can use the *Angular CLI* command:

```
ng g component dashboard
```

This command will create the template, css and component files for our dashboard page in a new folder called dashboard. In *dashboard.component.ts* we will implement two things. On component initialization we will load the user information. This is done in the `ngInit` method. User information is loaded by calling the *UserService* method `getCurrentUser` . User info will be put to the component field `currentUser` , and from there, it will be used in the template for presenting user information.

On this page we will add the `logout` method. The method will be used to logging out current user by using `logout` method from *UserService*.

If the users are successfully logged out, they will be navigated to welcome route.

```
@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.sass']
})
export class DashboardComponent implements OnInit {

  public currentUser : any = {};

  constructor(private userService: UserService, private ro

  ngOnInit() {
    this.userService.getCurrentUser().then(profile => this
        .catch(() => this.currentUser = {});
  }
```

The template for the dashboard page is pretty simple and can be found in *dashboard.component.html*.

```
<div class="navbar navbar-default navbar-fixed-top">
  <ul class="nav navbar-nav navbar-right">
    <li role="menuitem"><a class="dropdown-item" (click)="
  </ul>
</div>

<div class="page-header"></div>

<div class="container">

  <div class="row">
    <div class="col-lg-8 col-md-7 col-sm-6">
      <div class="panel panel-default">
        <div class="panel-heading text-center">Our Awesome
```

So what do we have here? First, we have a navigation bar that contains a *Logout* button. When the user clicks it, the `logout` method from the

dashboard component will be executed. In the center of the page we show current user email:

```
<div class="panel-body" align="center">
  Current User email: {{ currentUser.email }}
</div>
```

Routes

We will have two routes. Router component can be found in *app.routing.module.ts*. So we will have array with routes:

```
const appRoutes: Routes = [
    {
        path: 'welcome',
        component: LoginComponent,
        canActivate: [AnonymousGuard]
    },
    {
        path: 'dashboard',
        component: DashboardComponent,
```

Every *JSON* object is one route. So for the welcome path, we will use *LoginComponent*, and that path can be accessed only if the user is not logged in. For the dashboard path we will use *DasboardComponent*, and that route can be accessed only if the user is logged in. The default route, when the user loads our site without any path, will redirect to welcome page. So if user enters http://localhost:3000, they will be redirected to the welcome page.

AppModule

The last thing that we will show on the frontend will be in *app.module.ts*. For communication with our backend, as you may have noticed, we are not using *Angular 2 Http* module, but *AuthHttp*. Why? Because *AuthHttp* is a class from a *angular2-jwt*. We are using it to automatically attach a JSON Web Token (JWT) as an Authorization header when making HTTP requests from an *Angular 2* application. We have decided that as a header name we want to use `x-auth-token`

name. `noJwtError` is set to true, so that we can use *AuthHttp* for communication with the backend even when the user is not logged in. `tokenGetter` is the function used for finding the token of current user. In our case that is in local storage, and the token is stored with key `id_token`.

```
export function getAuthHttp(http: Http) {
  return new AuthHttp(new AuthConfig({
    headerName: 'x-auth-token',
    noTokenScheme: true,
    noJwtError: true,
    globalHeaders: [{'Accept': 'application/json'}],
    tokenGetter: (() => localStorage.getItem('id_token')),
```

Backend Setup—Node.js, Express.js

In this chapter we will describe our backend. We have decided that we want to create our backend application based on *Node.js* and *Express.js*. We will create REST API and the user should be able to log in with his *Facebook* account. At the end we will have a fully functional backend application on which the user will be able to create a new account, and to log into it with their *Facebook* account.

Dependencies

For creating REST API we will use *Express.js*. Parsing incoming request bodies in a middleware before our handlers will be done by *body-parser*. Handling *JWT* will be done by *express-jwt* and *jsonwebtoken*. Authentication will be done with *passport*, and for *Facebook* authentication we will use *passport-facebook-token* library. As database we will use *MongoDB*, and *mongoose* to communicate with the database. Last but not least, we will need to enable *CORS* on our server and for that we will use *cors* library.

Database Model

We will persist user information into database. As a database we will use *MongoDB*. We don't want to communicate with the database directly, so we will use *mongoose*. On the official project page we can find the definition of *mongoose*:

> *MongoDB object modeling tool designed to work in an asynchronous environment. Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.*

We have only one entity to model and that is user. We are only interested in the user`s email information from their *Facebook* profile, so that is the only thing that we will include. Also we will persist user access token and *Facebook* profile id.

```
var UserSchema = new Schema({
    email: {
          type: String, required: true,
          trim: true, unique: true,
          match: /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,
    },
    facebookProvider: {
          type: {
                id: String,
                token: String
```

An interesting thing to notice here, is that by putting `select` to `false` in `facebookProvider` object, we exclude `facebookProvider` field in result of queries by default. In order to get user with `facebookProvider` field, we need to specify explicitly in the query that we want that.

In *UserSchema* we have to add one static method that will be used for creating a new user if user doesn't exist already. That method is `upsertFbUser`.

```
UserSchema.statics.upsertFbUser = function(accessToken, re
    var that = this;
    return this.findOne({
          'facebookProvider.id': profile.id
    }, function(err, user) {
      // no user was found, lets create a new one
      if (!user) {
            var newUser = new that({
                  email: profile.emails[0].value,
                  facebookProvider: {
                        id: profile.id,
                        token: accessToken
                  }
            });

            newUser.save(function(error, savedUser) {
              if (error) {
```

What we try to do in this method, is, first, to find the user by their *Facebook* profile id. If we find a user with a matching *Facebook* profile id, that means that they already have an account, which is associated with their *Facebook* profile. In that case we don't need to do anything here. If we haven't found a user, we will create a new user and save it in the database.

Passport Configuration

As mentioned, for authentication, we will use the *Passport* library. *Passport* is authentication middleware for *Node.js*. It's extremely flexible and modular. *Passport* supports many authentication mechanisms, which are referred to as *strategies*, so there is a local strategy, for login with username and password, a *Facebook* strategy, a *Twitter* strategy, etc. A list of all strategies can be found on the official site.

We will use *passport-facebook-token*, the *Passport* strategy for authenticating with Facebook access tokens using the *OAuth* 2.0 API. There is also the passport-facebook library that contains a strategy for *Facebook* authentication, but this library is not suitable for REST API. It is better suited for *Express.js* applications which are used for server rendering.

The code for initialization of *Passport* with *Facebook* strategy looks like this:

```
passport.use(new FacebookTokenStrategy({
      clientID: 'YOUR-CLIENT-ID-HERE',
      clientSecret: 'YOUR-CLIENT-SECRET-HERE'
    },
    function (accessToken, refreshToken, profile, done) {
      User.upsertFbUser(accessToken, refreshToken, profile,
        return done(err, user);
```

In order to initialize the strategy we need to supply our *Facebook* application client ID and client secret. Those two pieces of information will be used to identify our application. We also must supply a verify callback function. In our case this function calls the static method `upserFbUser` from *User* model, which will check if the given user exist, and if not, create one.

Token Handling

JWT, as explained on *Wikipedia*:

> *JWT is a JSON-based open standard (RFC 7519) for creating access tokens that assert some number of claims. For example, a server could generate a token that has the claim "logged in as admin" and provide that to a client. The client could then use that token to prove that it is logged in as admin.*

In our example we will only store the user's ID in a token, sign it, and send it to the frontend. For creating *JWT* we will use *jsonwebtoken* library.

```
var createToken = function(auth) {
  return jwt.sign({
    id: auth.id
  }, 'my-secret',
  {
    expiresIn: 60 * 120
  });
};

var generateToken = function (req, res, next) {
  req.token = createToken(req.auth);
  next();
```

In the function `createToken` we are getting user (`auth`) as function argument, and use the id to create a token. In this case we are using *my-secret* as private key; in production we should either the secret for *HMAC algorithms*, or the *PEM encoded private key* for *RSA* and *ECDSA* as stated in the library documentation. The function `generateToken` has request and response, and because of that it has access to current user. So we will generate a token and put it in the request object. `sendToken` is a function that will take the token from request object and put it in the header.

For validation of the *JWT* in every frontend request, we will use *express-jwt*. How do we use *express-jwt*? If the token is valid, `req.auth` will be set with the *JSON* object decoded to be used by later middleware for authorization and access control. The `authenticate` function does exactly that:

```
var authenticate = expressJwt({
  secret: 'my-secret',
  requestProperty: 'auth',
  getToken: function(req) {
    if (req.headers['x-auth-token']) {
      return req.headers['x-auth-token'];
    }
```

Property secret should have the same value as in the `createToken` function. By setting `requestProperty` we have the changed property

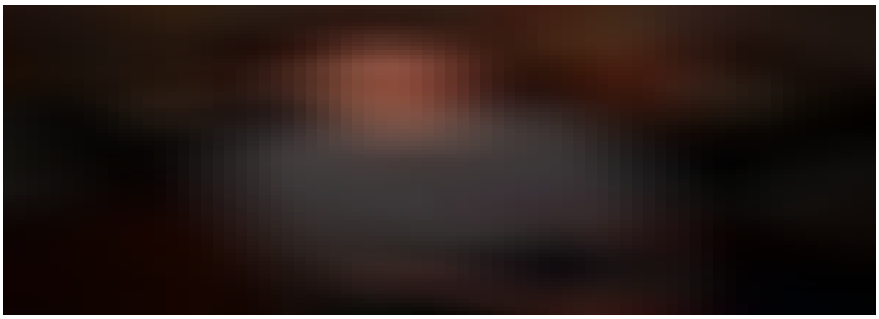
name into name which we will put into the decoded *JSON* object. The property `getToken` is a function that is used to extract a token from request, and in this case the token is in the header `x-auth-token` .

Authentication Flow

So, by now, we have all the building blocks. We have to connect all of our blocks in a system that will do the *Facebook* authentication. For authentication, we will create REST endpoint with the path : http://localhost:3000/api/v1/auth/facebook. We will also create an endpoint from which we can get the currently authenticated user: http://localhost:3000/api/v1/auth/me.

For authentication, we are creating a route for POST HTTP method. When the user calls this endpoint, we first use *Passport* to authenticate the user. After that, we add the middleware, in which we are checking if there is a user with the given *Facebook* account in system. If one is not found, we send response with 401 HTTP status code. If the user is found, we create an `auth` object, into which we put just user ID. The next middleware is for generating the token, and after that we use middleware for sending that token.
To get current user, we first need to create a middleware which will find the user in the database. That will be done via the `getCurrentUser` function. Function `getOne` takes the found user, and sends it as a response. The endpoint for getting current user is, basically :

- Authenticate user and find it's ID
- Find current user in database
- Return current user as response



CORS

Currently, we have two applications, *Angular 2* application and *Node.js/Express.js* application, hosted on two servers. If we don't set up *CORS*, our *Angular 2* application will not be able to communicate with the backend. In order to enable *CORS* on our backend, we will use the *Node.js* library *cors*.

By setting origin to true, we have allowed any origin to access the resource. `methods` is used to control **Access-Control-Allow-Methods** *CORS* header, and with this string we have allowed almost all HTTP methods. With the property `credentials` we are controlling the **Access-Control-Allow-Credentials** *CORS* header. The last, and very important property, `exposedHeaders`, is used to indicate which headers can be exposed as part of the response. If we haven't set this property, the client library would ignore our custom header with user token.

HTTPS—Make Sure You Use It

You MUST use HTTPS in production!

In this demo we will work on our local machine and we will NOT using HTTPS—but you MUST use HTTPS in production. Without it, all API authentication mechanisms are compromised.

You have been warned.

Conclusion

Congratulations! You now have a fully functional example of *Facebook* authentication. We have created an *Angular 2* application that can communicate with REST API and authenticate a user with by means of *Facebook* account. We have also created a *Node.js/Express.js* application that exposes REST API for that purpose.

The complete source code from this project can be found at the following *GitHub* repository:

GenFirst/angular2-node-fb-login

angular2-node-fb-login - Demo application that shows how to enable Facebook login with Angul...

github.com

I hope you have enjoyed this tutorial, and learned a thing, or two. There is obviously much more to be done, but this should get you started.

If you have any questions or issues getting this to work feel free to contact me on *Twitter* @robince885 or in the comments below. You can find more information about me on *LinkedIn*:

Ivan Vasiljević | LinkedIn

Teaching various subjects on the Chair for Informatics. Doing research on the domain-...

www.linkedin.com

I would like to thank @vdimitrieski for his help and support. Feel free to contact him as well. Some more information about him and what he is doing you can found on his *LinkedIn*:

Vladimir Dimitrieski | LinkedIn

Include this LinkedIn profile on other websites

www.linkedin.com

I would also like to thank my friends: Srđan Gavrilović for helping me with English and Milan Savić for helping me formatting code examples.