Guided Project | Step 1 - Twitter

Learning goals

After this lesson, you will be able to:

- Create database models with mongoose, based on what we want to do
- Create users and store their data in the database
- Let users log in our application and save their data in session and cookies
- Publish tweets from your account
- See all the tweets you have published

Introduction

In this Learning Unit we are going to create our own twitter clone! We are not going to implement the whole platform, but just some of the features it has: sign up, login, tweet, my tweets, follow users, and timeline.

We are going to split up this exercise in two learning units. In this first part, we will implement the sign up, the login and logout, and we will be able to publish tweets and see our tweets when we are logged in.

Starter code

Let's start checking out our starter code. We have generated the starter code with `express-generator` (https://expressjs.com/en/starter/generator.html), using the following command:

```
$ express twitter-lab --ejs --git
```

💡 Remember you have to install the package `express-generator` globally before use it in the console:

```
$ npm install express-generator -g
```

This command will create by default the `.gitignore` file, and will also install the `ejs` package in our project, adding the basic configuration to the `app.js` file.

We need some other packages to start working in our project. We have to add the following packages to our project:

- Mongo (https://www.npmjs.com/package/mongodb)
- Mongoose (https://www.npmjs.com/package/mongoose)
- EJS Layouts (https://www.npmjs.com/package/express-ejs-layouts)
- bcrypt (https://www.npmjs.com/package/bcrypt)

Let's connect to mongoose:

```
// app.js
const mongoose = require('mongoose');
// other code
mongoose.connect('mongodb://localhost/twitter-lab-development');
```

As you should know, Express has to be restarted every time we change some code. To avoid doing this manually, we will also gloally install nodemon
(https://www.npmjs.com/package/nodemon)

```
$ node install -g nodemon
```

To configure nodemon, we change the `package.json` file adding the following:

```
{
  // ...
  "scripts": {
    "start": "nodemon ./bin/www"
  }
  // ...
}
```

Before start coding, we should configure our main layout to load all the contents within the same layout. We will have it inside the route `/views/layouts/main-layout.ejs`, with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <% include ../partials/head %>

  <title>Ironhack Twitter</title>
</head>
<body>
  <div id="container">
    <img src="/images/twitter-logo.png"
         class="header-logo">

    <%- body %>
  </div>
</body>
</html>
```

As you can see, we have also added an image with the twitter logo in the public images folder. Finally, we have to configure the middleware to load the layout in all the pages of the site. We do that in the `app.js` file:

```
const expressLayouts = require('express-ejs-layouts');

// ...other code
app.use(expressLayouts);
app.set("layout", "layouts/main-layout");
```

We are ready to start with our own twitter version! Happy coding!

Authentication

If we visit twitter (https://twitter.com/), the first we may notice is that there is a login form to access to our profile. This is why authentication is one of the most important features we will implement in the project.

The main goal of the authentication is to discern between different users. It will allow us to access the platform, be able to know who has published each tweet, and who is following us (or who are we following).

The authentication process requires an account creation before doing anything else, so this is the first thing we will do: we will allow users to authenticate themselves with username and password.

Sign-Up

Model

If we want to store our users data in the database, we have to create a database model with Mongoose. As we said before, each user will have a username and a password, so let's create the model in the `/models/user.js` file, with the following Schema:

```
const mongoose = require("mongoose");
const Schema   = mongoose.Schema;

const userSchema = new Schema({
  username: String,
  password: String
}, {
  timestamps: { createdAt: "created_at", updatedAt: "updated_at" }
});

const User = mongoose.model("User", userSchema);
module.exports = User;
```

Remember that an `_id` attribute will be added by default to all documents in the collection. Also timestamps (http://mongoosejs.com/docs/guide.html#timestamps) is a mongoose Schema option to save the time we created/updated a document automatically.

Layout

Now we have to create the layout to sign up. We need to have a form with the different fields we will save in the database collection.

This sign up will be in the `/signup` route, so the users will have to go there to create an account into the platform.

We are going to add the following code in the `/views/auth/signup.ejs` file:

```
<h2>Signup</h2>

<form action="/signup" method="POST" id="form-container">
  <label for="username">Username</label>
  <input type="text" name="username"
    placeholder="JonSnow">
  <br><br>
  <label for="password">Password</label>
  <input type="password" name="password"
    placeholder="Your password">
  <br><br>
  <button>Create account</button>
</form>
```

Now we have to create the corresponding routes and the code that will allow us create and save new user accounts.

Routes

We need two different routes to create a new user:

- The `GET` route to show the form, and
- The `POST` route to receive the parameters and save the data in the database.

We will create all this functionality inside the `/routes/authController.js` file. In this file, we will have to require the packages and modules we need:

```
1  const express        = require("express");
2  const authController = express.Router();
3
4  // User model
5  const User           = require("../models/user");
6
7  // Bcrypt to encrypt passwords
8  const bcrypt         = require("bcrypt");
9  const bcryptSalt     = 10;
```

We are requiring `bcrypt` to be able to encrypt the user's password before save it in the database. We have to install the package and save it in the package.json through the following command:

```
$ npm install bcrypt --save
```

We define the first route as follows:

```
11  authController.get("/signup", (req, res, next) => {
12    res.render("auth/signup");
13  });
```

It only renders the view we created in the `/views/auth` folder. The `POST` route is more complicated. We have to do a few things:

- Get the data that the user has inserted in the form
- Check out that he has correctly filled up both fields
- Check out if the username already exists in the database
- If none of the conditions above happen, we can save the user

```
authController.post("/signup", (req, res, next) => {
  var username = req.body.username;
  var password = req.body.password;

  if (username === "" || password === "") {
    res.render("auth/signup", {
      errorMessage: "Indicate a username and a password to sign up"
    });
    return;
  }

  User.findOne({ "username": username }, "username", (err, user) => {
    if (user !== null) {
      res.render("auth/signup", {
        errorMessage: "The username already exists"
      });
      return;
    }

    var salt     = bcrypt.genSaltSync(bcryptSalt);
    var hashPass = bcrypt.hashSync(password, salt);

    var newUser = User({
      username,
      password: hashPass
    });

    newUser.save((err) => {
      if (err) {
        res.render("auth/signup", {
          errorMessage: "Something went wrong when signing up"
        });
      } else {
        // User has been created...now what?
      }
    });
  });
});

module.exports = authController;
```

You can see how we send an `errorMessage` inside the second parameter of the `render` method to indicate why the sign up failed. We have to show this message in the layout:

```
<% if (typeof(errorMessage) !== "undefined") { %>
  <div class="error-message"><%= errorMessage %></div>
<% } %>
```

To finish up this section, we have to add the controller into the `app.js` file, as follows:

```
const authController = require('./routes/authController');
// ..other code
// Routes
app.use("/", authController);
```

If we launch the website with `npm start`, we will be able to find our form in the `http://localhost:3000/signup` (http://localhost:3000/signup) URL. We should be able to create our first users :)

Login

Now we have already created our first user, we can access to the platform. We need to create the login functionality to do that. Once we have logged in, we have to implement some logic to be able to know we are already authenticated. We will use sessions and cookies to do that.

Session and Cookies

We will use `express-session` to create a session and save the data of the logged user. We will also use `connect-mongo` to create a cookie and store it as a session backup in the database:

```
$ npm install --save express-session
$ npm install --save connect-mongo
```

Layout

The first step is to create the layout that the user will use to send his credentials, username and password, to the server. We will create the following laoyut in the `/views/auth/login.ejs` file:

```
<h2>Login</h2>

<form action="/login" method="POST" id="form-container" class="login">
  <label for="username">Username</label>
  <input type="text" name="username" placeholder="JonSnow">
  <br><br>
  <label for="password">Password</label>
  <input type="password" name="password" placeholder="Your password">
  <br><br>
  <button>Sign in</button>

  <p class="account-message">
    Don't have an account? <a href="/signup">Sign up</a>
  </p>
</form>
```

As you can see, we have added a link under the signup form to let new users create an account if they don't have one. We should do the same in the sign up form. We can add the link in the `/views/auth/signup.ejs` file:

```
15  <p class="account-message">
16    Do you already have an account? <a href="/login">Login</a>
17  </p>
```

Routes

Now we can create the routes to allow the users to log in the app. Again, we need to create a `GET` and a `POST` over the route `/login`.

In the first one, we will show the layout we created in the previous step.

In the second one we will check out if the user has inserted his data correctly, and we will save his data in the session if he logs in successfully.

As the login functionality is also related with authorization, as signup, we will add the following code in the `/routes/authController.js` file:

```
authController.get("/login", (req, res, next) => {
  res.render("auth/login");
});

authController.post("/login", (req, res, next) => {
  var username = req.body.username;
  var password = req.body.password;

  if (username === "" || password === "") {
    res.render("auth/login", {
      errorMessage: "Indicate a username and a password to log in"
    });
    return;
  }

  User.findOne({ "username": username },
    "_id username password following",
    (err, user) => {
      if (err || !user) {
        res.render("auth/login", {
          errorMessage: "The username doesn't exist"
        });
        return;
      } else {
        if (bcrypt.compareSync(password, user.password)) {
          req.session.currentUser = user;
          // logged in
        } else {
          res.render("auth/login", {
            errorMessage: "Incorrect password"
          });
        }
      }
  });
});
```

To show the error messages if it's necessary, we have to add the corresponding tags in the HTML:

```
<% if (typeof(errorMessage) !== "undefined") { %>
  <div class="error-message"><%= errorMessage %></div>
<% } %>
```

We are ready to login! Try to fill up the form and see what happens.

Auth redirections

As you can see, when a new user signs up or an existent user logs in, nothing happens. We need to add two different redirections to create a flow in our application that allow the users to:

- First: sign up
- Second: log in

First of all, let's add the redirection from the sign up to the log in. Once we have created the account, the next step for the user is to log in. We will redirect them from sign up to log in. In the `POST` of the sign up, we will add a redirection to the log in.

Note that we put a comment in the code to indicate that the user was already created. Here is where we have to put the redirection.

```
51  newUser.save((err) => {
52    if (err) {
53      res.render("auth/signup", {
54        errorMessage: "Something went wrong when signing up"
55      });
56    } else {
57      res.redirect("/login");
58    }
59  });
```

Following the same logic, once the users log in, they should be redirected to their own page, where they can find their tweets. We will create this page later in this learning unit, so we can add the redirection and comment it until we need it.

```
72  if (bcrypt.compareSync(password, user.password)) {
73    req.session.currentUser = user;
74    // res.redirect("/tweets");
75  } else {
76    res.render("auth/login", {
77      errorMessage: "Incorrect password"
78    });
79  }
```

We should add one more redirection. What happens when we visit the root of the website? Nothing! We should redirect the users to the login page, so they can log in with their credentials. We will add the following code into the `authController` file:

```
11  authController.get("/", (req, res) => {
12    res.redirect("/login");
13  });
```

We have covered the whole process of authorization. There is just one step pending to cover: the logout.

Logout

In the logout we will have to destroy the session. It's not a good practice to create a `GET` method to do that, but for this exercise will be enough. In the `authController` we have to add the following:

```
80  authController.get("/logout", (req, res, next) => {
81    if (!req.session.currentUser) { res.redirect("/"); return; }
82
83    req.session.destroy((err) => {
84      if (err) {
85        console.log(err);
86      } else {
87        res.redirect("/login");
88      }
89    });
90  });
```

As you can see, we check out if we have started a session before destroy it. If we don't, we redirect the user to the login to let them log in.

Now we are ready to start tweeting!

Tweets

Now that we have the basic authentication setup, we should start with the actual functionality of our Tweeter. The steps to add the functionality are:

1. Create the Schema in Mongoose so our users will have tweets
2. We will create a `New Tweet` form and controller action
3. We will show `My tweets`

Ready?

Data Model

In order to add Tweets to our users, we will need to store them inside a new collection `tweets`. In mongoose, we need to create a new model to access it easily from our code.

Each tweet will contain a few attributes:

- `tweet` : *String* containing the user's tweet (140 characters)
- `user_id` : *ObjectId* that relates a tweet to a user
- `user_name` : *String* with the user-name (so we don't have to query it each time)
- `created_at` : The time of creation of the tweet
- `updated_at` : The time the tweet was last updated

Let's create a `tweet.js` model:

```
$ touch models/tweet.js
```

The `tweet model` will first define a Mongoose schema (`tweetSchema`), then create a `Tweet` model and finally export it so we can use it in our application:

```
const mongoose = require("mongoose");
const Schema   = mongoose.Schema;

const tweetSchema = new Schema({
  tweet: {
    type: String,
    required: [true, "Tweet can't be empty"]
  },
  user_id: {
    type: Schema.Types.ObjectId,
    ref: "User"
  },
  user_name: String,
}, {
  timestamps: { createdAt: "created_at", updatedAt: "updated_at" }
});

var Tweet = mongoose.model("Tweet", tweetSchema);
module.exports = Tweet;
```

After creating the `Tweet` model, we can use it by simply requiring the module and assigning it to a *const*. For example, we could use the model to access our mongo database:

> ❗ You don't need to add the following code your project

```
const Tweet = require("./models/tweet");

mytweetTweet = new Tweet({
  tweet: "My first Tweet",
  user_id: "1",
  user_name: "Ironhacker"
});

mytweetTweet.save(function(err) { /* .. etc .. */ });
```

Tweets controller

Now that we have the model defined, we can start by creating a **tweet Controller** that will manage all the actions for managing the tweets. Also, we will create a `tweets` folder inside our views, so we can group all the html related to our controller:

```
$ touch routes/tweetsController.js
```

Our controller will have, for now, only an `index` method that will show our username. We will get the username from `req.session.currentUser.username` :

```
const express          = require("express");
const tweetsController = express.Router();

tweetsController.get("/", (req, res, next) => {
  res.render(
    "tweets/index",
    { username: req.session.currentUser.username}
  );
});

module.exports = tweetsController;
```

And lastly, we will create our `views/tweets/index.ejs` view:

```
<div id="container">
  <h2>Hi @<%= username %>!</h2>
</div>
```

The last step is to include our new Tweets Controller in our `app.js` . We will mount this controller in `/tweets` ; this means that all the routes defined in the controller will start from `/tweets` :

```
const tweetsController = require("./routes/tweetsController");

app.use("/tweets", tweetsController);
```

Now that we have the basic structure done, we can test that it works by going to http://localhost:3000/tweets (http://localhost:3000/tweets)

> ❗ Make sure you're logged in or this will throw an error!

Protecting Routes

If we are not authenticated and we try to access the `/tweets` route, we will get an error; We can't find *username* because there is not a session defined.

We can add a basic middleware to our `tweetsController` to ensure that all actions must have an authenticated user; otherwise we will redirect to the login page:

```
tweetsController.use((req, res, next) => {
  if (req.session.currentUser) { next(); }
  else { res.redirect("/login"); }
});
```

New tweet

Adding the *New Tweet* functionality to our application has two parts:

1. Route/View to show the new tweet form
2. Route to receive the form post and create a tweet

GET: New Tweet Form

To create a new tweet we add a route in our controller:

```
tweetsController.get("/new", (req, res, next) => {
  res.render("tweets/new",
    { username: req.session.currentUser.username });
});
```

We should add a `views/tweets/new.ejs` view with a form that contains the tweet input

```
<div id="container">
  <h2>Hi @<%= username %>!</h2>

  <form action="/tweets" method="POST" id="new-tweet">
    <label for="tweetText">Tweet</label>
    <input type="text" name="tweetText" maxlength="140"
      placeholder="What's going on?" >
    <button class="button blue fright">Tweet!</button>
  </form>

  <% if (typeof(errorMessage) != "undefined") { %>
    <div class="error-message"><%= errorMessage %></div>
  <% } %>
</div>
```

This view will POST the `tweetText` to a new action (`/tweets`) that will receive the tweet and add it to the database.

POST: Create New Tweet

We need to create the action that will receive the new tweet form post. We only allow authenticated users to create posts in their own account.

This means we need to find the authenticated user, so we will add the `User` and `Tweet` models to our controller:

```
// tweetsController.js
// Models
const User  = require("../models/user");
const Tweet = require("../models/tweet");
```

And now we will add the logic to save the tweet:

1. Find the `currentUser` object
2. Creates a `newTweet` tweet instance and fills the information
3. Saves ito to the database
4. If OK: Redirects to `/tweets`
5. If Not OK: adds an `errorMessage` and render `tweets/new`

```
tweetsController.post("/", (req, res, next) => {
  const user  = req.session.currentUser;

  User.findOne({ username: user.username }).exec((err, user) => {
    if (err) { return; }

    const newTweet = Tweet({
      user_id:   user._id,
      user_name: user.username,
      tweet:     req.body.tweetText
    });

    newTweet.save((err) => {
      if (err) {
        res.render("tweets/new",
          {
            username: user.username,
            errorMessage: err.errors.tweet.message
          });
      } else {
        res.redirect("/tweets");
      }
    });
  });
});
```

My tweets

Now that wew have tweets, we should show them in our `tweets` page!

All we need to do is modify our controller to:

1. Find the current user
2. Find all her tweets, sort them by descending creation date
3. Pass them to our `index` view

```
const moment = require("moment");
// other code
tweetsController.get("/", (req, res, next) => {
  User
    .findOne({ username: req.session.currentUser.username }, "_id username")
    .exec((err, user) => {
      if (!user) { return; }

      Tweet.find({ "user_name": user.username }, "tweet created_at")
        .sort({ created_at: -1 })
        .exec((err, tweets) => {
          console.log("tweets");
          res.render("tweets/index",
            {
              username: user.username,
              tweets,
              moment });
        });
  });
});
```

And our `index` view will just iterate over all the tweets.

```
<div id="container">
  <a href="/tweets/new">New tweet</a>
  <a href="/logout">Logout</a>

  <h2>@<%= username %> tweets</h2>

  <% tweets.forEach(function(tweet) { %>
    <div class="tweet-container">
      <p><%= tweet.tweet %></p>
      <p class="date"><%= moment(tweet.created_at).format("LL, LTS") %></p>
    </div>
  <% }) %>
</div>
```

We use MomentJS (https://momentjs.com/) to format our dates. To install it simply install the npm package and require it:

```
$ npm install --save moment
```

💡 Notice that in order to use `moment` in your views, you need to require them from your controller and then pass it to the views as a parameter

Summary

In this learning unit we have seen how to create mongoose models based on what we want to build in our website. We have seen how to structure a project to separate the functionalities in different files. We have also created sessions and saved them in the database as a backup, and we created the necessary code to publish tweets from our profile.