Homework #6

Authentication and Mongoose Models

Overview

Repository Creation

👀 Click on this link: https://classroom.github.com/a/p18TO6FG (https://classroom.github.com/a/p18TO6FG) to accept this assignment in GitHub classroom.

- This will create your homework repository
- Clone it once you've created it

Description

Login, Registration, and Multiple Models

Create a link aggregator site (*like* reddit, hacker news, etc.) that supports user registration and login… along with the ability to post articles (links).

Registering or logging in will create an authenticated session that contains all of the logged in user's information. Some elements on pages will only appear when a user is logged in. Some pages will redirect to login if a user arrives unauthenticated.

Goals

- use `bcrypt.js` (argon2 (https://www.npmjs.com/package/argon2) is suggested by owasp (https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html), but currently the nodejs module for argon2 is not a straightforward install) to salt and hash a password and to compare a hash to a plain text password
- use the slides on authentication (../slides/16/auth.html) to implement login and registration
- use express-session to store user data / an authenticated session
- use related documents to model users and posted articles
- extract path components to determine what data to use to render a page

⚠️⚠️⚠️ WARNING ⚠️⚠️⚠️

This homework is for learning purposes only; **do not use it** for authentication on a deployed site

- our application will only be served locally, and consequently, it will not be served over an encrypted connection … and - related - cookies aren't set to secure
- it will allow case sensitive usernames
- not all errors accounted for or handled gracefully
- system level errors and user errors may not be distinguishable
- some error messaging may reveal **too much** information
- user interaction and error messaging will be minimal (for example, successful login should redirect to page that required login)
- some error messages reveal will info about the existence of a user
- our session secret will be in version control

Features

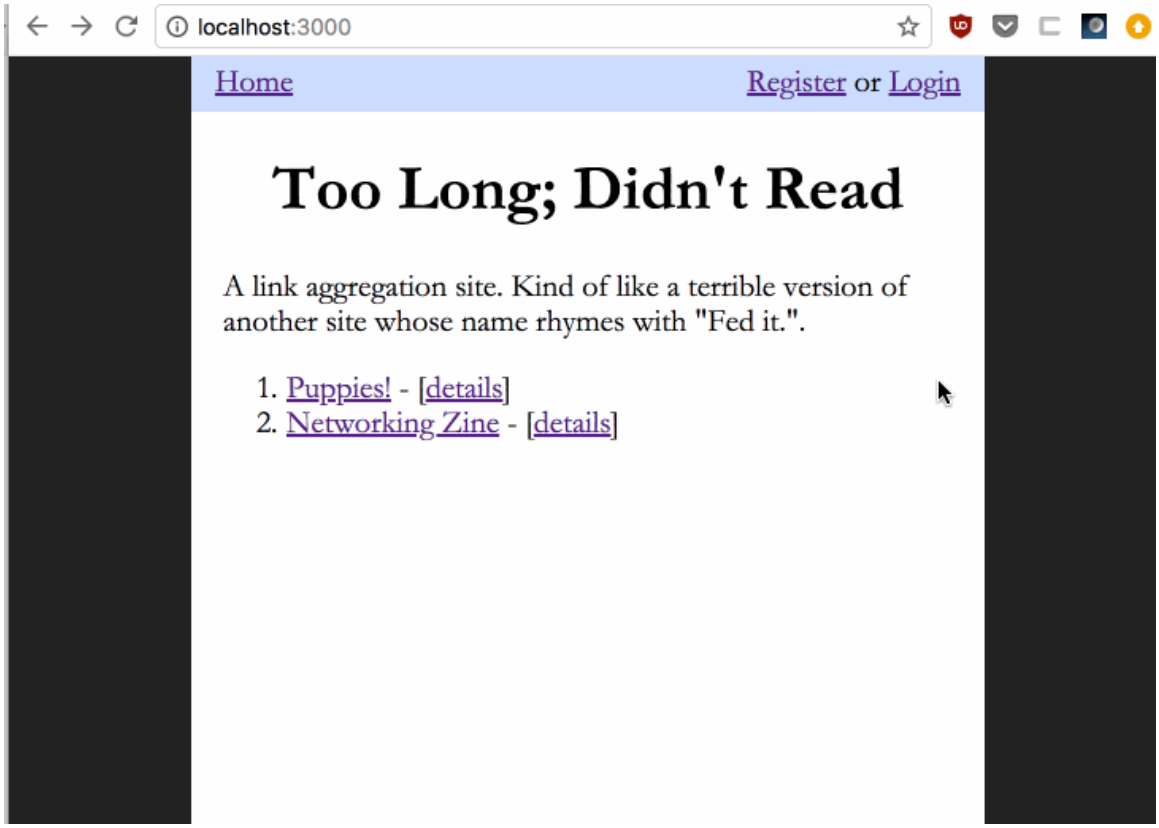
1. register a new account
2. login using an existing account
3. add a new article
4. view all posted articles
5. show a single article's details
6. prevent / allow access to certain ui elements or pages based on authenticated status

You'll have 5 pages and 3 forms:

- **/** - lists all articles
- **/register** - register form
- **/login** - login form
- **/article/add** - add new article form
- **/article/:slug** - detail page for a specific article

Example Interaction

Here's what it looks like to login, add a new article, and view the link and details page:.



Submission Process

You will be given access to a private repository on GitHub. The repository will have a partially built Express application. The final version of your assignment should be in GitHub:

- **Push** your changes to the homework repository on GitHub.

Make at Least 4 Commits

- Commit multiple times throughout your development process.
- Make at least 4 separate commits - (for example, one option may be to make one commit per part in the homework).

Part 1 - Setup and Authentication Functions

Starting Project

Your repository should have the following files and directories.

```
├── package.json
├── src
│   ├── app.mjs
│   ├── auth.mjs
│   ├── db.mjs
│   ├── public
│   │   └── css
│   └── views
│       ├── article-add.hbs
│       ├── login.hbs
│       ├── register.hbs
└── test
    ├── mock-user.mjs
    └── test-auth.mjs
```

Add the following:

- `.gitignore` - ignore `node_modules` and any other files that you would like to keep out of your repository (varies depending on your OS and editor… for example, if you're on vim on MacOS, you may want to ignore `.swp` files and `.DS_Store` etc.)
- `.eslintrc.cjs` - you can use a previous configuration for this if it's not present or if you'd like to overwrite the default linting config

Dependencies

The following modules are already listed in `package.json` and configured in `app.js` … so simply run `npm install` in the project root

- `express`
- `hbs`
- `express-session` - for session management
- `mongoose` - for database access
- `bcryptjs` - module for salting, hashing and comparing passwords (note - you can use argon2 instead)
- `mongoose-slug-plugin` - plugin for autogenerating slugs

Your app is configured to listen on port 3000. Do not change this.

About bcrypt.js

bcrypt.js (argon2 is recommended by owasp, but it is currently more difficult to install). `bcrypt.js` is a JavaScript implementation of password hashing function called (you guessed it!) bcrypt. We'll use it for login and registration. The result of using bcrypt contains both the hash and the salt! Check out the details in the first section of the wikipedia article (https://en.wikipedia.org/wiki/Bcrypt) and the diagram below illustrating the output of bcrypt:

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
|_____||_____________________||_______________________________|
   |             |                          |
   |             |                          +-- hash
   |             |
   |             +-- salt
   |
   +-- algorithm and cost factor
```

Check out how to use bcrypt.js (https://www.npmjs.com/package/bcryptjs).

Database Setup

- in `src/db.mjs` ....
- using mongoose, **create two schemas**:
  - `UserSchema`
    - it should have the following fields (all `String`, and all required):
      - `username` - user name used for login and public display
      - `email` - user's email
      - `password` - the combined salt and hash * username should be a unique field, so set as `{type: String, unique: true, required: true}` * typically hash and salt should be stored in separate fields, but because we're using `bcrypt` a single field is adequate for both
  - `ArticleSchema`
    - it should have the following fields (all `String`, validation is your discretion)
      - `title`
      - `url`
      - `description`
    - use `mongoose-slug-plugin` to add a slug to uniquely identify each article:
      - this plugin will autogenerate a `slug` field (no need to explicitly add it to your schema)
      - a slug is a string that serves as a short, human readable name
      - usually contains dashes to separate words, and a number suffix
      - for example, `this-is-a-slug`
      - `import` the module: `import mongooseSlugPlugin from 'mongoose-slug-plugin';`
      - use the plugin by adding this code: `<your schema name>.plugin(mongooseSlugPlugin, {tmpl: '<%=title%>'})'));`
      - this should go **before the model is registered** with `mongoose.model(...)`
      - you must use [referenced documents with population] (https://mongoosejs.com/docs/populate.html_ – rather than embedded documents (https://mongoosejs.com/docs/subdocs.html) </code></pre>
  - **again you must associate users and articles** by using references to documents in another collection:
    - when relating documents, add a `User`'s id to `ArticleSchema`
    - see the mongoose docs (https://mongoosejs.com/docs/populate.html) on references and population
- don't forget to register your models:
  - `mongoose.model('User', UserSchema);`
  - `mongoose.model('Article', ArticleSchema);`
- your db.js is configured to connect to `hw05`:
  - `mongoose.connect('mongodb://localhost/hw05');`

Create Helper Functions

Create the following functions in a module, `src/auth.mjs`:

1. `register`
2. `login`
3. `startAuthenticatedSession / endAuthenticatedSession`

You'll use these functions in your express application to implement a login and registration page.

Warning: the following functions will contain many levels of nested callbacks!

- we haven't covered promises yet, but if you want to remove nesting, using promises is one solution (though you'll have to research this on your own): both `mongoose` and `bcrypt` provide promises (instead of callbacks) as part of their api
- another way of dealing with this is wrapping some functionality in a function, but if you do this, you'll need to write a function that takes a callback!

register

```
register(username, email, password, errorCallback, successCallback)
```

Parameters:

- `username`: username
- `email`: email
- `password`: plain text password
- `errorCallback`: function to call if an error occurs
  - `errorCallback(errObj)`
  - `errObj` will have a key called message that contains an error message
- `successCallback`: function to call if registration works
  - `successCallback(userObject)`
  - `userObject` is the object that represents the newly saved user

Return:

- no return value (instead, calls callback functions, `errorCallback` or `successCallback`)

Description:

1. when you encounter any errors as you run through the registration process:
   - log the error to the console (server)
   - call the `errorCallback` passing in an error object containing a key called `message` and a value of a string describing the error that occurred (this gives the caller the ability to pass in a function that handles the error - for example, rendering a template with an error message)
   - some errors will require specific strings to be in the `message` property of the error object, while others are your discretion (read the instructions and run the tests for details)
2. check the length of the username and password passed in; they should both be greater than or equal to 8
3. if either the username or password does not meet this requirement, call the `errorCallback` function with an object containing a key called `message` and value of `USERNAME PASSWORD TOO SHORT`
4. check if the user already exists (case sensitive check is ok)
   - remember to pull out your `User` model by using `const User = mongoose.model('User');`
   - use `User.findOne((err, result) => { })` to check if the user already exists
     - you can check the object with `if(result)` to determine if a `User` object was returned
   - or use `User.find((err, result) => { })` ... the

- you can check if the length of the resulting `Array` is greater than 0

5. if the user already exists, call the `errorCallback` function with an object containing a key called `message` with the value, `USERNAME ALREADY EXISTS`
6. if the user doesn't exist yet, then it's ok to go ahead and create a new user…
7. salt and hash the password using the `bcryptjs` module
   - check out the documentation on the bcrypt module (https://www.npmjs.com/package/bcryptjs#usage---async)
   - do not use the sync calls (use the async version of bcrypt)
   - auto generate a hash and a salt:

```
// you can use a default value of 10 for salt rounds
bcrypt.hash(myPlaintextPassword, saltRounds, function(err, hash) {
    // do more stuff here!
});
```

8. notice the `hash` parameter in the callback above; it *actually* contains both the hash and the salt
9. now we have everything we need to create a new user
   - instantiate a new `User` object
   - set the `username` and `email` to whatever was passed in as arguments, and the `password` should be set to the salt and hash generated
   - call `save`
   - if the `save` succeeds, call the `successCallback` function with the newly saved user
   - otherwise call the `errorCallback` with an object that contains the key, `message`, and a generic error message, `DOCUMENT SAVE ERROR`, as the value
10. check that your function works by:
    1. running the unit tests, `npx mocha --loader=mock-import test/test-auth.mjs` (comment out all tests except for registration related ones), to check for:
       - a user object being passed to the success callback
       - the user object having the same name and email as the original arguments passed in to the `register` function
       - the password hash in the user object being the same as the hashed version of the original password passed in to the `register` function
       - the error callback being called if a duplicate user is found (`USERNAME ALREADY EXISTS`)
       - the error callback being called if a save issue occurred (`DOCUMENT SAVE ERROR`)
    2. writing a small script to test if running the function actually creates a user in the database!

```
import '../src.db.mjs';
import * as auth from '../src/auth.mjs';
auth.register('testtest', 'test@test.test', 'testtest',
  function(err) {console.log(err);},
  function(user) {console.log(user);}
);
```

Example Usage:

```
function success(newUser) {
  // start an authenticated session and redirect to another page
}
function error(err) {
  // render a template containing an error message
}
auth.register(req.body.username, req.body.email, req.body.password, error, success);
```

login

`login(username, password, errorCallback, successCallback)`

Parameters:

- `username` : username
- `password` : plain text password
- `errorCallback` : function to call if an error occurs
  - `errorCallback(errObj)`
  - `errObj` will have a key called message that contains an error message
- `successCallback` : function to call if login is successful
  - `successCallback(userObject)`
  - `userObject` is the object that represents the newly saved user

Return:

- no return value (instead, calls callback functions, `errorCallback` or `successCallback`)

Description:

1. when you encounter any errors as you run through the login process:
   - log the error to the console (server)
   - call the `errorCallback` passing in an error object containing a key called `message` and a value of a string describing the error that occurred (this gives the caller the ability to pass in a function that handles the error - for example, rendering a template with an error message)
   - some errors will require specific strings to be in the `message` property of the error object, while others are your discretion (read the instructions and run the tests for details)
2. find the user with username that was passed in

```
User.findOne({username: username}, (err, user) => {
 if (!err && user) {
        // compare with form password!
 }
});
```

3. if the user doesn't exist, call the `errorCallback` function with an object containing a `message` field that has the value `USER NOT FOUND`
4. if the user exists… then check if the password entered matches the password in the database
   - the password in the database is salted and hashed… and contains the salt
   - so a simple compare with `===` is not adequate
   - salt and hash the password and compare with the hash stored in the database
   - use the function, `bcrypt.compare` to do this (see the docs (https://www.npmjs.com/package/bcryptjs#usage---async):

```
bcrypt.compare(password, user.password, (err, passwordMatch) => {
 // regenerate session if passwordMatch is true
});
```

   - note that `passwordMatch` within the callback will be either true or false, signifying whether or not the salted and hashed version of the incoming password matches the one stored in the database
5. once the match is verified, call the `successCallback` function with the user that was found
6. check that your function works by:
   1. running the unit tests, `npx mocha --loader=mock-import test/test-auth.mjs` (comment out all tests except for registration and login related ones), to check for:
      - the user object being passed to the `successCallback` when a login attempt works

    - the following values for the `message` property on the error object passed to the `errorCallback`: `PASSWORDS DO NOT MATCH` and `USER NOT FOUND`
2. writing a small script to test if running the function succeeds in logging a user in (that is, confirm that the success callback is called)

```
import '../src.db.mjs';
import * as auth from '../src/auth.mjs';
// assuming the user, testtest was already registered previously
auth.login('testtest', 'testtest',
  function(err) {console.log(err);},
  function(user) {console.log(user);}
);
```

Example Usage:

```
function success(newUser) {
  // successfully logged in!
  // start an authenticated session and redirect to another page
}
function error(err) {
  // render a template containing an error message
}
auth.login(req.body.username, req.body.password, error, success);
```

---

startAuthenticatedSession

`startAuthenticatedSession(req, user, callback)`

Parameters:

- `req`: an express Request object that contains a session variable
- `user`: the user data to store in the session
- `callback`: the callback to call after a new session has been created
    - `callback(err)` - the callback only has a single parameter, `err`

Return:

- no return value (instead, calls `callback` function)

Description:

1. regenerate a session id
    - use `req.session.regenerate` to do this (see the docs (https://www.npmjs.com/package/express-session#sessionregeneratecallback))
2. add the user object passed in to the session; it should minimally contain `username` and `_id`, but other other properties (like email) can also be present
3. test your code by running `npx mocha --loader=mock-import test/test-auth.mjs` (you can uncomment all commented out tests from previous runs)

Example Usage / Partial Implementation:

```
 // assuming that user is the user retrieved from the database
req.session.regenerate((err) => {
  if (!err) {
    // set a property on req.session that represents the user
  } else {
        // log out error
  }
  // call callback with error
});
```

```
// when calling ....
auth.startAuthenticatedSession(req, newUser, (err) => {
        // choose appropriate code path based on if there's an error
});
```

Aside on `endAuthenticatedSession`

To invalidate a session, there is a `destroy` method that can be called on `req.session`. This is an implementation:

```
function endAuthenticatedSession(req, cb) {
  req.session.destroy((err) => { cb(err); });
}
```

Additional Notes:

Once `startAuthenticatedSession` is called, you can:

1. check if someone is logged in by looking at the user object in req.session (`if(req.session.user.username)`)
2. add the user object to every context object for every template by using this middleware:
   - use `res.locals` (see express docs (https://expressjs.com/en/api.html#res.locals)) to do this:

     ```
     // add req.session.user to every context object for templates
     app.use((req, res, next) => {
       // now you can use {{user}} in your template!
       res.locals.user = req.session.user;
       next();
     });
     ```

   - in any template (or even in `layout.hbs`), you can use the following to conditionally display ui elements based on authenticated status:

     ```
     {{#if user}}
     <h1>something that can only be seen if logged in</h1>
     {{/if user}}
     ```

3. finally, to retrieve a username from the session, use: `req.session.user.usernmae`

Part 2 - Layout, Homepage, Registration and Login

In this part, you'll implement the following pages:

1. `/` - home
2. '/register` - registration form
3. '/login` - login form

The following routes should be supported:

1. `GET /`
2. `GET /register`
3. `POST /register`
4. `GET /login`
5. `POST /login`

Layout and Homepage

Verify the Layout File

Find and read `views/layout.hbs` ; it has markup for a header that contains the following information:

- if the user is logged in:)* the user's username
  - a link to `/article/add` (to be implemented in part 3)
- if the user is not logged in:
  - links to:
    - `/login`
    - `/register`
  - the link to add an article should not appear!
- for both logged in and non-logged in users:
  - a link to home, `/`

Create a Homepage

`/` should show the header implemented above. In part 3, this page will be used to display data from the database.

Registration Overview

Registration should allow a user to create a new account and immediately start a new authenticated session. To do this, follow these steps:

1. show a registration form
2. after submitting the form, salt and hash the password using bcrypt
3. save the username and salt/hash combination
4. regenerate the session (create a new session id)
5. add some information, such as the username, to the session
6. redirect to home, `/` , if registration is successful

Steps 2 through 5 are handled using the `register` function in the `auth` module that you implemented in Part 1.

Step 6 should be implemented by passing a success callback function to `register`

There are two routes for registration:

1. `GET /register` - to display the form
2. `POST /register` - to process the form input

GET /register Implementation

- find the route for `GET /register` in `app.mjs`
- it should render a template, `register.hbs` that contains a form (`register.hbs` should already exist in your `views` folder)
- the form in `register.hbs` will...
  - `POST` to `/register`
  - (that is, when you press the submit button, a POST request will be made to `/register`)
- if there is a registration error, `register.hbs` can be re-rendered with a message showing the error above the form

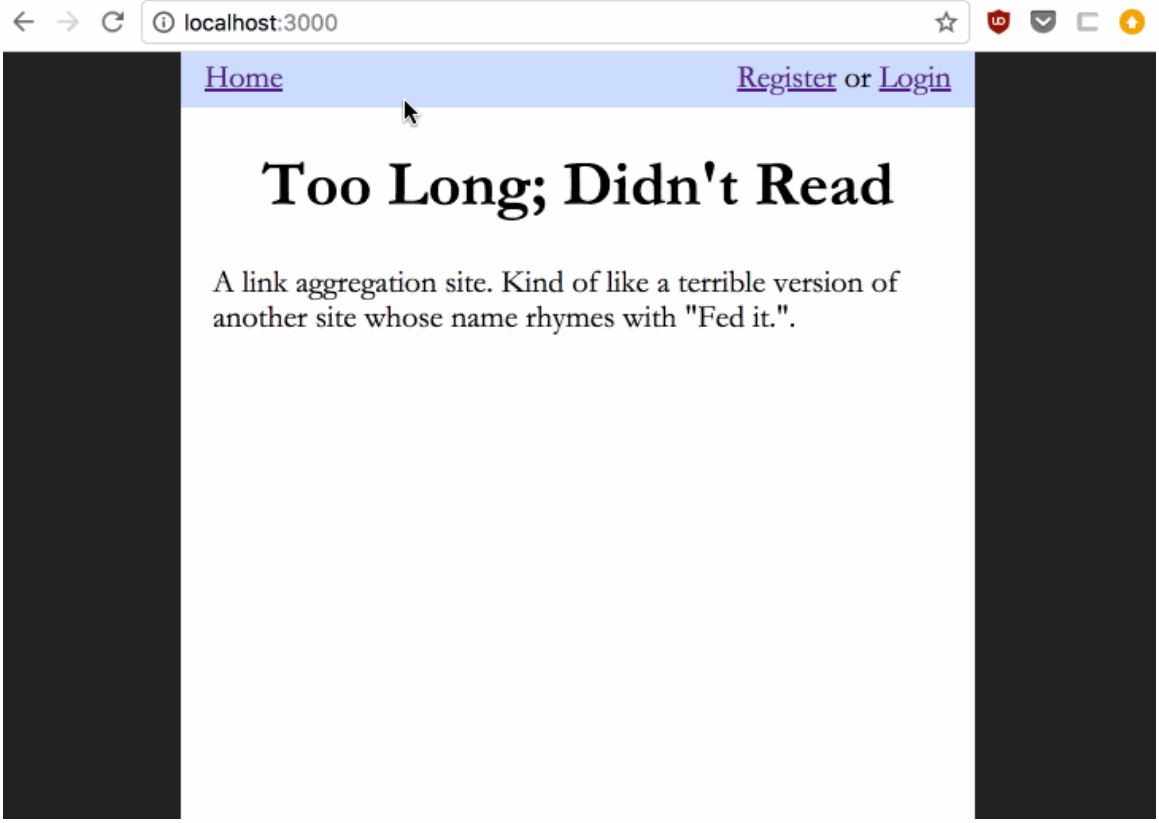
POST /register Implementation

- find the route for `POST /register` in `app.mjs`
- call your `register` function using the data from the POST request's body (provided by the form that the user filled out)

- pass two functions, error and success callbacks, to your call to `register`

success callback

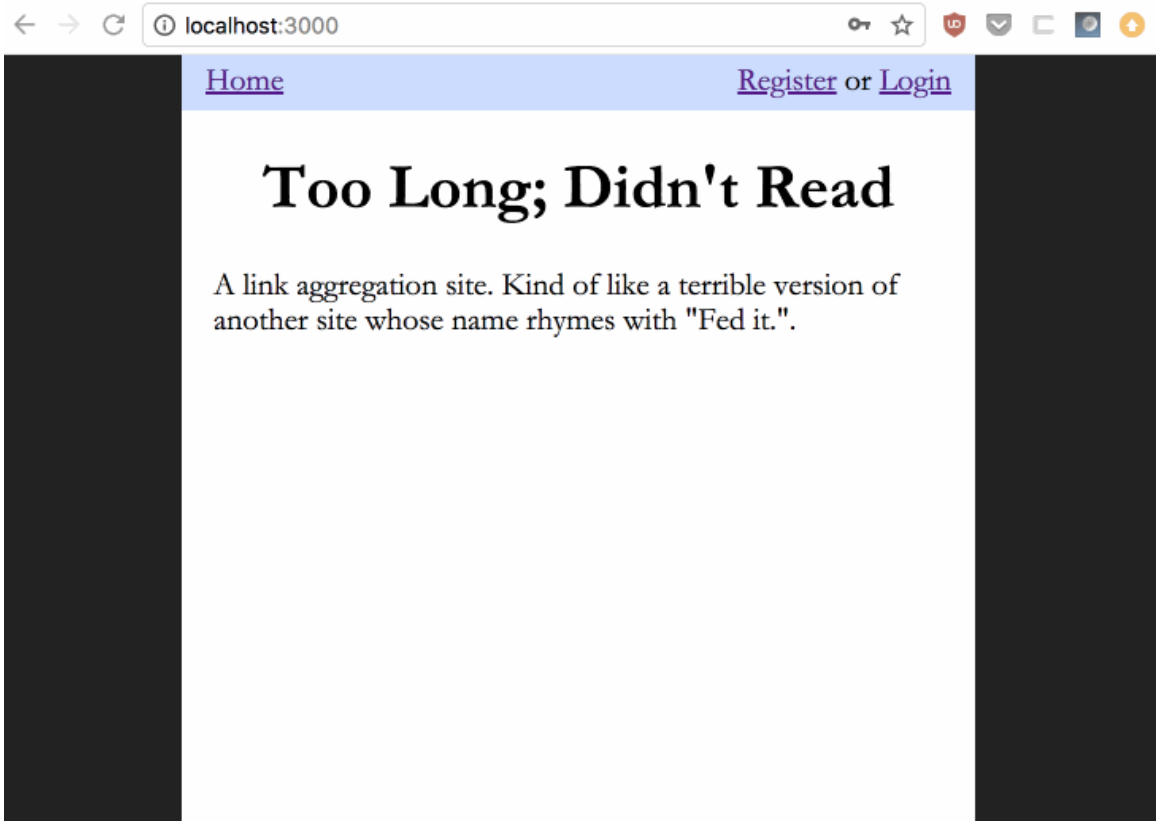
- the success callback is only called if the registration works
- so it can assume that the new user now exists
- …and consequently, it can start an authenticated session for the new user using the `startAuthenticatedSession` function from the `auth` implemented above
- in turn, once an authenticated session is started, *its* callback should redirect to home (`'/'`)
- here's what the resulting interaction might look like:



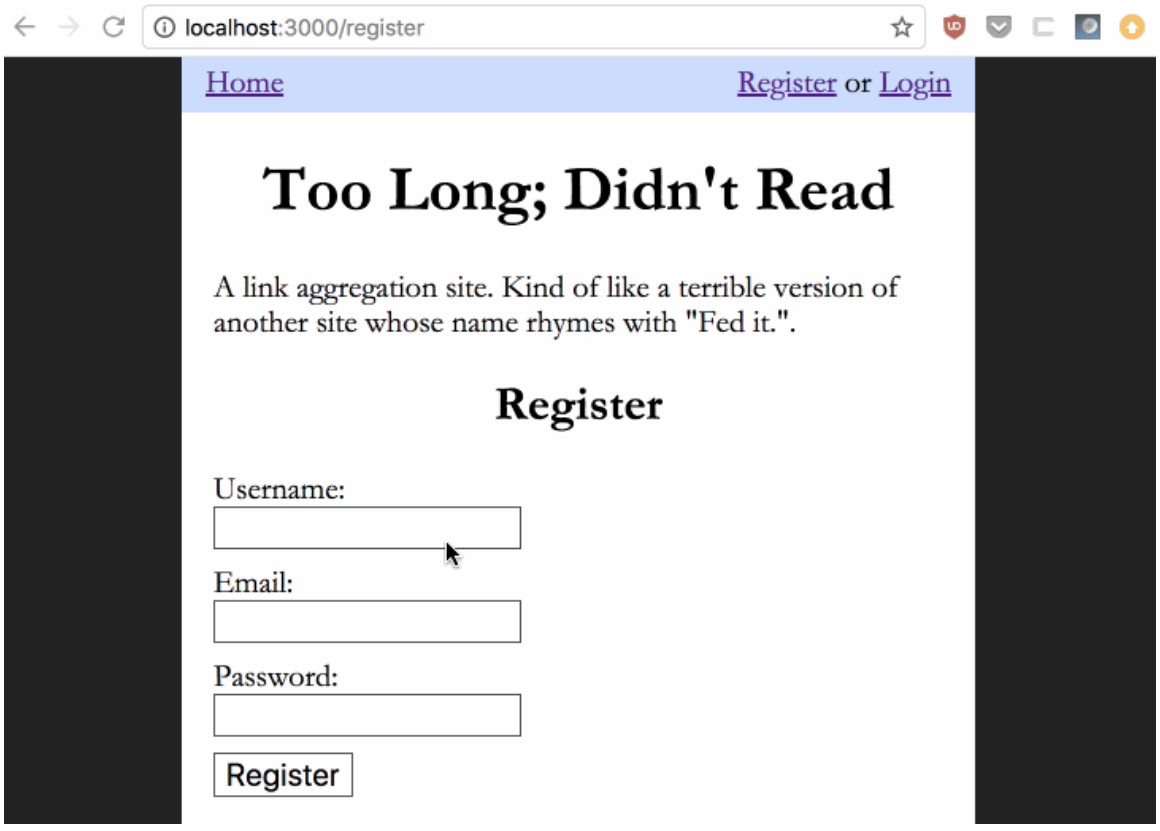
error callback

- if , the registration does not work, then rerender the register template with an error
- to do this, the error callback should use the response object to call `render`, and the context should supply an error message
- minimally, the following errors should be shown
  1. password length too short
  2. user already exists
- all other errors can simply display a generic error message
- see below for examples of registration errors:

- password or username length too short



- user already exists



Registration Testing

1. in your browser, got to your registration page
2. attempt to register a user
   - verify that you are redirected to the homepage

- use the mongo commandline client to check that you have user documents with username and password filled in

Login Overview

Login should allow a user to authenticate using a username and password. To do this, follow these steps:

1. show a login form
2. search the database for the username specified in the login form
3. after finding the user, salt and hash the incoming password and compare with the password in the database by using `bcrypt.compare`
4. if the passwords match then start a new authenticated session
5. redirect to the home page

Steps 2 through 4 are handled by the `login` function in the `auth` module implemnted in a previous part.

Step 5 should be implemented by passing a success callback function to `login`

There are two routes for login:

1. `GET /login` - to display the form
2. `POST /login` - to process the form input

GET /login Implementation

- find the route for `GET /login` in `app.mjs`
- it should render a template, `login.hbs` that contains a form (`login.hbs` should already exist in your `views` folder)
- the form in `login.hbs` will...
  - `POST` to `/login`
  - (that is, when you press the submit button, a POST request will be made to `/login`)
- if there is a login error, `login.hbs` can be re-rendered with a message showing the error above the form

POST /login Implementation

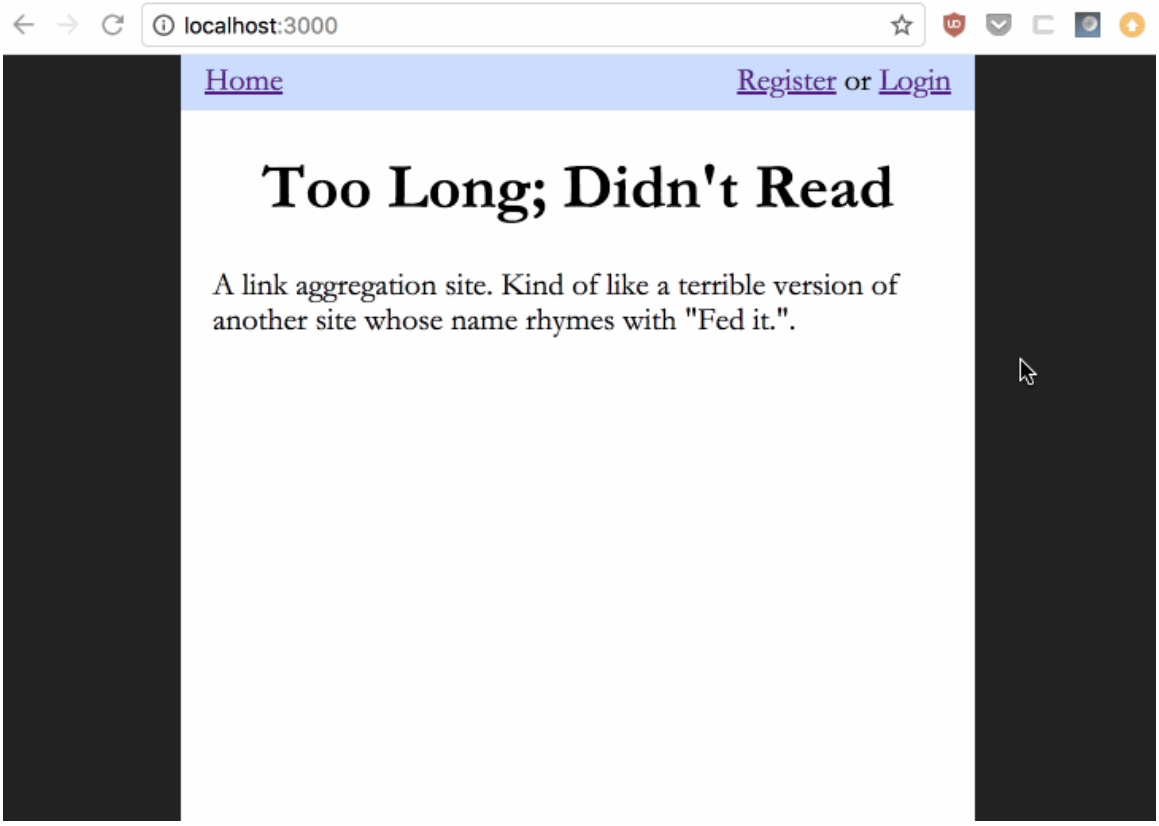
Handle the data POSTed by the login form…

- find the route for `POST /login` in `app.mjs`
- call your `login` function using the data from the POST request's body (provided by the login form that the user filled out)
- pass two functions, error and success callbacks, to your call to `login`

success callback

- the success callback is only called if the login works
- consequently, it can start an authenticated session for the logged in user using the `startAuthenticatedSession` function from the `auth` implemented earlier
- in turn, once an authenticated session is started, *its* callback should redirect to home (`'/'`)

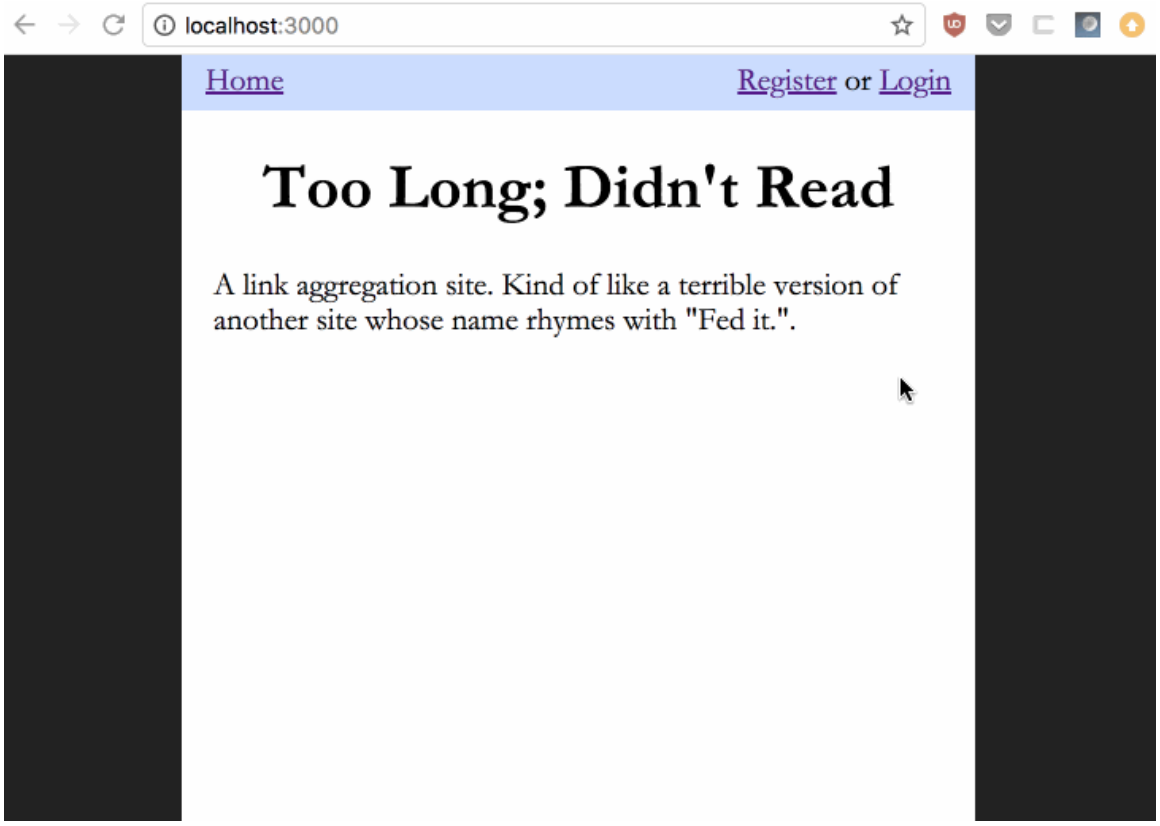
- here's what the resulting interaction might look like:



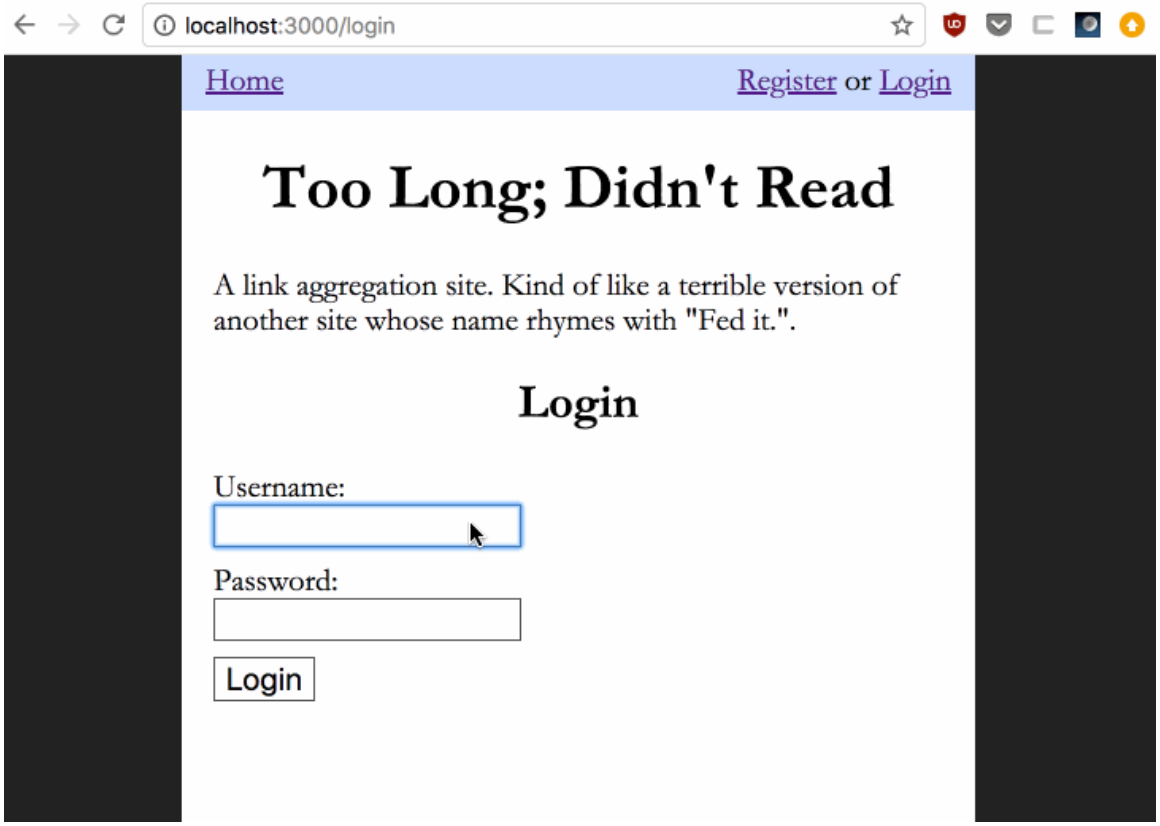
error callback

- if , the login does not work, then re-render the login template with an error
- to do this, the error callback should use the response object to call `render`, and the context should supply an error message
- minimally, the following errors should be shown
    1. user does not exist
    2. passwords do not match
- all other errors can simply display a generic error message
- see below for examples of login errors:

- user does not exist



- passwords to not match



Part 3 - Adding and Displaying Articles

Adding Articles

There are two routes for adding an article:

1. `GET /article/add` - to display the form
2. `POST /article/add` - to process the form input

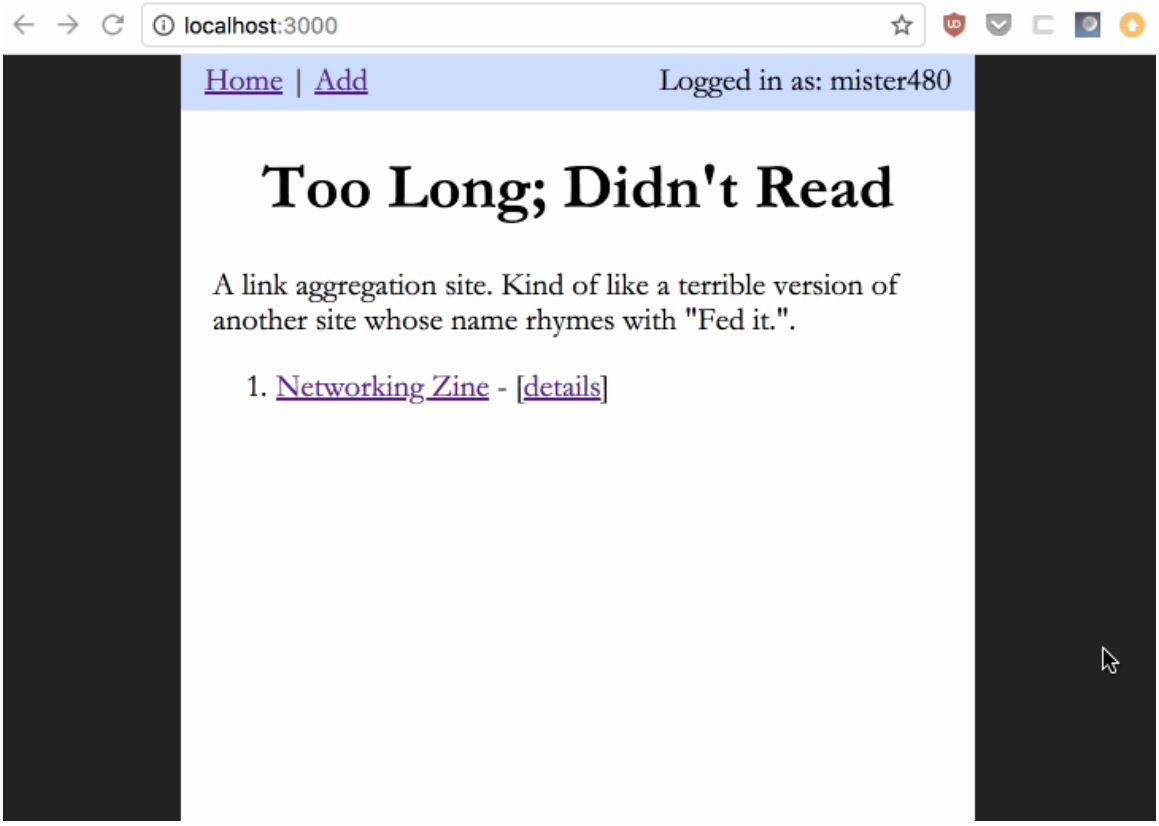
GET /article/add Implementation

- make sure that only logged in users can reach this url:
  - check `req.session.user`
  - redirect to `/login` if the user is not logged in
- find the route for `GET /article/add` in `app.mjs`
- it should render a template, `article-add.hbs` that contains a form (`login.hbs` should already exist in your `views` folder)
- the form in `article-add.hbs` will...
  - `POST` to `/article/add`
  - (that is, when you press the submit button, a POST request will be made to `/article/add`)
- it has the following fields:
  1. `title` - the title of the article
  2. `url` - the link to the article
  3. `description` - a short description of the article
- the following fields will be determined by your application
  1. `slug` - the unique, human-readable identifier for this article
  2. if you're using related documents, `user` - the user id associated with this article (if you're using embedded documents, you will find the logged in user in the database and add this article to it)

POST /article/add Implementation

- make sure that only logged in users can reach this url:
  - check `req.session.user`
  - redirect to `/login` if the user is not logged in
- create a new Article and associate it with a user
  - if you used related documents
    - remember to bring in your `Article` model: `const Article = mongoose.model('Article');`
    - you'll have to add the user id of the logged in user (this can be found in the `_id` property of the session data, `req.session.user`, that you have for the currently logged in user)
  - if you used embedded documents
    - you'll have find the user that created the article in the database
    - and push a new article into its list of embedded documents
- in either case if the article / user is saved successfully, redirect to the home page (`/`), and if there's an error, re-render the `article-add.hbs` and display an error message
- note that the `slug` field should automatically be added for you by the `mongoose-slug-plugin` plugin

- the entire interaction should look like this:



Testing

- go to `/article/add`
- make sure that you are redirected to login (if you're not logged in)
- register or login to your site
- go back to `/article/add`
- submit the form
- using the commandline client, check that a new article has been added!

Displaying All Articles

The homepage should display all of the articles added by users.

1. go back to your route handler and template for your home page, `/`
2. query the database for all articles posted, and drop the result into the context object when rendering the template.
3. in the template, display the title of the article…
4. make the title a link to the url of the article
   - for example, if `title` is `foo` and `url` is `http://bar.baz` … then the markup should be:
   - `<a href="http://bar.baz/>foo</a>`
5. additionally, link to a detail page for the article `/article/the-article-slug` ; the link text should be "details"
   - for example, if the `slug` field of the article is `qux-corge` … then the markup should be:
   - `<a href="/article/qux-corge">details</a>` … note that the slug part would have to be filled in by a template variable

Displaying an Article's Details

The detail page, `/article/the-article-slug` , should show all of the fields of an aritcle, including:

- title
- url (this should actually link to the url displayed)
- username of the user that added the article
- desription

Displaying All Articles and Article Details Example