Message.ly

Springboard

Download exercise <../express-messagely.zip>

Message.ly is a user-to-user private messaging app.

This exercise is meant to teach and reinforce useful common patterns around authentication and authorization.

Step 0: Setup

- Install requirements, and make Git repo.
- Create ***messagely*** database and import schema from ***data.sql***

Step 1: Take a Tour

Many parts of this exercise are already given to you, and shouldn't need to change:

- ***app.js***
  - Pulls in user routes, messages routes, and auth routes
- ***expressError.js***
  - Handle errors in express more gracefully
- ***db.js***
  - Sets up ***messagely*** database
- ***server.js***
  - Starts server on 3000
- ***config.js***

  This may be a new file for us. As you build the app (and, in particular, the further study), you may add to it.

  Its job is to be a centralized place for constants needed in different places in the application. Other places should ***require()*** in these values.

  In order to make it easier to keep secret things secret, it also will try to read a file named ***.env***. This is a traditional name for a file containing "environmental variables needed for an application".

  If you create a file like this:

  .env

  ```
  SECRET_KEY = abc123
  ```

  This ***config.js*** file will read and use it.

- ***middleware/auth.js***

  Useful middleware for "is a user logged in?" and "is the logged-in user the same as the ***:username*** provided in a route?"

  Look carefully at this code — it may be slightly different than other versions you've seen. Make sure you understand what it is doing!

Step 2: Fix the user model

We've provided a module file for the ***User*** class:

models/user.js

```
/** User class for message.ly */
```

```
/** User of the site. */

class User {

  /** register new user -- returns
   *    {username, password, first_name, last_name, phone}
   */

  static async register({username, password, first_name, last_name, phone}) { }

  /** Authenticate: is this username/password valid? Returns boolean. */

  static async authenticate(username, password) { }

  /** Update last_login_at for user */

  static async updateLoginTimestamp(username) { }

  /** All: basic info on all users:
   * [{username, first_name, last_name, phone}, ...] */

  static async all() { }

  /** Get: get user by username
   *
   * returns {username,
   *          first_name,
   *          last_name,
   *          phone,
   *          join_at,
   *          last_login_at } */

  static async get(username) { }

  /** Return messages from this user.
   *
   * [{id, to_user, body, sent_at, read_at}]
   *
   * where to_user is
   *   {username, first_name, last_name, phone}
   */

  static async messagesFrom(username) { }

  /** Return messages to this user.
   *
   * [{id, from_user, body, sent_at, read_at}]
   *
   * where from_user is
   *   {username, first_name, last_name, phone}
   */

  static async messagesTo(username) { }
}


module.exports = User;
```

Fill in the method bodies.

Make sure you read the docstrings carefully so your functions return the right output. Also, any method that tries to act on a particular user (like the ***.get()*** method) should throw an error if the user cannot be found.

If you get stuck, note that the ***Message*** class has been completed for you. You can look to the methods there for some inspiration or assistance with some of the more complex queries.

Once you have finished, you can run the tests we've provided for the ***User*** and ***Message*** models (make sure to create and seed the ***messagely_test*** database first!):

```
$ jest -i
```

Step 3: Fix the routes

We've provided stub files and docstrings from the routes.

routes/auth.js

```
/** POST /login - login: {username, password} => {token}
 *
 * Make sure to update their last-login!
 *
 **/


/** POST /register - register user: registers, logs in, and returns token.
 *
 * {username, password, first_name, last_name, phone} => {token}.
 *
 *  Make sure to update their last-login!
 */
```

routes/users.js

```
/** GET / - get list of users.
 *
 * => {users: [{username, first_name, last_name, phone}, ...]}
 *
 **/


/** GET /:username - get detail of users.
 *
 * => {user: {username, first_name, last_name, phone, join_at, last_login_at}}
 *
 **/


/** GET /:username/to - get messages to user
 *
 * => {messages: [{id,
 *                 body,
 *                 sent_at,
 *                 read_at,
 *                 from_user: {username, first_name, last_name, phone}}, ...]}
 *
 **/


/** GET /:username/from - get messages from user
 *
 * => {messages: [{id,
 *                 body,
 *                 sent_at,
 *                 read_at,
 *                 to_user: {username, first_name, last_name, phone}}, ...]}
 *
 **/
```

routes/messages.js

```
/** GET /:id - get detail of message.
 *
 * => {message: {id,
 *               body,
 *               sent_at,
 *               read_at,
 *               from_user: {username, first_name, last_name, phone},
 *               to_user: {username, first_name, last_name, phone}}
 *
 * Make sure that the currently-logged-in users is either the to or from user.
 *
 **/


/** POST / - post message.
 *
```

```
 * {to_username, body} =>
 *   {message: {id, from_username, to_username, body, sent_at}}
 *
 **/


/** POST/:id/read - mark message as read:
 *
 *  => {message: {id, read_at}}
 *
 * Make sure that the only the intended recipient can mark as read.
 *
 **/
```

In order, implement these routes. Make sure to check security appropriately:

- anyone can login or register
- any logged-in user can see the list of users
- only that user can view their get-user-detail route, or their from-messages or to-messages routes.
- only the sender or recipient of a message can view the message-detail route
- only the recipient of a message can mark it as read
- any logged in user can send a message to any other user

Further Study

Write Tests for Routes

We've provided a commented-out test file for the authentication routes. Uncomment this and make sure it works.

Working from this as an example, build integration tests for your user and message routes.

Make a Straightforward Front End

Feeling more bootstrappy?

Make a simple front end for your application:

- a registration page
- a login page
- a page to see all of your messages (both to you and from you)
- detail page for a message
- a page to make a new message, with a drop-down of users.

Since we're building a JSON API, the front-end will need to use AJAX to make requests to our server and update the DOM once you have the response data.

Send SMS With New Message

Twilio <http://twilio.com> is a useful provider of telephony services, including the ability for your app to send SMS messages.

Add a feature where, when a message is created, the recipient of that message gets a SMS leeting them know they've received a message, like this:

> *You've received a message.ly from whiskey-the-dog!*

To do this, you'll need to:

- sign up for an account with Twilio *(free and no credit card needed)*
- get your ***accountSid*** and ***authToken*** from the Twilio console
- install the ***twilio*** npm library
- learn how to use the JavaScript API for Twilio (docs on their site!)
- add code to the Message class to send an SMS

> **Note: Can Only Really Send To Your Cell #**

The free trial account for Twilio only lets you actually send texts to the number you used when you created the Twilio account. So, to see this feature working in your app, you'll need to make sure the recipient user of a message has your cell phone number.

If you upgrade to a real, paid Twilio account, you could send anyone an SMS, but it costs about 1 cent/SMS.

Warning: Make sure to keep your *accountSid*/*authToken* secret!

Don't hard code either of these in your code!

Instead, study the ***.env*** file and the ***config.js*** file and come up with a way that these could never appear in your GitHub!

Make a SMS-backed Change Your Password Feature

Users sometimes forget their passwords. Add a feature that provides a route where a user can change their password without knowing the old password.

To do this:

- it should generate a random 6-digit code
- it should send the user that code via an SMS
- it should add that code in a database, with other information needed (the current timestamp? the username?)
- it should provide a route that accepts a username, that code, and a new password, and, if valid, changes the password to that)

Refactoring the classes

Note that both the ***User*** and ***Message*** classes don't actually create instances — they're closer to the ***Cat*** model than the ***Dog*** model when we first learned about our OOP approach to creating models in Node.

Think about which methods (if any) would make more sense as instance methods instead of static methods, and refactor your application accordingly. Similarly, if there are any instance methods that you think would make your life easier, write them and use them in the app!

Solution

See our solution <solution/>