In this exercise, you will extend exercise8 by adding a user model and implementing session authentication.

Setup: Download exercise9StartingCode.zip from Canvas and unzip it. Open this exercise using VS code and take a closer look. This is essentially a solution to exercise8. At the terminal, type npm install. Then, run the app to make sure that everything works.

**Part 1. Create a User model**

1. In the models directory, create a file named user.js.
2. In User.js, create a schema to model the user objects. The schema should contain the following fields:

* firstName: a String type required field
* lastName: a String type required field
* email: a String type required and unique field
* password: a String type required field

Compile a model from the schema and name the model User. Export the model.

1. When you finish, show your user.js to the instructor or the TA to get credit for this part.

**Part 2. User sign up**

In this part, you will implement the functionality to allow users to sign up an account. Users will use a form to submit data to the server.

1. In the views directory, create a new folder named users. In the users directory, create a new file named create.ejs. Instead of typing everything from scratch, you can copy and paste the content of the create.ejs file in the restaurants directory. Then modify the form to include the following fields:

* Label: First Name
* Input text field
* Label: Last Name
* Input text field
* Label: Email
* Input text field
* Label: Password
* Input text field
* Submit button

Make sure that the name of each input field is the same as the field in the schema. Further, the form will submit data to the ‘/users’ endpoint via an HTTP POST request.

1. In this step, you will set up the routes and the controller functions for user sign up. Start by creating the routes. In the routes folder, create a new file named userRoutes.js.
2. In userRoutes.js:

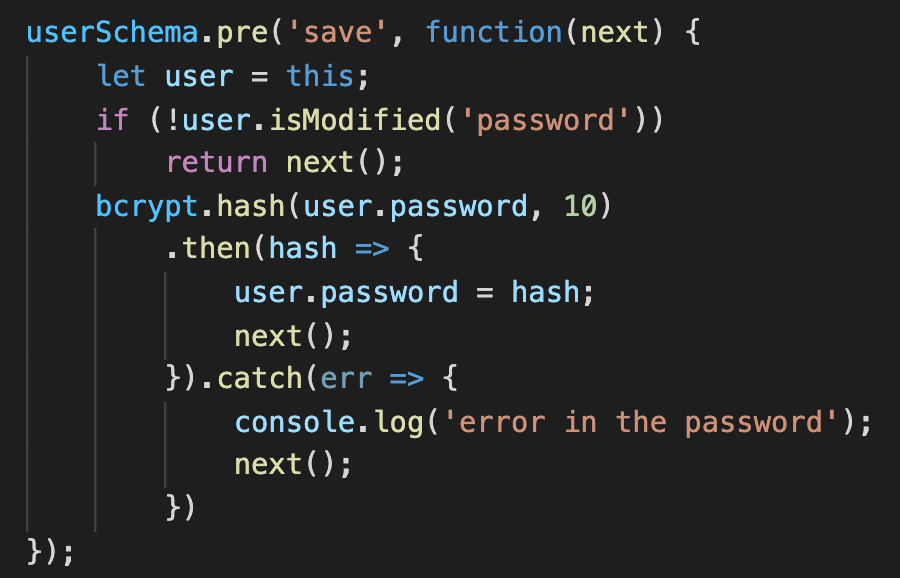
* require the express module
* create a Router object by calling express.router() and name it router
* require the User model

1. Add a new route handler to handle the GET request to the following end point: ‘/users/create’. This handler will render create.ejs in the views/users directory.
2. Export router. In app.js, require userRoutes and use it as a middleware, where the endpoint is set to ‘/users’. You will need to change the endpoints of the route handlers in userRoutes.js, ‘/users’ should be removed from the beginning of each endpoint.
3. To be able to test this out, add a new link to the registration page in the navigation. Open nav.ejs in the views/partials directory, copy and paste the following code in it:

<a href='/users/create'>Sign Up</a> |

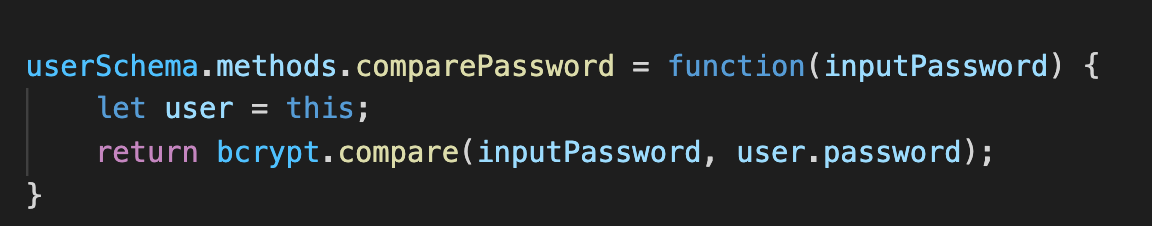
Start your app, click the sign up link to make sure that the form renders properly.

1. Next, add a new route handler to handle the POST request to the following endpoint: ‘/users’. In the route handler, the endpoint should just be ‘/’. create a new User object using the req.body parameter. Save the object in the database. Then, redirect the user to the home page.
2. Run your app, create a new user and submit. If everything works, you will be redirected to the home page. Check the database to make sure that the user is inserted in the collection. Take a closer look at the user in the database, you will see that the passwords are stored as plaintext. It is not safe to store users’ passwords this way because if the data is hacked, all of the passwords are leaked. Next, you will hash the password before saving it to the database. You will use Bcrypt to hash the passwords.
3. In the terminal, install bcrypt.
4. Open user.js in the models directory, require bcrypt. Then, in the same file, add the following lines of code. This should be placed before the model is created.



This code adds a pre middleware function in the user schema and it is executed before a document is being saved to the database. This function begins by checking the password of the user, if the password is not modified, it calls the next middleware. Otherwise, it calls the hash function which takes two arguments, the password the user entered, and the saltRound, which is a cost factor indicating the amount of time needed to calculate a single bcrypt hash. This function is asynchronous and it returns a promise. If the promise is resolved, it returns the hashed password, otherwise, it returns an error.

1. To be able to authenticate users in the next part, add the following code in user.js.



This code adds an instance method to the User model. Thus, every User object can call this method. This method calls the compare function, which takes two arguments, the password the user entered, and the hashed password stored in the database. This function is asynchronous and it returns a promise. If the promise is resolved, it returns the result of the comparison, a boolean type value, otherwise, it returns an error. You will call the comparePassword method in Part 3 when implementing the route handler for login.

1. Delete the users that you created in Step 8 from the database. Then, register a new user, check to make sure that the password stored in the database is a hashed password. Once everything works, demo your app to the instructor or the TA to get credit for this part.

**Part 3. User login**

In this part, you will implement the functionality to authenticate users’ login requests.

1. You will begin by creating the view template for logging in. Open create.ejs in the views/users directory, and save it as login.ejs. Then modify the form to include the following fields:

* Label: Email
* Input text field
* Label: Password
* Input text field
* Submit button

Make sure that the name of each input field is the same as the field in the user schema. Further, the form will submit data to the ‘/users/login’ endpoint via an HTTP POST request.

1. In userRoutes.js, add a new route handler to handle the GET request to the following end point: ‘/login. This handler will render login.ejs in the views/users directory.
2. To be able to test this out, add a new link to the login page in the navigation. Open nav.ejs in the views/partials directory, copy and paste the following code in it:

<a href='/users/login’>Login</a> |

1. Add a new route handler to handle the POST request to the following end point: ‘/login’. The handler will authenticate the user. If the user is authenticated, redirect to the home page; otherwise, redirect to the login page. Here is the pseudocode.

* From req.body, obtain the email and the password that user entered to login
* Query the database to find a user that matches the email, in the then clause:
  + If the query result is empty, the user account does not exist. Redirect to the login page
  + If the query result is not empty, authenticate the user by calling the comparePassword method on the user. Recall that we created this method in Part 2. This method returns a promise. In the .then clause, if the result is true, the user is authenticated, redirect the user to the homepage; otherwise, the password is not correct, redirect the user to the login page.
* In the catch clause, call next() in case any error happens during database query or authentication.

Start your app, click the login link to make sure that the form renders properly. Test the login functionality by entering

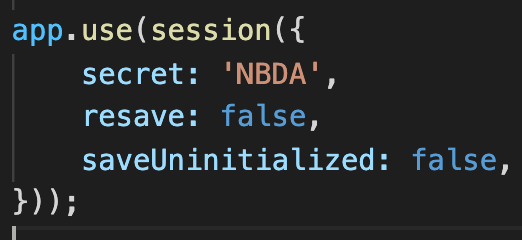
* wrong email with any password
* Correct email but wrong password
* Correct email and correct password

If everything works, you will be redirected to the login page when authentication fails, and you will be redirected to the homepage when the authentication succeeds. When everything works, demo your app to the TA or the instructor to get credit for this part.

**Part 4. Add session**

To be able to track the state of each connection, you will add sessions to the application. The session is created when a user logs in, and is destroyed when the user logs off. Further, the session is also used to authorize access to different pages of the app.

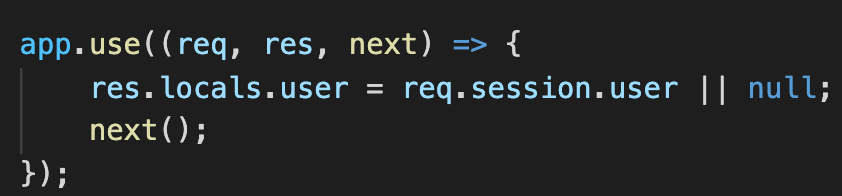
1. In the terminal, install the express-session module.
2. In app.js, require express-session. Then, type the following code to use session as a middleware. Feel free to change the secret to anything you like.



1. Open userRoutes.js, in the route handler for login, add the following line to initialize the session after a user is authenticated:

req.session.user = { id: user.\_id, name: user.firstName };

1. Now, the user information is stored in the session, and they can be accessed easily by the controllers. However, the view templates do not have direct access to the session variable. An easy way to get around this is to store this information in res.locals, which is an object that contains response local variables scoped to the request, and therefore available only to the view(s) rendered during that request / response cycle. Add the following middleware to app.js (make sure to place it after session middleware and before all of the routing middlewares).



Whenever there is a request, this middleware stores the session’s user variable to the res.locals object. Now, the view templates will be able to access this variable directly. For example, to get the id of the current user, simply call user.id in the view template.

1. In this step, you will create a profile page for the user.

* In nav.js, add a link: <a href='/users/profile’>My Profile</a> |
* Open create.ejs in the views/users directory, and save it as profile.ejs. The page only displays a <h3> element: Welcome, user’s first name. (Hint: this is stored in the res.locals.user variable, which the template has access to as we set up a middleware for this in step 4)
* In userRoutes.js, add a route handler to handle the GET request to this endpoint. This handler will render profile.ejs in the views/users directory.

1. Run your app, login and then click my profile to make sure this works.
2. Next, you will implement the functionality for logout.

* In nav.js, add a link for logout: <a href='/users/logout'>Log out</a> |
* In userRoutes.js, add a route handler to handle the GET request to this endpoint. The handler destroys the session and redirects the user to the homepage.

1. Run your app, Test your application to make sure that users can login, view the profile page, and logout.
2. As of now, both logged in users and guests can view and edit all of the resources in this web app. For example, a guest can view, create, edit and delete restaurants. How to limit access to these resources to authenticated users only? You will use session authentication for this.
3. Begin by updating the nav.js file so that

* A logged in user can see the following choices:
  + Home
  + Restaurants
  + Create New Restaurants
  + My Profile
  + Logout
* A guest can see the following choices:
  + Home
  + Sign Up
  + Login

Hint: in nav.ejs, use an if condition to check if the user variable is null, if so, then the user is a guest.

1. Run your app, when a guest is viewing the pages, only three choices are available. Login, then you should be able to see five choices listed above.

Now, you limited guests from accessing resources by rendering dynamic links in the nav. However, this does not completely limit guests from accessing the resources. This is because users can interact (send requests and receive responses) from any application on the world wide web without necessarily using the web pages that are tied to the application. A user can simply send a request to a web application using command-line or softwares like postman. Thus, in addition to the dynamic navigation bar, authentication is also implemented in the controllers.

1. In the controllers directory, create a new file named authController.js.
2. In authController.js, define and export a function named isLoggedIn. The function checks if req.session.user is null, if it is, redirect to the login page; otherwise, call next().



1. In authController.js, define and export a function named isLoggedOut. The function checks if req.session.user is null, if it is, call next(); otherwise, redirect to the profile page.
2. To prevent a guest from viewing/updating/deleting a restaurant, you will use the functions defined in the authController.js. In restaurantRoutes.js, import the isLoggedIn function. Use this function on all requests to the endpoint of ‘/’.

router.use('/', isLoggedIn);

Make sure to place this line before all of the route handlers. Since this function only passes the requests to the next route handler if a user is logged in, guests will not be able to reach any other route handlers in this file.

1. Similarly, to prevent a guest from viewing the profile page or logging out, import the isLoggedIn function in userRoutes.js. Add this function as the second parameter in the route handler for profile and logout. For example,

router.get('/profile', isLoggedIn, (req, res, next)=>{...});

When there is a GET request to the endpoint ‘/users/profile’, it is being processed by the isLoggedIn middleware, if the user is not logged in, the request is being redirected to the login page; otherwise, the request is being processed by the (req, res, next)=>{...} of this route handler.

1. Without logging in, try the following two urls in the web browser:

* localhost:8084/users/logout
* localhost:8084/users/profile

If everything works, all of these requests should redirect the user to the login page.

1. To prevent logged in users from signing up and logging in, import the isLoggedOut function in userRoutes.js. Add this function to the proper route handlers.
2. After a user logs in, try the following two url in the web browser:

* localhost:8084/users/login
* localhost:8084/users/create

If everything works, both of these requests should redirect the user to the profile page.

1. Finally, you will separate routes and controllers for users. In the controllers directory, create a new file named userController.js. In this file, copy paste the controller functions from userRoutes.js. In userRoutes.js, require userController. Replace each callback function with the corresponding function that you defined in the controller module.
2. Run your app to make sure that everything still works. After you complete this step, show it to the instructor or the TA to get credit for this part.

You are done. Zip your project to a zip file and upload it to Canvas.