Lab 12: *User Authentication* Backend

USER LOGIN

PASSPORT, BCRYPT, EJS

AUTHENTICATION

IMPORTANT NOTE: Install Node JS for Backend

Table of Content: *Implementing a Simple (HTTP) Web Server*

Lab Introduction

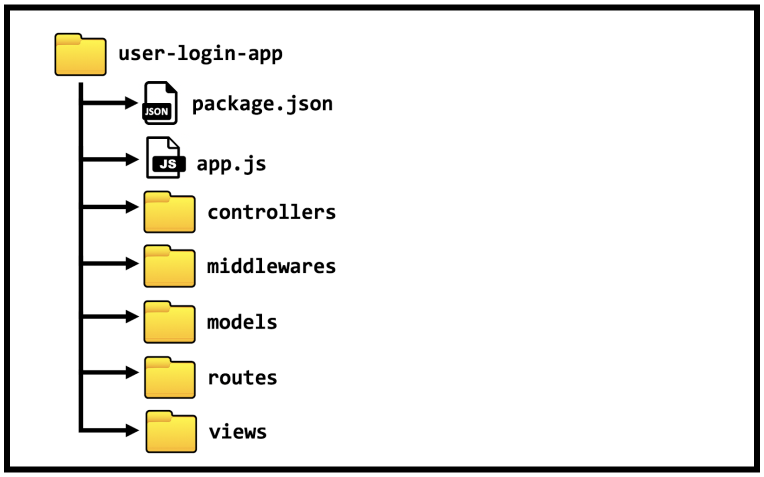
Prerequisites
None. This is the first lab in the User Authentication sequence.

Motivation
Use Passport & Bcrypt in an Express App

Goal
Build a User Login application, using server-side HTML rendering

Learning Objectives
- Passport
- Local Strategy
- Protected routes
- Bcrypt
- Express Sessions
- Express Flash
- EJS
- Middleware
- dotenv

Server-side Architecture:
Start this project by making a project folder where all your assets & scripts will be organized.
Create all necessary files and folders as illustrated below.



Concepts

User Authentication

User Authentication is the process by which you verify a user's credentials for access to non-public, restricted data.

- **Passport:** Node module for user authentication, supports Sessions, JWT, OAuth
- **Local Strategy:** Strategy for user authentication whereby the express app manages all of the username and password data. Alternate strategies may outsource this to: google, facebook, github, twitter. Yout app may allow multiple strategies, giving the user options on how to login.
- **Protected Routes:** A route or endpoint that a request cannot access if they do not have the proper authentication. Example may include not accessing a user-profile page unless the Request comes from that particular user.

Sessions

Sessions allows a server to identify a client between requests using a browser cookie. The client's session ID is given to the browser in the response.

- **Express-Session:** Manages session cookies and appending session id to Request object.
- **Passport-Session:** Extends Express-Session to include user authentication status in cookie

---

Bcrypt

Bcrypt is a Node module used to securely hash passwords and compare login in attempts to hashed passwords. You should never store user passwords without encrypting them.

EJS

Embedded JavaScript, is a templating engine for rendering server-side HTML containing JS data.

Express Flash

Express modules that allows messages to be transmitted across the server app using the Request Object

---

dotenv

Node module that setups a process.ENV that contains hidden or secret data that should not be exposed in github repos.

Iteration 0: Design Game Client of REST API

'Approach' → Plan phase

Goal #0: Specify the views, controls, and logic of client app

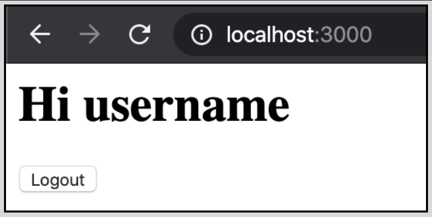
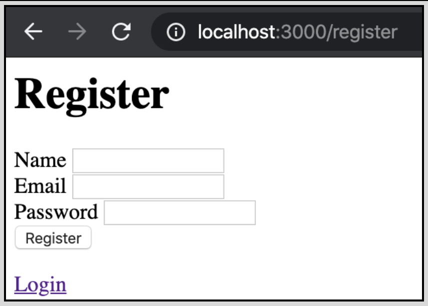
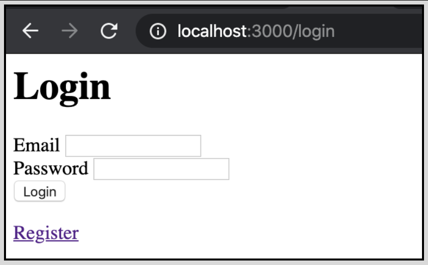
Approach: **Pencil & paper**

Before coding a server app, first plan out routes, views, controls, and how HTTP Request will be resolved.

'Apply' → Do phase

Design Steps

Step 1: Create Mockups of your views.

1	 localhost:3000 Hi username Logout	**Home (Protected Route: User Request)** *Private profile page for User* ● Option 1: Logout *(goto 3)*
2	 localhost:3000/register Register Name Email Password Register Login	**Register (Protected Route: Public Request)** *Inputs: name, email, password* *Registration form to make a new account* ● Option 1: submit *(goto 3)*
3	 localhost:3000/login Login Email Password Login Register	**Login (Protected Route: Public Request)** *Inputs: email, password* *Login form to access a user profile* ● Option 1: submit *(goto 1) or (goto 3) on error*

'Assess' → Test phase

Once you understand the mockups, then start incrementally building the project!

Iteration 1: Server Setup

'Approach' → Plan phase

Goal #1: Setup a simple server application

Approach: Configuration file, Express app

Start the project by setting up the package.json with dependencies for building a simple server app.

'Apply' → Do phase

Steps

- **Step 1**: *package.json* → Project configuration file. Define a start command. Dependency: express
- **Step 2**: *app.js* → Express app that listens on port

Step 1 (JSON): ***package.json*** → Create configuration file

`package.json` contains the metadata for managing, building, & launching your Node app.

package.json

```
{
   "name": "user-login-app",
   "version": "1.0.0",
   "description": "This project implements user authentication using passport",
   "main": "app.js",
   "scripts":{
      "start": "npm install && node app"
   },
   "author": "me",
   "dependencies": {
      "express": "*"
   }
}
```

Step 2 (JS): ***app.js*** → Create app

Define a barebones express app that listens on a port

app.js

```
const express = require('express');
const app = express();
const port = 3000;

app.listen(port, () => console.log(`listening on... ${port}`) );
```

'Assess' → Test phase

Launch app from terminal

```
npm start
```

Terminal should show app is listening

```
listening on... 3000
```

Iteration 2: EJS to render server-side HTML

'Approach' → Plan phase

Goal #2: Use EJS to render HTML data in the HTTP Response

Approach: EJS

Embedded JavaScript (EJS) is a template engine that renders HTML

'Apply' → Do phase

JavaScript Steps

- **Step 1**: *package.json* → Add dependency: 'ejs'
- **Step 2**: *index.ejs* → define HTML elements
- **Step 3**: *user-routes.js* → assigns controllers to the app's endpoint routes (http methods)
- **Step 4**: *user-controllers.js* → functions that resolve client requests into server responses
- **Step 5**: *app.js* → import user routes and a function that sets up logic & middleware for app

Step 1 (JSON): ***package.json*** → Refactor: dependencies

Add 'ejs' into the app dependencies

package.json

```
{
  "name": "user-login-app",
  "version": "1.0.0",
  "description": "This project implements user authentication using passport",
  "main": "app.js",
  "scripts": {
     "start": "npm install && node app"
  },
  "author": "me",
  "dependencies": {
     "express": "*",
     "ejs": "*"
  }
}
```

Step 2 (EJS): ***index.ejs*** → Create: views/index.ejs

Define the HTML elements for index (home) page

views/index.ejs

```
<h1>Hi<h1>
```

Step 3 (JS): ***user-routes.js*** → Create: routes/user-routes

Create user-routes that manages the app's endpoints (http methods) and attaches controllers (functions)

routes/user-routes.js

```
var express = require('express');
var router = express.Router();
const controller = require('../controllers/user-controllers');

router.get('/', controller.getIndex );

module.exports = router;
```

Step 4 (JS): **user-*controllers.js*** → Create: controllers/user-controllers

Create user-controllers that manages the functions from client requests to resolve into a server response

controllers/user-controllers.js

```
class UserControllers{
   getIndex(request, response){
       response.render( 'index.ejs' );
   }
}

module.exports = new UserControllers();
```

Step 5 (JS): ***app.js*** → Refactor: app

Refactor app to import the user-routes and define a function to set up the logic & middleware for the app.

app.js

```
const express = require('express');
const app = express();
const port = 3000;
const userRoutes = require('./routes/user-routes');

function setupApp(){
   app.use('/', userRoutes);
}

setupApp();
app.listen(port, () => console.log(`listening on... ${port}`) );
```

'Assess' → Test phase

Launch app from terminal

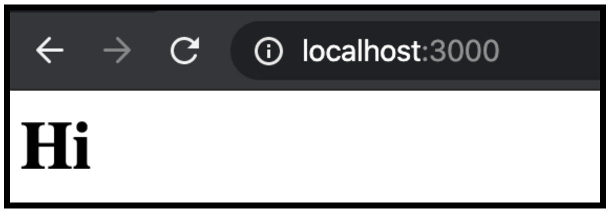
```
npm start
```

Terminal should show app is listening

```
listening on... 3000
```

Test (Browser):

Open browser to: http://localhost:3000



Hi

Iteration 3: EJS to embed JavaScript into HTML

'Approach' → Plan phase

Goal #3: Use EJS to implement JavaScript data into the HTML

Approach: EJS

Embedded JavaScript (EJS) can embed app data into the HTML

'Apply' → Do phase

JavaScript Steps

- **Step 1**: app.js → setup app to use 'ejs' as its view engine
- **Step 2**: user-controllers.js → add second parameter to render method passing data to HTML
- **Step 3**: index.ejs → dereference the JavaScript data using ejs syntax

Step 1 (JS): ***app.js*** → Refactor: setupApp()

Setup app to use 'ejs' as its view engine

app.js → setupApp

```
function setupApp(){
   app.set('view-engine', 'ejs');
   app.use('/', userRoutes);
}
```

Step 2 (JS): ***user-controllers.js*** → Refactor: getIndex()

Add a second parameter to the render method passing data to HTML

controllers/user-controllers.js

```
getIndex(request, response){
   response.render( 'index.ejs', {name:'data'} );
}
```

Step 3 (EJS): ***index.ejs*** → Refactor

Dereference the JavaScript data using ejs syntax

views/index.ejs

```
<h1>Hi <%= name %> </h1>
```

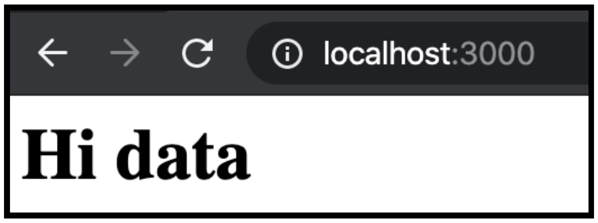
'Assess' → Test phase

Launch app from terminal

```
npm start
```

Test (Browser):

Open browser: http://localhost:3000



Hi data

Iteration 4: Setup Login & Register pages

'Approach' → Plan phase

Goal #4: Setup a controller that launches the 'Join Game' view from 'Main Menu' view

Approach: EJS

Use EJS to create two new HTML pages: Login page, Register page

---

'Apply' → Do phase

JavaScript Steps

- **Step 1**: login.ejs → create a login.ejs file
- **Step 2**: register.js → create a register.ejs file
- **Step 3**: user-controllers.js → add a getLogin function to render the login page
- **Step 4**: user-controllers.js → add a getRegister function to render the register page
- **Step 5**: user-routes.js → Routes for GET requests on paths: (login & register), assign controllers

Step 1 (EJS): ***login.ejs*** → Create: views/login.ejs

In views, create a login.ejs file and add some text

views/login.ejs

```
Login
```

Step 2 (EJS): ***login.ejs*** → Create: views/register.ejs

In views, create a register.ejs file and add some text

views/register.ejs

```
Register
```

Step 3 (JS): ***user-controllers.js*** → Refactor: UserControllers (class)

Refactor UserControllers class to add a getLogin function that renders the login page

controllers/user-controllers.js → UserControllers

```
getLogin(request, response){
   response.render( 'login.ejs' );
}
```

Step 4 (JS): ***user-controllers.js*** → Refactor: UserControllers (class)

Refactor UserControllers class to add a getRegister function that renders the register page

controllers/user-controllers.js → UserControllers

```
getRegister(request, response){
   response.render( 'register.ejs' );
}
```

Step 5 (JS): ***user-routes.js*** → Refactor

Define routes for the GET requests for login and register paths, and assign controllers

routes/user-routes.js

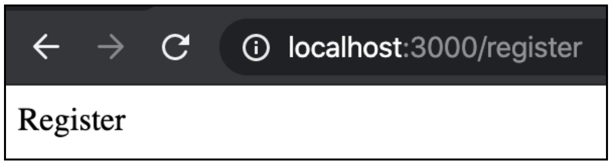
```
var express = require('express');
var router = express.Router();
const controller = require('../controllers/user-controllers');

router.get('/', userController.getIndex );
router.get('/login', controller.getLogin);
router.get('/register', controller.getRegister);

module.exports = router;
```

'Assess' → Test phase

Launch app from terminal

```
npm start
```

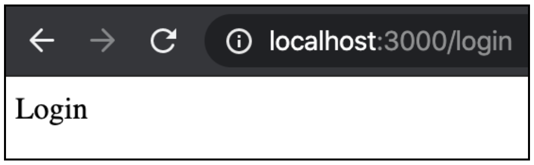
Open browser pages to the two routes



localhost:3000/register

Register

Go to: http://localhost:3000/register



localhost:3000/login

Login

Go to: http://localhost:3000/login

Iteration 5: Register & Login Forms

'Approach' → Plan phase

Goal #5: HTML forms for Register and Login pages

Approach: EJS, HTML forms

Use EJS to define HTML forms for Register and Login pages

'Apply' → Do phase

JavaScript Steps

- **Step 1**: register.ejs → HTML form for register page
- **Step 2:** login.ejs → HTML form for login page

Step 1 (EJS): ***register.ejs*** → Refactor HTML

HTML form for register page

views/register.ejs

```
<h1>Register</h1>
<form action='/register' method='POST'>
  <div>
      <label for='name'>Name</label>
      <input type='text' id='name' name='name' required>
  </div>
  <div>
      <label for='email'>Email</label>
      <input type='email' id='email' name='email' required>
  </div>
  <div>
      <label for='password'>Password</label>
      <input type='password' id='password' name='password' required>
  </div>
  <button type='submit'>Register</button>
</form>
<a href='/login'>Login</a>
```

Step 2 (EJS): ***login.ejs*** → Refactor HTML

HTML form for login page

views/login.ejs

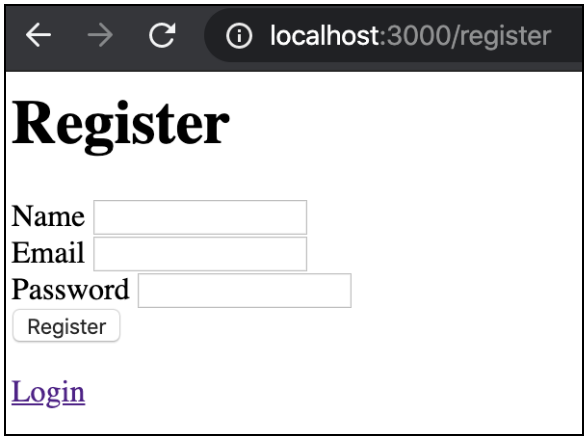
```
<h1>Login</h1>
<form action='/login' method='POST'>
  <div>
      <label for='email'>Email</label>
      <input type='email' id='email' name='email' required>
  </div>
  <div>
      <label for='password'>Password</label>
      <input type='password' id='password' name='password' required>
  </div>
  <button type='submit'>Login</button>
</form>
<a href='/register'>Register</a>
```

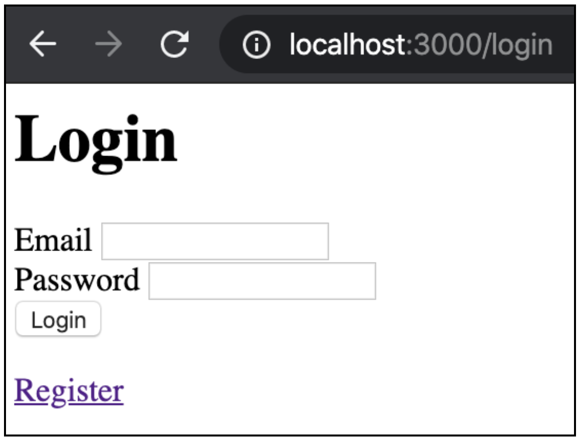
'Assess' → Test phase

Launch app from terminal

```
npm start
```

Open browser pages to the two routes





Go to: http://localhost:3000/register

Go to: http://localhost:3000/login

Iteration 6: Post Form to Server

'Approach' → Plan phase

Goal #6: Setup server to listen for POST request of form data

Approach: **Body-Parser**

Setup body-parser to access form data

'Apply' → Do phase

JavaScript Steps

- **Step 1**: app.js → import body-parser module to access data in the request's body
- **Step 2**: app.js → set app to use the body-parser module
- **Step 3**: user-controllers.js → destructure data from POST request body & display in terminal
- **Step 4**: user-controllers.js → destructure data from POST request body & display in terminal
- **Step 5**: user-routes.js → Routes for POST request on paths: (login & register), assign controllers

Step 1 (JS): ***app.js*** → Refactor: requires body-parser

Import body-parser (middleware) to access data in the request's body

app.js

```
const bodyParser = require('body-parser')
```

Step 2 (JS): ***app.js*** → Refactor: setupApp()

Refactor setupApp function so the express app initializes and uses the bodyParser module

app.js → setupApp

```
function setupApp(){
   app.set('view-engine', 'ejs');
   app.use( bodyParser.urlencoded( {extended:false} ) );
   app.use('/', userRoutes);
}
```

Step 3 (JS): ***user-controllers*** → Refactor: UserControllers (class)

Define postLogin method in UserControllers class. Destructure data from the request body & show to console. Then send a response of 'success' to the client.

controllers/user-controllers.js → UserControllers

```
postLogin(request, response){
   const {email, password} = request.body;
   console.log(email, password);
   response.send('success');
}
```

Step 4 (JS): ***user-controllers*** → Refactor: UserControllers (class)

Define postRegsiter method in UserControllers class. Destructure data from the request body & show to console. Then send a response of 'success' to the client

controllers/user-controllers.js → UserControllers

```
postRegister(request, response){
   const {name, email, password} = request.body;
   console.log(name, email, password);
   response.send('success');
}
```

Step 5 (JS): ***user-routes*** → Refactor

Define routes for POST requests on paths: (login & register), and assign controllers

routes/user-routes.js

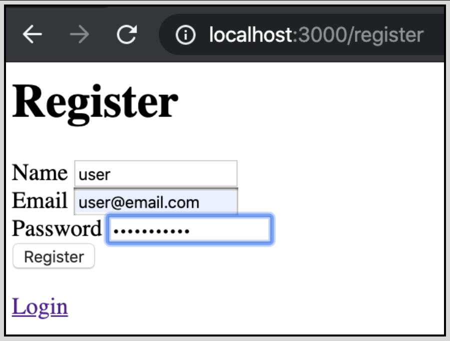
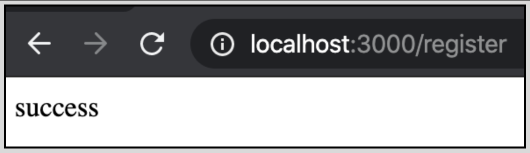
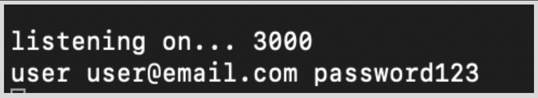
```
var express = require('express');
var router = express.Router();
const controller = require('../controllers/user-controllers');

router.get('/', controller.getIndex );
router.get('/login', controller.getLogin);
router.get('/register', controller.getRegister);
router.post('/login', controller.postLogin);
router.post('/register', controller.postRegister);

module.exports = router;
```

'Assess' → Test phase

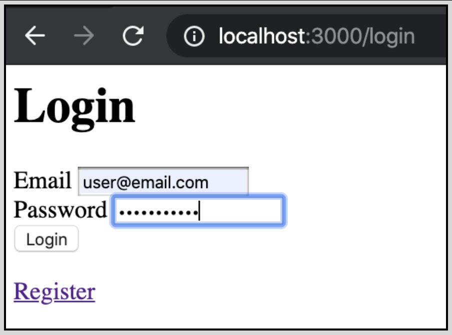
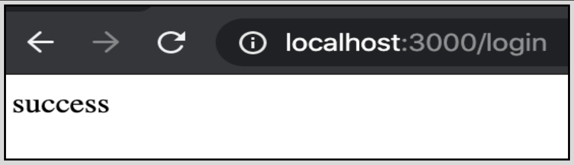
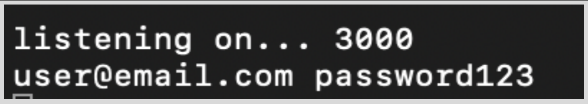
Launch app from terminal

```
npm start
```

Test (Register)

1	Go to https://localhost:3000/register Fill in the form and submit	 localhost:3000/register **Register** Name: user Email: user@email.com Password: •••••••••••
2	**Browser** should display success	 localhost:3000/register success
3	**Server** should display form data	 `listening on... 3000` `user user@email.com password123`

Test (Login)

1	Go to https://localhost:3000/login Fill in the form and submit	 localhost:3000/login **Login** Email: user@email.com Password: •••••••••••
2	**Browser** should display success	 localhost:3000/login success
3	**Server** should display form data	 `listening on... 3000` `user@email.com password123`

Iteration 7: Data Model for Users

'Approach' → Plan phase

Goal #7: Define a data model for the users collection

Approach: **Class: Users**

Define a javascript class to manage the users data

'Apply' → Do phase

JavaScript Steps

- **Step 1**: package.json → add dependency: 'shortid'
- **Step 2:** users-model.js → class that models users collection with method to add users
- **Step 3**: user-controllers.js → import the users model into the user-controllers module
- **Step 4**: user-controllers.js → postRegister method should add user data to the model

Step 1 (JSON): ***package.json*** → Refactor: dependencies

Add 'shortid' to project dependencies

package.json

```
{
  "name": "user-login-app",
  "version": "1.0.0",
  "description": "This project implements user authentication using passport",
  "main": "app.js",
  "scripts": {
     "start": "npm install && node app"
  },
  "author": "me",
  "dependencies": {
     "express": "*",
     "ejs": "*",
     "shortid": "*",
  }
}
```

Step 2 (JS): ***users-model.js*** → Create: Users (class)

Define a class to model the users collection that has a constructor and add methods.

models/users-model.js

```
const shortid = require('shortid');

class Users{
   constructor(){
       this.users = [];
   }
   add(name, email, password){
       const id = shortid.generate();
       const user = {id:id, name:name, email:email, password:password};
       this.users.push(user);
       console.log(this.users)
   }
}
module.exports = new Users();
```

Step 3 (JS): ***user-controllers.js*** → Refactor: require users-model

import the users model into the users-controllers module

controllers/user-controllers.js

```
const users = require('../models/users-model');
```

Step 4 (JS): ***user-controllers.js*** → Refactor: UserControllers (class)

Refactor postRegister method to add a user to model & redirect to Login page on success. On error, it should redirect to the Register page.

controllers/user-controllers.js

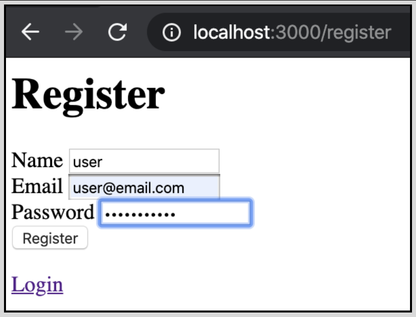
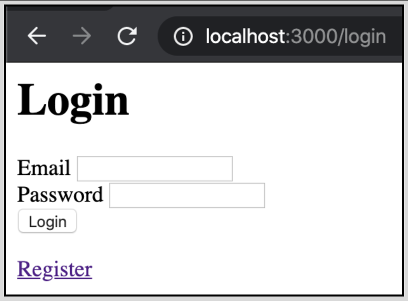
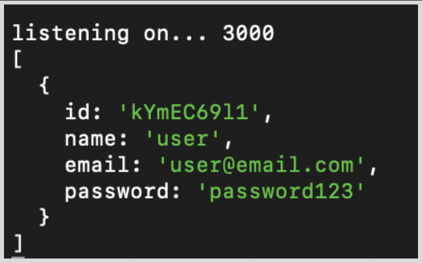
```
postRegister(request, response){
   try{
      const {name, email, password} = request.body;
      users.add(name, email, password);
      response.redirect('/login');
   }
   catch{
      response.redirect('/register');
   }
}
```

'Assess' → Test phase

Launch app from terminal

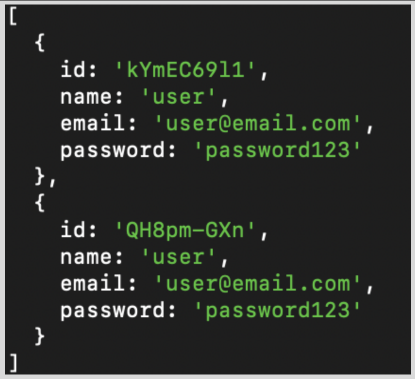
```
npm start
```

Test (Register)

1	Go to https://localhost:3000/register Fill in the form and submit	 localhost:3000/register **Register** Name user Email user@email.com Password •••••••••••• Register Login
2	**Browser** redirects to Login page	 localhost:3000/login **Login** Email Password Login Register
3	**Server** should display user data	

```
listening on... 3000
[
  {
    id: 'kYmEC69l1',
    name: 'user',
    email: 'user@email.com',
    password: 'password123'
  }
]
```

Problem

You may register the same user data multiple times into the users collection. The app should prevent the user from registering a pre-existing email address. Note: The fix to this is trivial, but it's not covered in this lab.	

```
[
  {
    id: 'kYmEC69l1',
    name: 'user',
    email: 'user@email.com',
    password: 'password123'
  },
  {
    id: 'QH8pm-GXn',
    name: 'user',
    email: 'user@email.com',
    password: 'password123'
  }
]
```

Iteration 8: Sessions with Express

'Approach' → Plan phase

Goal #8: Use Express to establish sessions with clients

Approach: **Express-Session**

Express-Session is middleware that generates session IDs into the request/response objects. Browser saves this session ID as a cookie and appends it to request head on each new request.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: package.json → add dependency: 'express-session'
- **Step 2**: app.js → import the 'express-session' module into the app
- **Step 3**: app.js → initialize a session and have the app use it.
- **Step 4**: user-controllers.js → display the session id from the GET request object.

Step 1 (JSON): package.json → Refactor: dependencies

Add 'express-session' to project dependencies

package.json

```
{
  "name": "user-login-app",
  "version": "1.0.0",
  "description": "This project implements user authentication using passport",
  "main": "app.js",
  "scripts": {
     "start": "npm install && node app"
  },
  "author": "me",
  "dependencies": {
     "express": "*",
     "ejs": "*",
     "shortid": "*",
     "express-session": "*"
  }
}
```

Step 2 (JS): app.js → Refactor: requires

Import the 'express-session' module into the app

app.js

```
const session = require('express-session');
```

Step 3 (JS): app.js → Refactor: setupApp()

Refactor setupApp function to configure a session and use it. Sessions require a secret string for hashing. Set saveUninitialized to true, so that session IDs remain the same for requests from the same client.

app.js

```
function setupApp(){
   app.set('view-engine', 'ejs');
   app.use( bodyParser.urlencoded({ extended:false }) );
   const sessionConfig = { secret:'secret-word', resave:false, saveUninitialized:true };
   app.use( session(sessionConfig) );
   app.use('/', userRoutes);
}
```

Step 4 (JS): user-controllers.js → Refactor: getIndex, getLogin, getRegister

Refactor the getIndex, getLogin, getRegister methods to display the session id from the request object.

controllers/user-controllers.js → UserControllers

```
getIndex(request, response){
   console.log(request.session.id);
   response.render( 'index.ejs', {name: 'data'} );
}
```

```
getLogin(request, response){
   console.log(request.session.id);
   response.render( 'login.ejs' );
}
```

```
getRegister(request, response){
   console.log(request.session.id);
   response.render( 'register.ejs' );
}
```

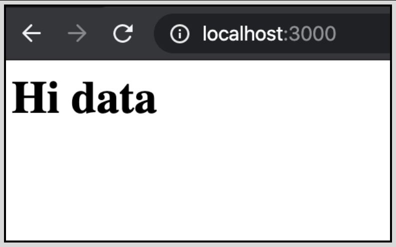
'Assess' → Test phase

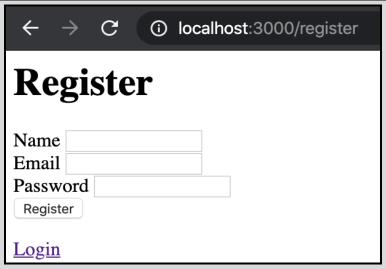
Launch app from terminal

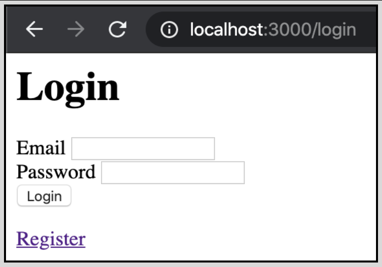
```
npm start
```

Open browser:

(Test: Session IDs)

1	Go to http://localhost:3000/
	
	Session ID displays
	Note: it shouldn't change between requests
		 localhost:3000
		# Hi data
			```
			listening on... 3000
			KVCSoR5ww50taaIxd_ZLxEcDlKQCRz34
			```

2	Go to http://localhost:3000/register
	
	Session ID displays
	Note: it shouldn't change between requests
		 localhost:3000/register
		Register
		Name
		Email
		Password
		Register
		Login
			```
			listening on... 3000
			KVCSoR5ww50taaIxd_ZLxEcDlKQCRz34
			KVCSoR5ww50taaIxd_ZLxEcDlKQCRz34
			```

3	Go to https://localhost:3000/login
	
	Session ID displays
	Note: it shouldn't change between requests
		 localhost:3000/login
		Login
		Email
		Password
		Login
		Register
			```
			listening on... 3000
			KVCSoR5ww50taaIxd_ZLxEcDlKQCRz34
			KVCSoR5ww50taaIxd_ZLxEcDlKQCRz34
			KVCSoR5ww50taaIxd_ZLxEcDlKQCRz34
			```

Note: Try using incognito mode in the browser to get a different session ID.

How does it work?

Every time the server receives a request, express stores it in memory using session middleware to manage session data and appends the ID to the request/response objects. The browser gets the session id from the response header. If there is a session id stored in the browser for a given url, it sends it in its requests.

Iteration 9: Authenticate User with Passport

'Approach' → Plan phase

Goal #9: Use Passport to Authenticate Users with usernames & passwords

Approach: **Passport, Passport-Local**

Use Passport module to handle user authentication. Use Passport-Local to manage logic for usernames & password in this app, and not use other services such as Google, Facebook, etc.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: package.json → add dependencies for: 'passport', 'passport-local'
- **Step 2**: users-model.js → add findUser method that finds a user given a key and a value
- **Step 3**: passport-config.js → module to configures passport with a local strategy
- **Step 4**: app.js → import the 'passport-config' module into the app
- **Step 5**: app.js → app to initialize passport and use passport with sessions
- **Step 6**: user-controls.js → import the 'passport-config' module into user-controls
- **Step 7**: user-controls.js → postLogin invoke passport's authenticateUser function as middleware

Step 1 (JSON): package.json → Refactor: dependencies

Add 'passport' and 'passport-local' to project dependencies

package.json

```
{
  "name": "user-login-app",
  "version": "1.0.0",
  "description": "This project implements user authentication using passport",
  "main": "app.js",
  "scripts": {
     "start": "npm install && node app"
  },
  "author": "me",
  "dependencies": {
     "express": "*",
     "ejs": "*",
     "shortid": "*",
     "express-session": "*",
     "passport": "*",
     "passport-local": "*"
  }
}
```

Step 2 (JS): users-model.js → Refactor: Users (class)

Add findUser method that takes parameters key, value. The function returns the item in the users collection that matches the value, if there is no match, it returns undefined.

models/users-model.js → Users

```
findUser(key, value){
   const user = this.users.find( item => item[key] === value)
   return user;
}
```

Step 3 (JS): passport-config.js → Create file

Create a module that configures passport with a local strategy for authenticating a user. Passport needs a function that handles authetnicateUser and needs the names for username & password from our forms.

middlewares/passport-config.js

```
const { Strategy } = require('passport-local');
const passport = require('passport');
const users = require('../models/users-model');

function authenticateUser(email, password, done){
   const user = users.findUser('email', email);
   if (user === undefined){
       console.log("No user with that email");
       return done();
   }
   if (password === user.password){
       console.log("User Authenticated");
       return done();
   }
   else{
       console.log("Password incorrect");
       return done();
   }
}

function setupPassport(){
   const formNames = { usernameField:'email', passwordField:'password' };
   const localStrategy = new Strategy( formNames, authenticateUser);
   passport.use(localStrategy)
}

setupPassport();
module.exports = passport;
```

Step 4 (JS): app.js → Refactor: requires

Import the 'passport-config' module into the app

app.js

```
const passport = require('./middlewares/passport-config');
```

Step 5 (JS): app.js → Refactor: setupApp()

Refactor setupApp function to have the app use an initialized instance of passport & passport session to append authentications into the session header of the request/response.

app.js

```
function setupApp(){
   app.set('view-engine', 'ejs');
   app.use( bodyParser.urlencoded({ extended:false }) );
   const sessionConfig = { secret:'secret-word', resave:false, saveUninitialized:true };
   app.use( session(sessionConfig) );
   app.use( passport.initialize() );
   app.use( passport.session() );
   app.use('/', userRoutes);
}
```

Step 6 (JS): user-controls.js → Refactor: requires

Import the 'passport-config' module into user-controls

app.js

```
const passport = require('../middlewares/passport-config');
```

Step 7 (JS): user-controls.js → Refactor: postLogin()

Refactor postLogin to invoke passport's authenticateUser function as middleware to manage authenticating logins.

controllers/user-controllers → UserControllers

```
postLogin(request, response, next){
        const config = {};
        config.successRedirect = '/';
        config.failureRedirect = '/login';
        const authHandler = passport.authenticate('local', config);
        authHandler(request, response, next);
}
```

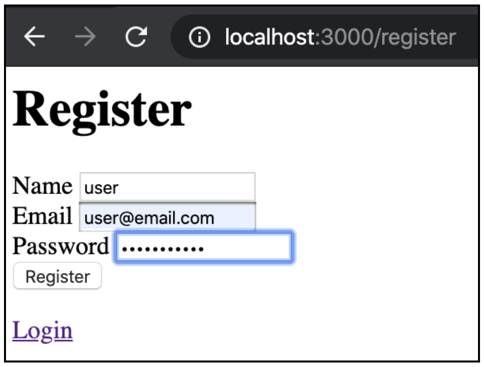
'Assess' → Test phase

Launch app from terminal

```
npm start
```

Setup (Register a User)

Open browser: http://localhost:3000/register & **register a user**, then test the 3 cases for logins

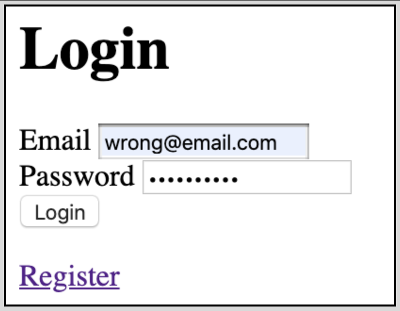
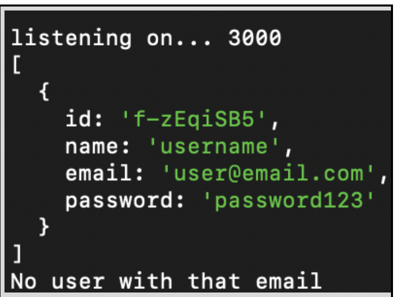
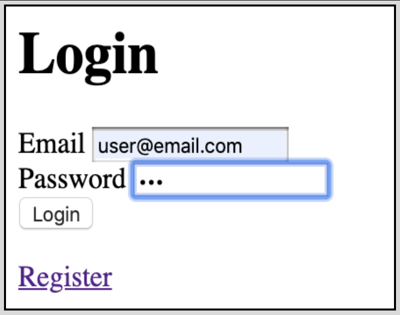
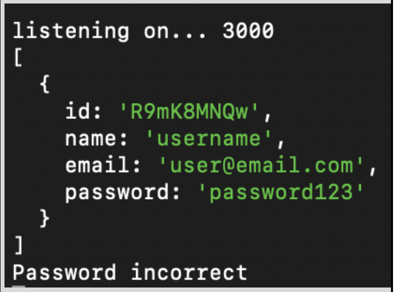
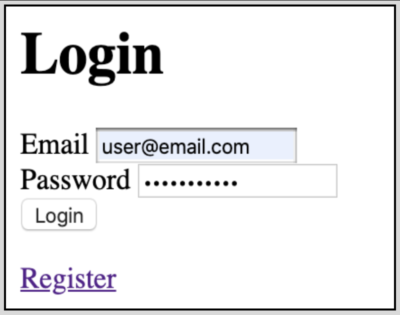
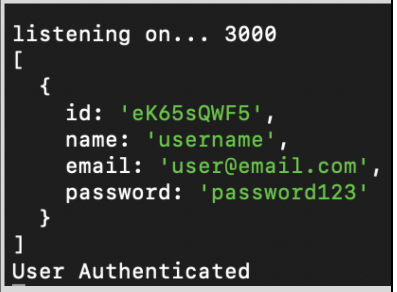


Register

Name user
Email user@email.com
Password •••••••••••
Register

Login

Test: (Login User) → 3 cases

1	**Case: Failure (Email)** Go to https://localhost:3000/login ***Submit wrong email.***	 Login Email wrong@email.com Password •••••••••• Login Register	 ```listening on... 3000 [{ id: 'f-zEqiSB5', name: 'username', email: 'user@email.com', password: 'password123' }] No user with that email```
2	**Case : Failure (Password)** Go to https://localhost:3000/login ***Submit wrong password.***	 Login Email user@email.com Password ••• Login Register	 ```listening on... 3000 [{ id: 'R9mK8MNQw', name: 'username', email: 'user@email.com', password: 'password123' }] Password incorrect```
3	**Case: Success** Go to https://localhost:3000/login ***Submit correct email and password.***	 Login Email user@email.com Password ••••••••••• Login Register	 ```listening on... 3000 [{ id: 'eK65sQWF5', name: 'username', email: 'user@email.com', password: 'password123' }] User Authenticated```

Iteration 10: Serialize/Deserialize Users with Passport

'Approach' → Plan phase

Goal #10: Serialize/Deserialize Users with Passport

Approach: **Passport**

To restore a user for each requests during a session, passport must implement a method that defines how to serialize the user data and another method on how to deserialize a user's data

---

'Apply' → Do phase

JavaScript Steps

- **Step 1**: passport-config.js → authenticateUser pass user data so its accessible for serialization
- **Step 2**: passport-config.js → setupPasspot defines serializeUser and deserializeUsers functions
- **Step 3**: user-controllers.js → getIndex passes user name to index.ejs to render in HTML

Step 1 (JS): ***passport-config.js*** → Refactor: authenticateUser()

Refactor authenticateUser to pass the done method two parameters, the first parameter is for an error and should be null, the second parameter is for data, so pass user data if it exists and false if it doesn't. This data is set into the Request object.

middlewares/passport-config.js

```
function authenticateUser(email, password, done){
   const user = users.findUser('email', email);
   if (user === undefined){
       console.log("No user with that email")
       return done(null, false);                              //err, user
   }
   if (password === user.password){
       console.log("User Authenticated")
       return done(null, user);                               //err, user
   }
   else{
       console.log("Password incorrect")
       return done(null, false);                              //err, user
   }
}
```

Step 2 (JS): ***passport-config.js*** → Refactor: setupPassport()

Refactor setupPassport to define passport's serializeUser and deserializeUser methods. Serialize the user with the id. To deserialize the user, invoke the findUser method from users using the id.

middlewares/passport-config.js

```
function setupPassport(){
   const formNames = { usernameField:'email', passwordField:'password' };
   const localStrategy = new Strategy( formNames, authenticateUser);
   passport.use(localStrategy);
   passport.serializeUser( (user,done) => done(null, user.id) );
   passport.deserializeUser( (id,done) => done(null, users.findUser('id',id))  );
}
```

Step 3 (JS): ***user-controllers.js*** → Refactor: getIndex()

Refactor getIndex to get the user's name from the request header and send to EJS to render as HTML

controllers/user-controllers → UserControllers

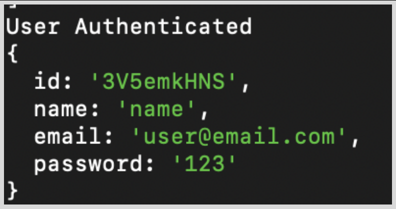
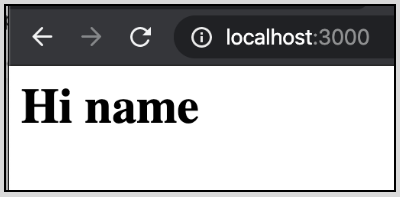
```
getIndex(request, response){
   response.render( 'index.ejs', {name: request.user.name} );
}
```

'Assess' → Test phase

Launch app from terminal

```
npm start
```

Test (Home page)

Register a user and log in. Redirects to home page, user name should display	 `User Authenticated { id: '3V5emkHNS', name: 'name', email: 'user@email.com', password: '123' }`	 localhost:3000 **Hi name**

Iteration 11: Messages with Express-Flash

'Approach' → Plan phase

Goal #11: Use the Request object to pass messages within the App

Approach: Express-Flash

Messages appended in the request. Messages displayed without sending back a response to the client.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: package.json → add dependencies for: 'express-flash'
- **Step 2**: app.js → import the 'express-flash' module into app
- **Step 3**: app.js → setupApp to have app use the flash middleware
- **Step 4**: passport-config.js → authenticateUser pass its fail messages into the Request object
- **Step 5**: user-controllers.js → postLogin configure passport use express-flash on failure messages
- **Step 6**: login.ejs → EJS syntax to check if an error message exists, and if so, render it in HTML

Step 1 (JSON): ***package.json*** → Refactor: dependencies

Add 'express-flash' to project dependencies

package.json

```
{
  "name": "user-login-app",
  "version": "1.0.0",
  "description": "This project implements user authentication using passport",
  "main": "app.js",
  "scripts": {
     "start": "npm install && node app"
  },
  "author": "me",
  "dependencies": {
     "express": "*",
     "ejs": "*",
     "shortid": "*",
     "express-session": "*",
     "passport": "*",
     "passport-local": "*",
     "express-flash": "*"
  }
}
```

Step 2 (JS): ***app.js*** → Refactor: requires

Import the 'express-flash' module into app

app.js

```
const flash = require('express-flash');
```

Step 3 (JS): ***app.js*** → Refactor: setupApp()

Refactor setupApp function to have app use the flash middleware

app.js

```
function setupApp(){
   app.set('view-engine', 'ejs');
   app.use( bodyParser.urlencoded({ extended:false }) );
   const sessionConfig = { secret:'secret-word', resave:false, saveUninitialized:false };
   app.use( session(sessionConfig) );
   app.use( flash() );
   app.use( configPassport.initialize() );
   app.use( configPassport.session() );
   app.use('/', userRoutes);
}
```

Step 4 (JS): ***passport-config.js*** → Refactor: authenticateUser()

Refactor authenticateUser to pass fail messages into the request object in the done parameters

middlewares/passport-config.js

```
function authenticateUser(email, password, done){
   const user = users.findUser('email', email);
   if (user === undefined){
      return done(null, false, {message:'No user with that email'});  //err, user, msg
   }
   if (password === user.password){
       return done(null, user);
   }
   else{
       return done(null, false, {message:'Password incorrect'});      //err, user, msg
   }
}
```

Step 5 (JS): ***user-controllers.js*** → Refactor: postLogin()

Refactor postLogin to have passport use express-flash on failure messages

controllers/user-controllers → UserControllers

```
postLogin(request, response, next){
        const config = {};
        config.successRedirect = '/';
        config.failureRedirect = '/login';
        config.failureFlash = true;
        const authHandler = passport.authenticate('local', config);
        authHandler(request, response, next);
}
```

Step 6 (EJS): ***login.ejs*** → Refactor: EJS syntax

Use EJS syntax to check if an error message exists, and if so, render it in HTML

views/login.ejs

```
<h1>Login</h1>

<% if (messages.error) { %>
 <%=  messages.error %>
<% } %>

<form action='/login' method='POST'>
  <div>
      <label for='email'>Email</label>
      <input type='email' id='email' name='email' required>
  </div>
  <div>
      <label for='password'>Password</label>
      <input type='password' id='password' name='password' required>
  </div>
  <button type='submit'>Login</button>
</form>
<a href='/register'>Register</a>
```

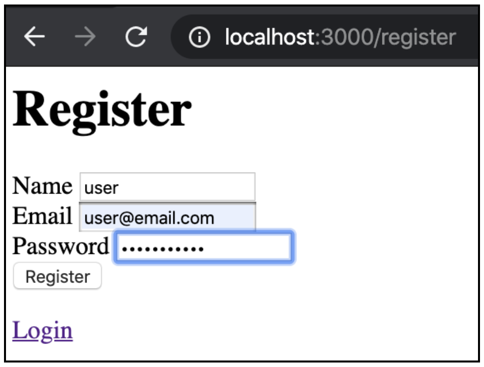
'Assess' → Test phase

Launch app from terminal

```
npm start
```

Setup (Register a User)

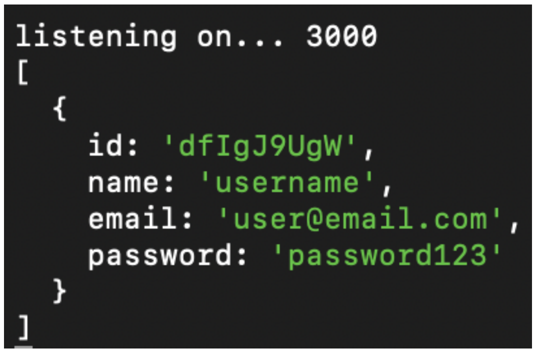
Open browser: http://localhost:3000/register & **register a user**, then test the 3 cases for logins


localhost:3000/register

Register

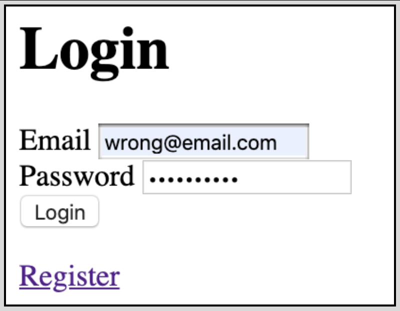
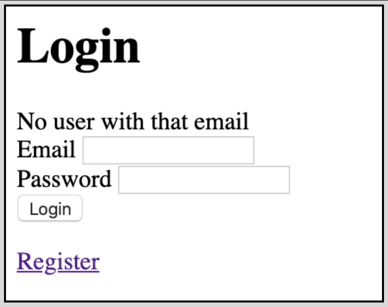
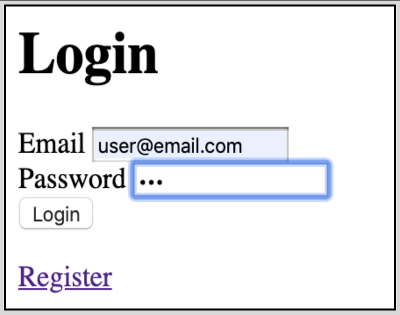
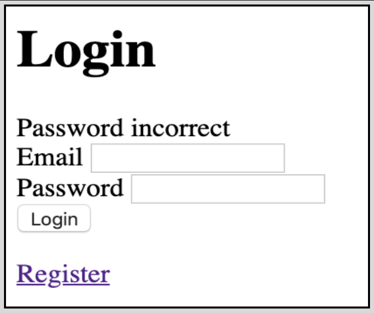
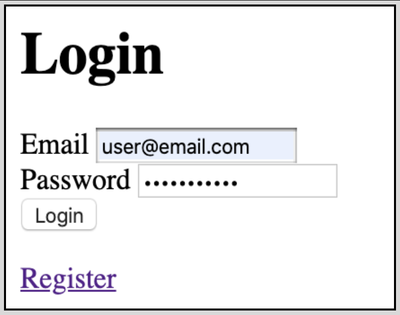
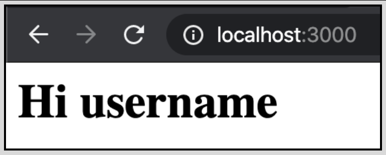
Name user
Email user@email.com
Password ....
Register

Login



```
listening on... 3000
[
  {
    id: 'dfIgJ9UgW',
    name: 'username',
    email: 'user@email.com',
    password: 'password123'
  }
]
```

Test: (Login User) → 3 cases

1	**Case: Failure (Email)** Go to https://localhost:3000/login ***Submit wrong email.***	 **Login** Email wrong@email.com Password •••••••••• Login Register	 **Login** No user with that email Email Password Login Register
2	**Case : Failure (Password)** Go to https://localhost:3000/login ***Submit wrong password.***	 **Login** Email user@email.com Password ••• Login Register	 **Login** Password incorrect Email Password Login Register
3	**Case: Success** Go to https://localhost:3000/login ***Submit correct email and password.***	 **Login** Email user@email.com Password ••••••••••• Login Register	 localhost:3000 **Hi username**

How does it work?

(Key, value) assigned for messages that may then be accessed by all Express middleware by appending data into the request header. This allows Flash messages to be passed without sending back a response to the client.

Iteration 12: Protect Routes (Home)

'Approach' → Plan phase

Goal #12: Protect Routes from Non-Authenticated Users: (Home)

Approach: **Passport Sessions, (isAuthenticated prop)**

Prevent non-authenticated users from accessing private information. Checks the Request via Passport sessions.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: auth.js → checks if user is authenticated, if so continue, otherwise redirect to login
- **Step 2**: user-routes.js → import 'checkAuthenticated' middleware function into user-routes
- **Step 3**: user-routes.js → use the checkAuthenticated function before using the controller function

Step 1 (JS): ***auth.js*** → Create: middlewares/auth.js

Create middleware function that checks if user is authenticated, if so continue, otherwise redirect to login

middlewares/auth.js

```
exports.checkAuthenticated = function(request, response, next){
   if (request.isAuthenticated()){
      return next();
   }
    response.redirect('/login')
}
```

Step 2 (JS): ***user-routes.js*** → Refactor: requires

Import 'checkAuthenticated' middleware function into user-routes

routes/user-routes.js

```
const { checkAuthenticated } = require('../middlewares/auth');
```

Step 3 (JS): ***user-routes.js*** → Refactor: route: GET '/'

Refactor the route on GET '/' to use the checkAuthenticated function before using the controller function

routes/user-routes.js

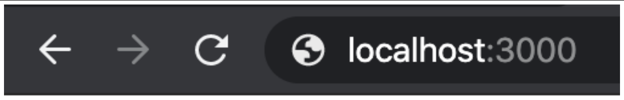
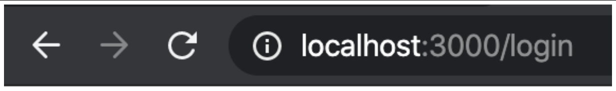
```
router.get('/', checkAuthenticated, controller.getIndex );
```

'Assess' → Test phase

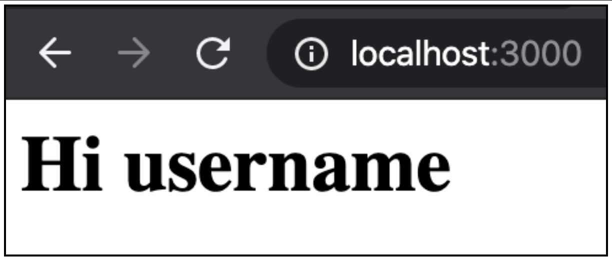
Launch app from terminal

```
npm start
```

Test Home (Unauthorized User)

If the user isn't logged in, Then any attempt to go to Home URL Redirects to the Login page	 localhost:3000  localhost:3000/login

Test Home (Authorized User)

If the user is logged in, then successfully go to Home URL	 localhost:3000 Hi username

How does it work?

Express middleware is server logic (i.e. modules, libraries, functions) that gets invoked between receiving the HTTP request and issuing the response.

Iteration 13: Protect Routes: (Login, Register)

'Approach' → Plan phase

Goal #13: Protect Routes from Authenticated Users: (Login, Register)

Approach: **Passport Sessions, (isAuthenticated prop)**

Redirect requests from authorized users from accessing register and login pages

---

'Apply' → Do phase

JavaScript Steps

- **Step 1**: auth.js → checks if user is authenticated, if so redirect to home, otherwise continue
- **Step 2**: user-routes.js → import 'checkNotAuthenticated' middleware function into user-routes
- **Step 3**: user-routes.js → use checkNotAuthenticated function before using controller functions

Step 1 (JS): ***auth.js*** → Create method: checkNotAuthenticated()

Create middleware function that checks if user is authenticated, if so redirect to home, otherwise continue

middlewares/auth.js

```
exports.checkNotAuthenticated = function(request, response, next){
   if (request.isAuthenticated()){
      return response.redirect('/');
   }
   next();
}
```

Step 2 (JS): ***user-routes.js*** → Refactor: requires

Import 'checkNotAuthenticated' middleware function into user-routes

routes/user-routes.js

```
const { checkAuthenticated, checkNotAuthenticated } = require('../middlewares/auth');
```

Step 3 (JS): ***user-routes.js*** → Refactor: routes: login, register

Refactor login,register routes to use checkNotAuthenticated function before using the controller functions

routes/user-routes.js

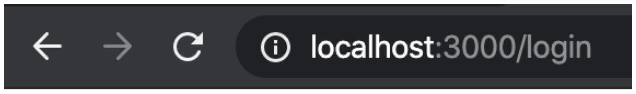
```
router.get('/login', checkNotAuthenticated, controller.getLogin);
router.get('/register', checkNotAuthenticated, controller.getRegister);
router.post('/login', checkNotAuthenticated, controller.postLogin);
router.post('/register', checkNotAuthenticated, controller.postRegister);
```

'Assess' → Test phase

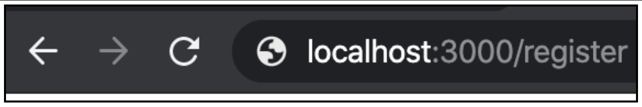
Launch app from terminal

```
npm start
```

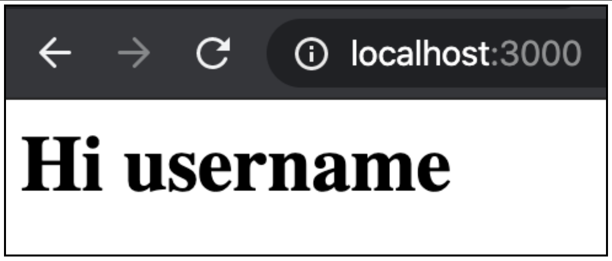
Test Login (Unauthorized User)

If no user is logged in, Then successfully go to '/login"	 localhost:3000/login

Test Register (Unauthorized User)

If no user is logged in, Then successfully go to '/register"	 localhost:3000/register

Test Login & Register (Authorized User)

If the user is logged in, Then they redirect to Home URL	 localhost:3000 Hi username

Iteration 14: User Logout

'Approach' → Plan phase

Goal #14: Option for user to logout and redirect to login page

Approach: Passport Sessions, (logout function)

Passport sessions provides a logout method within the Request object

---

'Apply' → Do phase

JavaScript Steps

- **Step 1**: user-routes.js → POST to '/logout' that assigns middleware and controller functions
- **Step 2**: user-controllers.js → postLogout controller invokes logout function in Request object.
- **Step 3**: index.ejs → add HTML form to POST logout from the Home page.

Step 1 (JS): ***user-routes.js*** → POST '/logout'

Add a route for POST request to '/logout' that assigns middleware and controller functions

routes/user-routes.js

```
router.post('/logout', checkAuthenticated, controller.postLogout);
```

Step 2 (JS): ***user-controllers.js*** → Refactor: UserControllers (class)

Add postLogout method to UserControllers class that invokes logout function in Request object.

controllers/user-controllers.js → UserControllers

```
postLogout(request, response){
   request.logOut();
   response.redirect('login');
}
```

Step 3 (EJS): ***index.ejs*** → Refactor: EJS

Add HTML form for POST to logout in the Home page.

views/index.ejs

```
<h1>Hi <%= name %> </h1>
<form action="/logout" method="POST">
   <button type="submit">Logout</button>
</form>
```

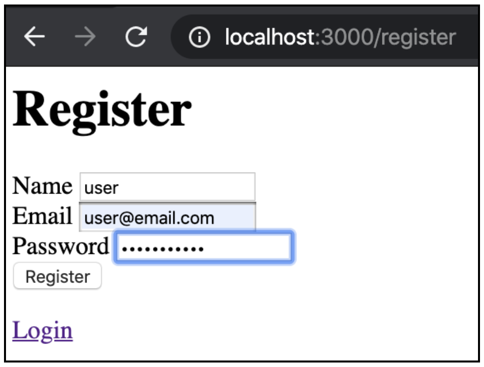
'Assess' → Test phase

Launch app from terminal

```
npm start
```

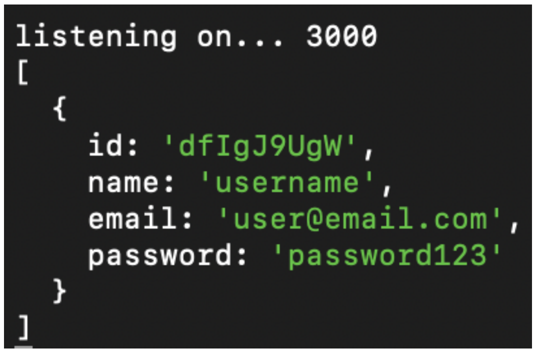
Setup (Register a User)

Open browser: http://localhost:3000/register & **register a user**, then test the 3 cases for logins


localhost:3000/register

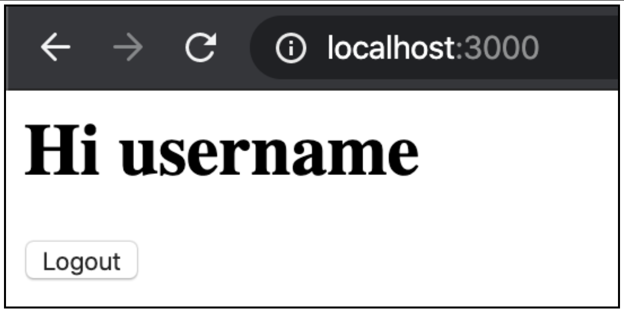
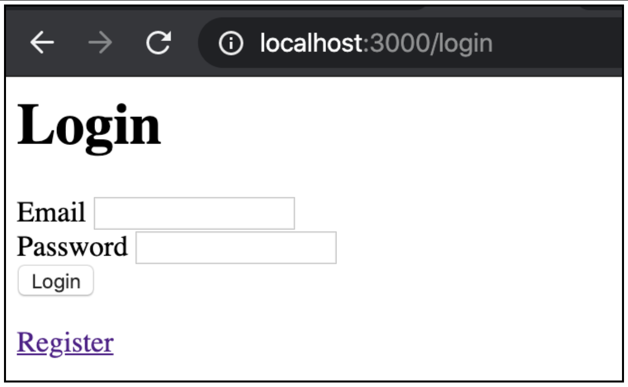
Register

Name	user
Email	user@email.com
Password	•••••••••••



```
listening on... 3000
[
  {
    id: 'dfIgJ9UgW',
    name: 'username',
    email: 'user@email.com',
    password: 'password123'
  }
]
```

Test Home (Logout)

Logout button should: • Logout user • Redirect to login page	 localhost:3000 **Hi username** Logout  localhost:3000/login **Login** Email Password Login Register

Iteration 15: Hash Passwords with Bcrypt

'Approach' → Plan phase

Goal #15: Hash Passwords with Bcrypt

Approach: Bcrypt

Bcrypt is a Node module used to securely hash passwords and compare login in attempts to hashed passwords. You should never store user passwords without encrypting them.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: package.json → add dependencies for: 'bcrypt'
- **Step 2**: users-model.js → import the 'bcrypt' module into users-model
- **Step 3**: users-model.js → hash the password using bcrypt.
- **Step 4**: passport-config.js → Import the 'bcrypt' module into passport-config.js
- **Step 5**: passport-config.js → authenticateUser compares user submission to the hash.

Step 1 (JSON): ***package.json*** → Refactor: dependencies

Add 'bcrypt' to project dependencies

package.json

```
{
  "name": "user-login-app",
  "version": "1.0.0",
  "description": "This project implements user authentication using passport",
  "main": "app.js",
  "scripts": {
     "start": "npm install && node app"
  },
  "author": "me",
  "dependencies": {
     "express": "*",
     "ejs": "*",
     "shortid": "*",
     "express-session": "*",
     "passport": "*",
     "passport-local": "*",
     "express-flash": "*",
     "bcrypt": "*"
  }
}
```

Step 2 (JS): ***users-model.js*** → requires

Import the 'bcrypt' module into users-model

models/users-model.js

```
const bcrypt = require('bcrypt');
```

Step 3 (JS): ***users-model.js*** → Refactor: add()

Refactor add method to hash the password using bcrypt. This requires add() to be async. Use salt of 10.

models/users-model.js → Users

```
async add(name, email, password){
   const id = shortid.generate();
   const hashedPassword = await bcrypt.hash(password, 10);
   const user = {id:id, name:name, email:email, password:hashedPassword};
   this.users.push(user);
   console.log(this.users)
}
```

Step 4 (JS): ***passport-config.js*** → requires

Import the 'bcrypt' module into passport-config.js

middlewares/passport-config.js

```
const bcrypt = require('bcrypt');
```

Step 5 (JS): ***passport-config.js*** → Refactor: authenticateUser

Refactor authenticateUser method to compare submission to the hash. This requires it to become async.

middlewares/passport-config.js

```
async function authenticateUser(email, password, done){
   const user = users.findUser('email', email);
   if (user === undefined){
       return done(null, false, {message: 'No user with that email'});
   }
   if ( await bcrypt.compare(password, user.password) ){
       return done(null, user);
   }
   else{
       return done(null, false, {message: 'Password incorrect'});
   }
}
```

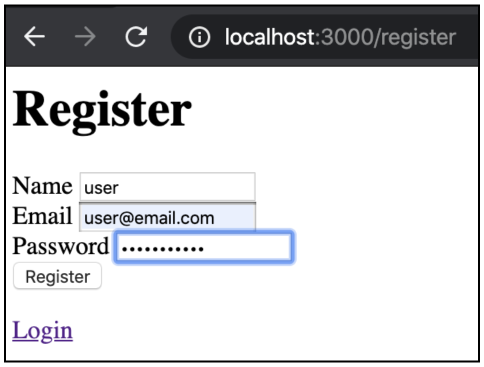
'Assess' → Test phase

Launch app from terminal

```
npm start
```

Test (Password Hashing)

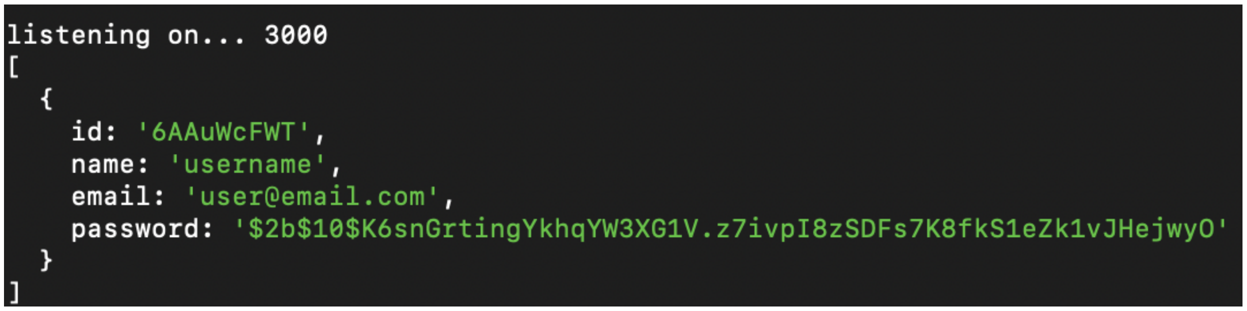
Open browser: http://localhost:3000/register & **register a user**, then check if hashed password


localhost:3000/register

Register

Name user
Email user@email.com
Password ••••••••••••
Register

Login



```
listening on... 3000
[
  {
    id: '6AAuWcFWT',
    name: 'username',
    email: 'user@email.com',
    password: '$2b$10$K6snGrtingYkhqYW3XG1V.z7ivpI8zSDFs7K8fkS1eZk1vJHejwyO'
  }
]
```

Iteration 16: Hide Secrets with dotenv

'Approach' → Plan phase

Goal #16: Hide Secret Information in process.env with dotenv

Approach: **dotenv, .env files, process.env**

Secure secret configuration data in a .env file and use dotenv package to access it during runtime

---

'Apply' → Do phase

JavaScript Steps

- **Step 1**: package.json → add dependencies for: 'dotenv'
- **Step 2**: app.js → If not production, then import the 'dotenv' and configure it into runtime env.
- **Step 3**: app.js → secret for express-session should load from process.env.SESSION
- **Step 4**: .env → create an environment file to hold your app's secret configuration information
- **Step 5**: .gitignore → create a gitignore file to prevent sharing the .env file and node_modules

Step 1 (JSON): ***package.json*** → Refactor: dependencies

Add 'dotenv' to project dependencies

package.json

```
{
  "name": "user-login-app",
  "version": "1.0.0",
  "description": "This project implements user authentication using passport",
  "main": "app.js",
  "scripts": {
     "start": "npm install && node app"
  },
  "author": "me",
  "dependencies": {
     "express": "*",
     "ejs": "*",
     "shortid": "*",
     "express-session": "*",
     "passport": "*",
     "passport-local": "*",
     "express-flash": "*",
     "bcrypt": "*",
     "dotenv": "*"
  }
}
```

Step 2 (JS): ***app.js*** → Require

If not in production, then import the 'dotenv' module and configure it into the runtime process env.

app.js → top of code

```
if (process.env.NODE_ENV !== 'production' ){
   require('dotenv').config()
}
```

Step 3 (JS): ***app.js*** → Refactor: setupApp()

Refactor setupApp function, so the secret for express-session loads from process.env.SESSION

app.js

```
function setupApp(){
   app.set('view-engine', 'ejs');
   app.use( bodyParser.urlencoded({ extended:false }) );
   const sessionConfig = { secret: process.env.SESSION, resave:false,saveUninitialized:true };
   app.use( session(sessionConfig) );
   app.use('/', userRoutes);
}
```

Step 4 (env): ***.env*** → Create file

Create an environment file to hold your app's secret configuration information

.env

```
 SESSION = secret-word
```

Step 5 (git): ***.gitignore*** → Create file

Create a gitignore file to prevent sharing the .env file and node_module

.gitignore

```
.env
node_modules
```

Note: In production, configure the environmental variables directly on the hosting machine to ensure that remain private. Avoid pushing any secret information onto git repo.

'Assess' → Test phase

Launch app from terminal

```
npm start
```

Conclusions

Final Comments

In this lab you implemented a User Login App. This lab covered: Passport, Bsync, EJS, Express-Sessions, Express-Flash, dotenv.

Future Improvements

- Connect to the Users collection to a database
- Prevent duplicate emails from being input
- Use additional Passport strategies
- Try other passport authorizations: JWT or oauth
- Deploy into production on heroku

Lab Submission

Compress your project folder into a zip file and submit on Moodle.

TED TODO: Split this into 2 Labs: 18: Sessions, 19: User Auth

18: Express Sessions to maintain data for same Request (8 iterations) Part 1

19: Passport, user authentication application (8 iterations) Part 2