Lab 9: Angular Authentication

Introduction: In this lab, we will combine the backend Node authentication solution using POWS in Lab 07 with an Angular frontend. We'll do a more complete user model using MongoDB and the Mongoose library for NodeJS.

Credits: An early version of this lab was written by Visutr Boonnateephisit. It adapts code from Lab 04: UserAuth and Jason Watmore's authentication tutorial (`https://jasonwatmore.com/post/2019/06/10/angular-8-user-registration-and-login-example-tutorial`), though I've replaced the JWT auth tokens in that tutorial with POWS.

NodeJS Runtime and Angular CLI Installation

Get NVM, Node, and Angular installed per previous labs.

Angular Application

1. Create a directory `quoteboard` for your application with a subdirectory `backend`.
2. In the backend directory, run `npm init` to start the project. Specify `src/server.ts` as the entry point. You can use defaults for all of the other settings at this point. To run, just do a `node src/server.ts`.
3. Install packages: `sudo apt install redis-server mongodb`
4. In the backend, run

```
% npm install --save argon2 cors express express-session csurf \
                     mongoose redis connect-redis \
                     passport passport-local promise
```

5. We will use Argon2 for hashing, express-session to generate session tokens, Mongoose as a connector for MongoDB, Passport for password-based authentication, and Promise for asynchronous responses to requests.

   A JavaScript application is only allowed to make AJAX requests to the server that it originated from. The origin of a request is identified by hostname (localhost, mydomain.com, etc.) or IP address (127.0.0.1, 192.168.1.1, etc.) in combination with the port (3000, 4200), the client is hosted on. Our backend is going to be listening for incoming requests on port 3000, but our Angular application will be served from port 4200, meaning that our Angular client has a different origin than our API server. This means without some intervention, every request by our Angular application will be rejected. However, we will enable CORS (Cross-origin Resource Sharing) to allow our API server to respond to a list of known origins.

6. To get correct typechecking from VSCode or WebStorm and the Typescript compiler, add some development dependencies:

```
% npm install --save-dev @types/express @types/node typescript
```

7. Here's a basic structure for the main file `src/server.ts`. Note that you may get some errors due to library APIs changing since the time this tutorial was written. However, you should be able to diagnose the causes of such issues and fix them yourself, or ask the instructor for help.

```
// Load Express

import express from 'express';
import session from 'express-session'

const app = express();

// Turn on parsing of incoming JSON

app.use(express.json());

// Enable CORS

import cors from 'cors';
app.use(cors());

// Enable CSRF tokens using the double submit cookie pattern for CSRF protection

import csurf from 'csurf';
const csrfFilter = csurf({ cookie: { sameSite: true } });
app.use(csrfFilter);
app.all("*", (req, res, next) => {
    const csrfReq = req as { csrfToken?(): string };
    const token = csrfReq.csrfToken();
    res.cookie("XSRF-TOKEN", token, { sameSite: true });
    return next();
});
app.use(function (err, req, res, next) {
    if (err.code !== 'EBADCSRFTOKEN') return next(err);
    console.log('CSRF error accessing', req.url);
    res.status(403);
    res.send('Error: invalid CSRF token');
});

// Connect to Mongo

const config = {
    dbUrl: 'mongodb://localhost/quoteboard-development',
    listenPort: 3333
};

import mongoose from 'mongoose';

console.log('Attempting to connect to MongoDB...');
mongoose.connect(config.dbUrl, { })
    .then(() => {
        console.log('Mongo connection successful.');
    })
    .catch((err) => {
        console.log('Mongo connection failed. Reason:');
        console.log(err);
    });

// Load user controller
```

```
import { userController } from './controllers/user-controller';
app.use(userController);

// Log invalid incoming URLs

app.use((req: any, res, next) => {
    if (!req.route) {
        console.log('Invalid request for URL ' + req.url);
    }
    next();
});

// Start server

console.log('Listening on port ' + config.listenPort);
app.listen(config.listenPort);
```

8. Let's start with an empty controller in `src/controllers/user-controller.ts`:

```
import express from 'express';

const userController = express();
export { userController };

// Routes

userController.route('/api/register').post(registerRoute);

function registerRoute() {

}
```

9. Set up the typsecript compiler in `tsconfig.json` at the top of the backend project tree:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true,
    "outDir": "build",
    "esModuleInterop": true
  },
  "exclude": [
    "node_modules"
  ],
  "include": [
    "src/**/*"
  ]
}
```

   To get VSCode to automatically recompile the Typescript code to JavaScript as we change it, once `tsconfig.json` is set up (more information is available at `https://code.visualstudio.com/docs/typescript/typescript-compiling` if needed), then in VSCode, press Ctrl+Shift+B to open your

task list and select `tsc: watch - tsconfig.json`. Every time you save, the Typescript code will automatically be compiled.

To get WebStorm to do the same thing, go to the project settings, "Language and Frameworks," "Typescript," and choose "Recompile on changes." You may need to restart WebStorm after you make this change for it to take effect.

10. Go ahead and run the application (`node build/server.js`). You should be able to visit `http://localhost:3333` in the browser and see the error message on the console.

Preparing the frontend application

1. Create a new Angular application under `quoteboard/` called `frontend`:

   ```
   $ ng new frontend
   ```

   Select Angular routing, and select SCSS as the stylesheet type.

2. Let's make our registration component first (`ng generate component register`). Add `ReactiveFormsModule` to the application's module imports and the following HTML for the component:

   ```
   <div class="row">
     <div class="col-lg-12">

       <h2>Register</h2>

       <form [formGroup]="registerForm" (ngSubmit)="onSubmit()">
         <div class="form-group">
           <label for="name">Name</label>
           <input id="name" type="text" formControlName="name" class="form-control"
                  [ngClass]="{ 'is-invalid': f.name.invalid && (submitted || (f.name.dirty
                   && f.name.touched))}" required/>
           <div *ngIf="f.name.invalid" class="invalid-feedback">
             <div *ngIf="f.name.errors.required">Name is required</div>
           </div>
         </div>
         <div class="form-group">
           <label for="email">Email</label>
           <input id="email" type="text" formControlName="email" class="form-control"
                  [ngClass]="{ 'is-invalid': f.email.errors && (submitted || (f.email.dirty &&
                   f.email.touched))}" required/>
           <div *ngIf="f.email.errors && f.email.dirty && f.email.touched" class="invalid-feedback">
             <div *ngIf="f.email.errors.required">Email is required</div>
           </div>
         </div>
         <div class="form-group">
           <label for="password">Password</label>
           <input id="password" type="password" formControlName="password" class="form-control"
                  [ngClass]="{ 'is-invalid': f.password.errors && (submitted || (f.password.dirty
                   && f.password.touched))}" required/>
           <div *ngIf="f.password.errors" class="invalid-feedback">
             <div *ngIf="f.password.errors.required">Password is required</div>
             <div *ngIf="f.password.errors.minlength">Password must be at least 6 characters</div>
           </div>
         </div>
         <div class="form-group">
   ```

```
        <label for="passwordConfirmation">Password confirmation</label>
        <input id="passwordConfirmation" type="password" formControlName="passwordConfirmation"
               class="form-control"
               [ngClass]="{ 'is-invalid': (f.passwordConfirmation.errors ||
                 registerForm.hasError('notMatching')) && (submitted ||
                 f.passwordConfirmation.dirty
                 || f.passwordConfirmation.touched) }" required/>
        <div *ngIf="f.passwordConfirmation.errors || registerForm.hasError('notMatching')"
             class="invalid-feedback">
             <div *ngIf="f.passwordConfirmation.errors &&
             f.passwordConfirmation.errors.required">Password
               confirmation is required</div>
          <div *ngIf="registerForm.hasError('notMatching')">Password confirmation must match
            password</div>
        </div>
      </div>
      <div class="form-group">
        <button [disabled]="loading" class="btn btn-primary">
          <span *ngIf="loading" class="spinner-border spinner-border-sm mr-1"></span>
          Register
        </button>
        <a routerLink="/login" class="btn btn-link">Cancel</a>
      </div>
    </form>
  </div>
</div>
```

3. We'll need three services: alerts, authentication, and users. The structure is from Jason Watmore's tutorial. Thanks, Jason! Put the alert service in `src/app/_services/alert.service.ts`:

```
import { Injectable } from '@angular/core';
import { Router, NavigationStart } from '@angular/router';
import { Observable, Subject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class AlertService {
  private subject = new Subject<any>();
  private keepAfterRouteChange = false;

  constructor(private router: Router) {
    // clear alert messages on route change unless 'keepAfterRouteChange' flag is true
    this.router.events.subscribe(event => {
      if (event instanceof NavigationStart) {
        if (this.keepAfterRouteChange) {
          // only keep for a single route change
          this.keepAfterRouteChange = false;
        } else {
          // clear alert message
          this.clear();
        }
      }
    });
  }
```

```
  getAlert(): Observable<any> {
    return this.subject.asObservable();
  }

  success(message: string, keepAfterRouteChange = false): void {
    this.keepAfterRouteChange = keepAfterRouteChange;
    this.subject.next({ type: 'success', text: message });
  }

  error(message: string, keepAfterRouteChange = false): void {
    this.keepAfterRouteChange = keepAfterRouteChange;
    this.subject.next({ type: 'error', text: message });
  }

  clear(): void {
    // clear by calling subject.next() without parameters
    this.subject.next();
  }
}
```

4. Next, the authentication service:

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Injectable({ providedIn: 'root' })
export class AuthenticationService {
  private currentUserSubject: BehaviorSubject<any>;
  public currentUser: Observable<any>;
  constructor(
    private http: HttpClient,
  ) {
    if (!sessionStorage.getItem('currentUser') && localStorage.getItem('currentUser')) {
      sessionStorage.setItem('currentUser', localStorage.getItem('currentUser'));
    }
    this.currentUserSubject = new
        BehaviorSubject<any>(JSON.parse(sessionStorage.getItem('currentUser')));
    this.currentUser = this.currentUserSubject.asObservable();
  }
  public get currentUserValue(): any {
    return this.currentUserSubject.value;
  }

  // Store user details in local storage to keep user logged in between page refreshes

  setUser(user): void {
    sessionStorage.setItem('currentUser', JSON.stringify(user));
    this.currentUserSubject.next(user);
  }

  login(username, password): any {
    const subject = this.http.post<any>('/api/login', {
```

```
      email: username, password });
    return subject
      .pipe(map(user => {
        this.setUser(user);
        return user;
      }));
  }

  logout(): any {
    sessionStorage.clear();
    localStorage.clear();
    this.currentUserSubject.next(null);
    const url = '/api/logout';
    const subject = this.http.post<any>(url, null);
    return subject
      .pipe(map(status => {
        return status;
      }));
  }
}
```

5. And now, the user service:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({ providedIn: 'root' })
export class UserService {
  constructor(private http: HttpClient) { }
  getAll(): any {
    const subject = this.http.get<any[]>('/api/users');
    return subject;
  }

  register(user): any {
    const subject = this.http.post('/api/register', user);
    return subject;
  }

  delete(id): any {
    id = encodeURIComponent(id);
    const subject = this.http.delete(`/api/users/${id}`);
    return subject;
  }

  update(userData): any {
    const subject = this.http.put(`/api/users/${userData.id}`, userData);
    return subject;
  }
}
```

6. Now for the body of our first component:

```
import { Component, OnInit } from '@angular/core';
```

```
import { FormBuilder, FormGroup, Validators} from '@angular/forms';
import { Router } from '@angular/router';
import { first } from 'rxjs/operators';
import { AuthenticationService } from '../_services/authentication.service';
import { UserService } from '../_services/user.service';
import { AlertService } from '../_services/alert.service';

function passwordMatchValidator(group: FormGroup): any {
  if (group) {
    if (group.get('password').value !== group.get('passwordConfirmation').value) {
      return { notMatching : true };
    }
  }
  return null;
}

@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.scss']
})
export class RegisterComponent implements OnInit {
  registerForm: FormGroup;
  loading = false;
  submitted = false;

  constructor(
    private formBuilder: FormBuilder,
    public router: Router,
    private authenticationService: AuthenticationService,
    private userService: UserService,
    public alertService: AlertService
  ) {
    // redirect to home if already logged in
    if (this.authenticationService.currentUserValue) {
      this.router.navigate(['/']);
    }
  }

  ngOnInit() {
    this.registerForm = this.formBuilder.group({
      name: ['', Validators.required],
      email: ['', Validators.required],
      password: ['', [Validators.required, Validators.minLength(6)]],
      passwordConfirmation: ['', [Validators.required]]
    });
    this.registerForm.setValidators(passwordMatchValidator);
  }

  // convenience getter for easy access to form fields
  get f() { return this.registerForm.controls; }

  onSubmit() {
    this.submitted = true;
```

```
    // stop here if form is invalid
    if (this.registerForm.invalid) {
      return;
    }

    this.loading = true;
    this.userService.register(this.registerForm.value)
      .pipe(first())
      .subscribe(
        data => {
          this.alertService.success('Registration successful', true);
          this.router.navigate(['/login']);
        },
        error => {
          this.alertService.error('Error contacting server: ' + error.message);
          this.loading = false;
        });
  }
}
```

7. For the top-level template, try

```
<div id="holder">
  <header class="fixed-top">
    <nav class="navbar navbar-expand">
      <div class="navbar-nav" *ngIf="currentUser else noUser">
        <img height="40" class="nav navbar-nav navbar-left" routerLink="/"
          src="assets/images/logo.png"/>
        <a class="nav navbar-nav nav-item nav-link navbar-left" routerLink="/">Quoteboard</a>
        <div class="nav navbar-nav navbar-center">
          <a class="nav-item nav-link navbar-center" routerLink="/home">Home</a>
          <a class="nav-item nav-link navbar-center"
            routerLink="/profile">{{currentUser.name}}'s Profile</a>
          <a class="nav-item nav-link navbar-center" *ngIf="currentUser.isAdmin"
            routerLink="/admin">Admin Dashboard</a>
        </div>
        <a class="nav navbar-nav nav-item nav-link navbar-right" (click)="logout($event)"
          href="#">Logout</a>
      </div>
      <ng-template #noUser>
        <div class="navbar-nav">
          <img height="40" class="nav navbar-nav navbar-left" routerLink="/"
            src="assets/images/logo.png"/>
          <a class="nav navbar-nav nav-item nav-link navbar-left" routerLink="/">Quoteboard</a>
          <a class="nav navbar-nav nav-item nav-link navbar-center"
            routerLink="/register">Register</a>
          <div class="nav navbar-nav navbar-right">
            <a class="nav-item nav-link" routerLink="/login">Login</a>
          </div>
        </div>
      </ng-template>
    </nav>
  </header>
```

```
  <div id="body">
    <div class="container">
      <app-alert></app-alert>
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

8. And for the top level component's logic:

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { AuthenticationService } from './_services/authentication.service';
import { AlertService } from './_services/alert.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  currentUser: {
    email: string,
    name: string,
    token: string,
    isAdmin: boolean
  };

  title = 'Quoteboard';

  constructor(
    private router: Router,
    private authenticationService: AuthenticationService,
    private alertService: AlertService
  ) {
    this.authenticationService.currentUser.subscribe(x => this.currentUser = x);
  }

  logout(event): void {
    this.authenticationService.logout();
    this.router.navigate(['/login']);
    event.preventDefault();
  }
}
```

9. You'll need to create the alert component (`ng generate component alert`) with content

```
<div class="row">
  <div class="col-lg-12">
    <div *ngIf="message" [ngClass]="message.cssClass">{{message.text}}</div>
  </div>
</div>
```

10. Now, the alert component logic:

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subscription } from 'rxjs';
import { AlertService } from '../_services/alert.service';

@Component({
  selector: 'app-alert',
  templateUrl: './alert.component.html',
  styleUrls: ['./alert.component.scss']
})
export class AlertComponent implements OnInit, OnDestroy {
  private subscription: Subscription;
  message: any;

  constructor(private alertService: AlertService) { }

  ngOnInit(): void {
    this.subscription = this.alertService.getAlert()
      .subscribe(message => {
        switch (message && message.type) {
          case 'success':
            message.cssClass = 'alert alert-success';
            break;
          case 'error':
            message.cssClass = 'alert alert-danger';
            break;
        }
        this.message = message;
      });
  }

  ngOnDestroy(): void {
    if (this.subscription) {
      this.subscription.unsubscribe();
    }
  }
}
```

11. A few other things: add `HttpClientModule` as an import in the top-level module description. Install Bootstrap (`npm install bootstrap`) and add text

    ```
    @import "~bootstrap/dist/css/bootstrap.min.css";
    ```

    to `styles.scss` and add the following as a beginning point for your styling in the top-level component:

    ```
    #body {
      padding-bottom: 120px;    /* height of footer */
      margin: 0;
      padding-top: 55px;
    }

    #holder {
      min-height: 100%;
      position: relative;
    }
    ```

```
.navbar-nav.navbar-center {
  position: absolute;
  left: 50%;
  transform: translatex(-50%);
}

.navbar-nav.navbar-right {
  position: absolute;
  left: 100%;
  transform: translatex(-100%);
}

header {
  background: #3f6433;
  color: #fff;
}

header a {
  color: #fff;
}

nav {
  background: #3f6433;
  min-height: 0px;
  width: 100%;
  height: 55px;
}
```

12. Add some routes to `app-routing.module.ts`:

```
const routes: Routes = [
  // { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent }
];
```

13. Now your registration component should be hooked up and essentially working. Try it out – you should be able to render the registration form, and the validations should be working correctly. Watch the log when you click the "Register" button. You will see that it's doing a POST to `http://localhost:4200/api/register`. But we actually want the request to go to `http://localhost:3000/api/register`. We could hardcode the backend URL in the frontend application, but as it turns out, having different ports for the frontend and backend will cause trouble for our session cookies and CSRF cookies, as the change in port makes the browser think it's a different site and won't send the cookies. The solution to this is to use the Angular CLI server as a *proxy* for the backend service. To set this up, create a file `src/proxy.conf.json` in the frontend with contents

```
{
  "/api": {
    "target": "http://localhost:3333",
    "secure": false
  }
}
```

and in `angular.json`, add the proxy configuration to the `ng serve` configuration:

```
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "options": {
    "browserTarget": "frontend:build",
    "proxyConfig": "src/proxy.conf.json"
  },
```

Now your API server requests will go to the CLI server on port 4200, but if they begin with `/api`, they will be forwarded to your backend on port 3333.

Back to the backend

OK now we go back to the back end and get registration working.

1. First let's get session cookies working. In the main server, after loading Express:

   ```
   // Session cookies and CSRF tokens require cookie parser

   import cookieParser from 'cookie-parser';
   app.use(cookieParser());
   ```

   Then, after the CSRF token configuration, add

   ```
   // Load Express session config

   import session from 'express-session';
   import redis from 'redis';
   import connectRedis from 'connect-redis';

   const RedisStore = connectRedis(session);
   const redisClient = redis.createClient({ host: 'localhost' });
   const sessionConfig = session({
       store: new RedisStore({
           host: 'localhost',
           port: 6379,
           client: redisClient
       }),
       cookie: { secure: false, sameSite: true },
       secret: 'keyboard cat',
       resave: false,
       saveUninitialized: true
   });
   app.use(sessionConfig);
   ```

   Now if you test in the browser you should be receiving a session cookie on first connect, but you'll be getting a CSRF error.

2. Next let's fix up CSRF. We want to use the so-called "double submit cookies" approach to preventing CSRF attacks:[1] the backend, on the first unauthenticated request from the browser, generates a cryptographically strong pseudorandom value and returns it as a cookie (we use the cookie name `XSRF-TOKEN`). After that, any state-changing Ajax request (POST/PUT/PATCH/DELETE) must contain both the cookie (to ensure that the request is coming from the same browser that made the initial request) and a request header (to ensure that the JavaScript code generating the request is running in

[1] `https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html`

the same browser that made the initial request and came from the same site). To make this work, first, you need to configure CSURF library server side, as above. Make sure you tell Express to use cookie parser and body parser before configuring CSURF. Next, we need a GET request that the server can request before it sends any POST/PUT/PATCH/DELETE. One example is a server version request:

```
// Return version to frontend

app.route('/api/version').get(version);

function version(req, res) {
    res.send({ version: '0.1' });
}
```

3. To use this endpoint to fetch the CSRF cookie, add the following to the registration component:

```
  ...
  ngOnInit(): void {
    ...
    // Get backend version (and CSRF token)

    this.versionService.getVersion()
      .pipe(first())
      .subscribe(
        (versionInfo) => {
          this.serverVersion = versionInfo.version;
        },
        (error) => {
          this.alertService.error('Cannot contact server. Try again later.');
        });
  }
```

On initialization of the component, we're asking a "version service" for the backend server version. Not so much because we want the version, but because we want the CSRF token issued by the server! The version service should go in `src/app/_services/version.service.ts`:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Injectable({ providedIn: 'root' })
export class VersionService {
  constructor(
    private http: HttpClient
  ) {
  }

  getVersion(): any {
    const url = '/api/version';
    const subject = this.http.get<any>(url);
    return subject
      .pipe(map(version => {
        return version;
      }));
  }
}
```

Go ahead and try accessing the registration form. Now you should see that as soon as the component is initialized, it sends a request for `/api/version`. The server responds with a cookie called `XSRF-TOKEN` that we can use in subsequent POST/PATCH/PUT/DELETE requests.

4. Last modification for CSRF: we have to get the frontend to add a CSRF header to any POST / PUT / PATCH / DELETE request, using the `XSRF-TOKEN` cookie retrieved from a GET request. The cookie will automatically be attached to every request, and it will be accessible to our frontend JavaScript code as long as its `httpOnly` flag is set to false. Angular provides a module to intercept every outgoing Ajax request, read the cookie value, and use it to set a header. Set this up in `app.module.ts` with the import

```
HttpClientXsrfModule.withOptions({
  cookieName: 'XSRF-TOKEN',
  headerName: 'X-XSRF-TOKEN'
})
```

   This tells Angular to take the value of the `XSRF-TOKEN` cookie on outgoing requests and use that value to set a request header `X-XSRF-TOKEN`. On receipt, the backend will compare and validate the cookie and header before allowing the POST/PUT/PATCH/DELETE request. Now your POST request should pass CSRF checking, and you'll get a 404 (Not Found) error. To test if CSRF protection is really working, try changing the XRSR-TOKEN cookie to a different (invalid) value. You should then see a 403 (Forbidden) error.

5. OK! We're using best practices to avoid CSRF attacks, finally. Next, let's implement the back end route for user registration. First, here's a user service. Create a file src/services/user.service.ts and add the following content:

```
import passport from 'passport';
import * as passportLocal from "passport-local";
const LocalStrategy = passportLocal.Strategy;
import * as argon2 from 'argon2';
import * as fs from 'fs';
import { User } from '../models/user';

passport.use(new LocalStrategy ({usernameField: 'email'}, (username, password, done) => {
    return User.findOne({ email: username }, (error, user) => {
        if (error) {
            return done(error);
        }
        if (!user) {
            return done(null, false, {
                message: 'Invalid email or password'
            });
        }
        if (user) {
            return validateUserPassword(user.passwordHash, password)
                .then((validated) => {
                    if (validated) {
                        return done(user);
                    } else {
                        return done(null, false, {
                            message: 'Invalid email or password'
                        });
                    }
                })
```

```
                .catch((verror) => {
                    console.log('Error in password validation:');
                    console.log(verror);
                    return done(verror);
                });
        }
    });
}));

function validateUserPassword(hash, password) {
    return argon2.verify(hash, password)
        .then((verified) => {
            return verified;
        })
        .catch((error) => {
            console.log('Error in hash validation:', error);
            return error;
        });
}

const userService = {

    GetUsers: function () {
        return new Promise((resolve, reject) => {
            User.find()
                .then((users) => {
                    resolve(users);
                })
                .catch((error) => {
                    reject(error);
                });
        });
    },

    RegisterUser: (user) => {
        return registerUser(user)
            .then((result) => {
                return result;
            })
            .catch((error) => {
                console.log('Error in userService');
                console.log(error);
                return error;
            });
    },

    UpdateUser: (user, updates) => {
        let updatesAllowed = { };
        if (updates.id) updatesAllowed['id'] = updates.id;
        if (updates.name) updatesAllowed['name'] = updates.name;
        return new Promise((resolve, reject) => {
            user.updateOne(updatesAllowed, (err, result) => {
                if (err) {
                    console.log('Update failed: ' + err);
```

```
                    reject();
                } else {
                    console.log('Update success:');
                    console.log(result);
                    resolve();
                }
            });
        });
    },

    GetUser: (email) => {
        return getUser(email)
            .then((result) => {
                return result;
            })
            .catch((error) => {
                console.log('Error getting user');
                console.log(error);
                return error;
            });
    },

};

function registerUser(userObject) {
    return new Promise((resolve, reject) => {
        const user = new User();
        user.email = userObject.email;
        user.name = userObject.name;
        argon2.hash(userObject.password, { type: argon2.argon2id }).then((hash) => {
            user.passwordHash = hash;
            user.save((error) => {
                if (error) {
                    console.log('Error saving user');
                    console.log(error);
                    reject(error);
                }
                resolve({ emailAddress: user.email });
            });
        }).catch((error) => {
            console.log('Error hashing user password');
            console.log(error);
            reject(error);
        });
    });
}

interface LoginResult {
    user?: Object,
    token?: string,
    message?: string
}

function loginUser(req, res) {
```

```
    return new Promise<LoginResult>((resolve, reject) => {
        passport.authenticate('local', (user, error, info) => {
            if (error) {
                console.log('Error authenticating user:');
                console.log(error);
                reject(error);
            }
            if (info) {
                console.log('Info from authenticate user:');
                console.log(info);
                reject(info);
            }
            if (user) {
                console.log('Retrieved user data:');
                console.log(user);
                resolve({ user });
            }
        })(req, res);
    });
}

function getUser(email) {
    return User.findOne({email})
        .then((user) => {
            if (!user) {
                return ({message: 'Error getting user'});
            } else {
                return user;
            }
        });
}

export { userService };
```

6. Next, a partial user controller:

```
import * as express from 'express';
import { userService } from '../services/user.service'

const userController = express();
export { userController };

// Check double-submit cookies

userController.use((req, res, next) => {
    try {
        userService.VerifyXSRFToken(req, res);
    } catch (e) {
    }
    next();
});

// Routes
```

```
userController.route('/api/register').post(registerRoute);
userController.route('/api/login').post(loginRoute);

// Middleware to authenticate user after double submit cookie verification
interface AuthenticatedRequest extends Request {
    user: any,
    params: any,
    query: any,
    body: any
}
function checkIfAuthenticated(req: AuthenticatedRequest, res, next) {
    if (req.user) {
        req.user
            .then((user) => {
                delete req.user;
                req.user = user;
                return next();
            })
            .catch((err) => {
                console.log('Invalid authentication: ' + err.message);
                res.status(401).send({ message: 'User login required.' });
            });
    } else {
        console.log('Unauthenticated request.');
        res.status(401).send({ message: 'User login required.' });
    }
}

// Middleware to authorize request after authentication
function checkIfPermitted(req: AuthenticatedRequest, res, next) {
    const userRequested = req.params.userId;
    const userAuthenticated = req.user.id;
    if (userRequested === userAuthenticated || req.user.isAdmin) {
        return next();
    }
    console.log('Unauthorized request by user ' + req.user.id);
    res.status(401).send({ message: 'Unauthorized request.' });
}
// Middleware to authorize admin requests after authentication
function checkIfAdmin(req: AuthenticatedRequest, res, next) {
    const userAuthenticated = req.user.id;
    if (req.user.isAdmin) {
        return next();
    }
    console.log('Unauthorized request by user ' + req.user.id);
    res.status(401).send({ message: 'Unauthorized request.' });
}

//Route to register a user
function registerRoute(req, res) {
    if (req && req.body && req.body.email) {
        console.log('Request to authenticate ' + encodeURIComponent(req.body.email));
        userService.RegisterUser(req.body)
            .then((result) => {
```

```
                if (result) {
                    let message = 'Registration successful.';
                    res.send({message});
                } else if (result && result.errmsg && result.errmsg.match('duplicate key')) {
                    console.log("duplicate key")
                    res.status(409).send({message: 'Email is already taken.'});
                } else {
                    console.log('Unexpected registration result:');
                    console.log(result);
                    res.sendStatus(500);
                }
            })
            .catch((error) => {
                console.log('Caught registration error:');
                console.log(error);
                res.sendStatus(500);
            });
    } else {
        console.log('Invalid request: ' + encodeURIComponent(req.body.toString()));
        res.sendStatus(400);
    }
}

// Route to login a user

function loginRoute(req, res) {
    // Code for login
}

// Route to logout a user

userController.route('/logout').post(userLogout);

function userLogout(req, res) {
    if (req.session.username) {
        req.session.username = null;
        res.sendStatus(200);
    } else {
        res.sendStatus(400);
    }
}
```

7. Here is a user model:

```
import mongoose from 'mongoose';

const UserSchema = new mongoose.Schema({
    email: {
        type: String,
        unique: true,
        required: true
    },
    passwordHash: {
        type: String,
```

```
        required: true
    },
    name: {
        type: String,
        required: true
    },
    message: {
        type: String
    },
    isAdmin: {
        type: Boolean,
        default: false
    }
});
const User = mongoose.model('User', UserSchema);
export { User };
```

See if you can put the pieces together to get the registration flow working. Then you have two tasks:

1. Get the login flow working.
2. Force a reset of the session token on login.

That's it for today's lab! We see that a full-fledged full stack application using a NodeJS backend requires a lot of moving parts, but when put together carefully, we can build an application that follows best practices for avoidance of CSRF attacks and session fixation attacks, things we get for free with Rails and Devise.