Proje için gerekli paket kurulumları

Passport js nodejs uygulamalarımızda authentication işlemlerinde kullanabileceğimiz bir pakettir. Google, Facebook, Github gibi çeşitli hesaplar vasıtası ile de giriş yapılabilir. Öncelikle boş, yeni bir proje açıp gerekli dosyalarımızı oluşturduktan sonra passportjs paketini import ediyoruz.

npm init -y komutu ile **package.json** dosyasını oluşturuyoruz.

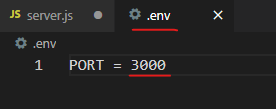
npm i express ve npm i passport ile paket kurulumlarını yaptık.

Redirect işlemlerinde kullanıcılara doğru mesajlar verebilmek için gereken paket: **connect-flash**

Email ve şifre ile giriş işlemlerinin yapılması için kurulması gereken paket **passport-local**

Oturum işlemlerini yönetmemizi sağlayan paket: **express-session**

Şimdi de tüm veritabanı bağlantıları, tokenlar vs. birarada barındıran ve gerektiği yerde gereken modülün çağrılmasını sağlayan bir paket olan **dotenv paketini** kuralım.

npm i dotenv ile indirdik. Sonra **.env** adında bir dosya oluşturduk. Burada port numarasını belirledik:

app.listen(process.env.PORT, () => {

    console.log(`server is running on port ${process.env.PORT}`);

});

Görüldüğü üzere port ayaklandırma işlemini dotenv paketinin özelliklerini kullanarak yapıyoruz.

Son olarak ejs ve express-ejs-layouts paketlerini kurduktan sonra ejs için gerekli tüm kodlar:

const ejs = require('ejs');

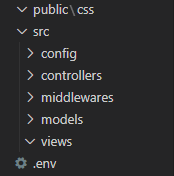
const expressLayouts = require('express-ejs-layouts');

const path = require('path');

app.use(expressLayouts);

app.set('view-engine', 'ejs');

app.set('views', path.resolve(\_\_dirname, './src/views'));

MVC yapısının oluşması için gerekli dizinleri oluşturduk:

Config’de database bağlantı kodlarımızı yazalım.

Burada database.js adlı dosya oluşturduktan sonra mongoose paketinin kurulumunu yaptık: npm i mongoose

MongoDB Compass uygulamasından yeni bir db oluşturduk.

.env’de db için dosya yolunu yazdık:

MONGODB\_CONNECTION\_STRING = mongodb://localhost/panel

Sonra database.js’te db bağlantısı sağlamaya çalıştık:

const mongoose = require('mongoose');

mongoose.connect(process.env.MONGODB\_CONNECTION\_STRING)

    .then(() => console.log('connected database successfully'))

    .catch((error) => console.log('database connection error' + error));

Sonra bu dosyayı server.js’te çağıralım:

require('./src/config/database');

Öncelikle bir admin teması bulup dosyalarını indiriyoruz:

https://startbootstrap.com/theme/sb-admin-2

Bu temanın dosyalarını public dizini altında admin dizininin içerisine yapıştırıyoruz. Public dizininin içerisinde **admin** ve **public** isimli iki dizin oluşturduk.

Login, register ve forgot\_password sayfalarında ortak olan şablon kodlar vardı, onları alıp auth\_layout.ejs dosyasına yapıştırdık ve örneğin;

app.get('/login', (req, res) => {

    res.render('login', { layout:'./layout/auth\_layout.ejs' });

});

Bu kodda login sayfasına get isteği gönderildiğinde önce auth\_layout.ejs dosyasına gidip kodları çalıştırdıktan sonra body kısmında <%- body %> yazan kısma da login ile ilgili kodları çalıştıracak.

Authentication işlemleri ile ilgili yaptığımız route’ların hepsini routes içerisinde oluşturduğumuz auth\_router.js’e aktarıyoruz.

const router = require('express').Router();

router.get('/login', (req, res) => {

    res.render('login', {layout: './layout/auth\_layout.ejs'});

});

Bunun gibi register ve forgot-password rotalarını da yazdıktan sonra export ediyoruz:

module.exports = router;

ve bu router’ı server.js’te çağırıyoruz:

const authRouter = require('./src/routes/auth\_routes');

app.get('/', authRouter),

auth\_router’ın son hali:

const router = require('express').Router();

const authController = require('../controllers/auth\_controller');

router.get('/login', authController.showLoginForm);

router.get('/register', authController.showRegisterForm);

router.get('/forgot-password', authController.showForgotPasswordForm);

module.exports = router;

Şimdi de register sayfamızdaki formu doldurduktan sonra gönderilecek olan post isteğini kodlayalım, ondan önce register sayfasındaki form etiketinde metodu ve action’u yazalım:

<form class="user" action="/register" method="POST">

Ve Register butonunun type’ını submit yapmalıyız:

<button type="submit" ...

Sonra form inputlarını doldurduktan sonra register butonuna basıldığında user bilgilerinin gönderilip gönderilmediğini test etmek için auth\_controller’da bir metod yazalım:

const register = (req, res) => {

    res.send(`

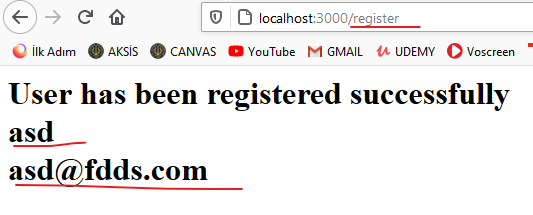
<h1> User has been registered successfully <br> ${req.body.firstName}<br> ${req.body} </h1>

`);

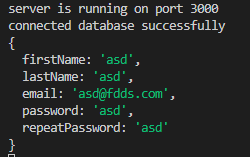
};

Sonra bu metodu user\_router’da ilgili yerde çağıralım:

router.post('/register', authController.register);

Test etmeden önce, ekranda request’imizin body’sinden gelen user bilgilerini yazdırabilmek için urlencoded middleware’ini kullanmamız gerekiyor. Bu metod formdan gelen bilgileri, post isteği ile gelen user bilgilerini elde etmemizi sağlayacak. Server.js’te çağırdık:

app.use(express.urlencoded({ extended: true }));

Sonunda register formundaki inputları doldurup register butonuna bastığımızda, firstName ve email alanlarını yazdırarak test ettik ve başarılı sonuç aldık:

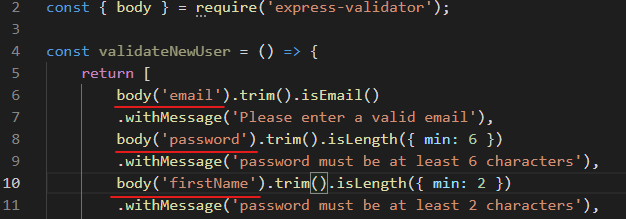
console.log(req.body); ile yazdırdığımızda da

konsoldaki çıktı:

Şimdi de formlara girilen verilerin validation işlemini yapacağız. Veriler controller’a varmadan kontrol edip ona göre controller’lara göndereceğiz. Bunu da Express-validator modülü ile yapalım:

npm i --save express-validator

Bu validation işlemlerini middleware olarak çalıştıracağımız için middlewares dizinimizde validation\_middleware.js adlı bir dosya oluşturuyoruz.

Bu modül bizden dizi döndürmemizi istediği için alanlarımızı validation yapılmış halde yazıp return ediyoruz, express-validator modülüne ait metodlarla gerekli kontrolleri yapıyoruz.

Sonra auth\_middleware’de olası hata yakalanması durumunda gerekli kontrolleri yapması için :

const { validationResult } = require('express-validator');

Bu nesne ile yakalanan hataları bir dizinin içerisine atalım:

authController’a gitmeden validation yapmamız için auth\_router’daki post metodumuzda validator middleware’imizi çağırmamız gerekiyordu:

router.post('/register', validationMiddleware.validateNewUser(), authController.register);

Çıkabilecek hataları dizi olarak sayfaya gönderip kullanıcıya gerekli uyarı mesajlarını verebilmek için render() metodu ile sayfaya gönderiyoruz.

Bu noktada kullanıcı oturumu (session) yönetimi yapmamız gerekiyor ki olası validation hatalarında, hataları kullanıcının sonraki denemelerinde de bir kereye mahsus tutabilmek için işimizi kolaylaştıracak paket olan connect-flash paketini kullanmamız gerekiyor. Bu paketi kullanabilmek için de session yapısına ihtiyacımız var, bunun için de express session paketini kurduk.

Express-session paketini kurduktan sonra (npm i express-session) import ediyoruz:

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

Session’u middleware olarak yazıyoruz:

app.use(session({

    secret: process.env.SESSION\_SECRET,

    resave: false,

    saveUninitialized: true // true olursa req.session’u hafızada kaydeder

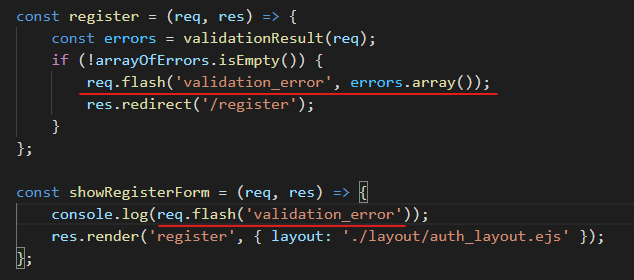
}));

Şimdi de connect-flash paketini çalıştıralım. Bu paket sayesinde validation hatalarını tutabiliriz. Paketi kurduktan sonra (npm i connect-flash) import edelim:

const flash = require('connect-flash');

Sonra server.js içerisinde session middleware’inden sonra bu paket için bir middleware oluşturalım:

app.use(flash());

Kullanıcı register formundayken kayıt olmak istediğinde olası validation hatalarını diziye çevirip, flash ile validation\_error nesnesi yardımı ile tuttuk, tuttuktan sonra formu tekrar göstermesi için kullanıcıyı register sayfasına redirect ettik, kullanıcı formu tekrar gördüğü zaman, flash ile tuttuğumuz hata listesini flash mesajı olarak bu sayede kullanıcıya yansıtabileceğiz. Flash mesajları kullanıcıya gösterildikten sonra session yapısında silinir.

flash’ı middleware olarak çalıştırdıktan sonraki aşamada flash’ın tutmuş olduğu validation\_error nesnesindeki hataları ekrana yansıtabilmek için response parametresinin locals alanına bu nesnseyi eklemek zorundayız, bunu da bir middleware ile sağlıyoruz:

app.use(flash()); // flash çalıştıktan sonra res.locals’e flash nesnesi atanır.

app.use((req, res, next) => {

    res.locals.validation\_error = req.flash('validation\_error');

next();

});

Mesajlarımızı bootstrap alert yapısı ile göstermek istediğimizde:

<% validation\_error.forEach(error => { %>

<div class="alert alert-danger alert-dismissible fade show" role="alert">

    <strong> <%- error.msg %> </strong>

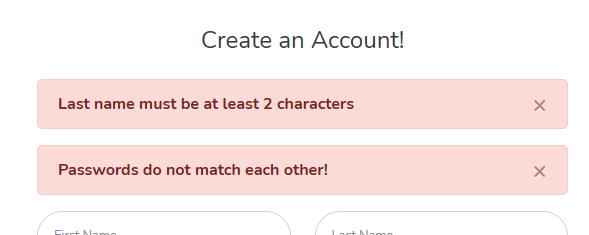
    <button type="button" class="close" data-dismiss="alert" aria-label="Close">

      <span aria-hidden="true">&times;</span>

    </button>

  </div>

<% }) %>



Register sayfasındaki validation hatalarında bu şekilde mesaj çıktısı alıyoruz.

Validation mesajlarını gördüğümüz anda input alanlarının silinmiş olduğunu gördük, bunun önüne geçmek için yine flash ile tutup bir sonraki request’te ekranda yansıtacağız. Flash middleware’ine gidip alanları response’taki locals alanına kaydediyoruz ki register.ejs sayfasında bu verileri çekebilelim.

const register = (req, res) => {

    const errors = validationResult(req);

    if (!errors.isEmpty()) {

        req.flash('validation\_error', errors.array());

        req.flash('firstName', req.body.firstName);

        req.flash('lastName', req.body.lastName);

        res.redirect('/register');

    }

};

Flash modülü ile request’in bosy’sindeki alanları anahtar kelimeler ile kaydettik. Sonra bunları flash middleware’inde kullanıyoruz:

app.use((req, res, next) => {

    res.locals.validation\_error = req.flash('validation\_error');

    res.locals.firstName = req.flash('firstName');

    res.locals.lastName = req.flash('lastName');

    next();

});

Böylece kullanıcı input değerlerini yanlış girip register butonuna bastıktan sonra validation mesajları aldığında input alanları boş olmayacak, bir önceki girdiği değerler yer alacak, eski değerleri kaybolmayacak.

Validation işlemleri bittiğine göre session bilgisini veritabanına kaydetme sırası geldi. Çünkü server’ı kapatıp açtığımızda tüm session’lar siliniyordu. Bunun önüne geçmek amacıyla session bilgisini mongodb’ye kaydetmek için connect-mongodb-session paketini kurmalıyız.

npm i connect-mongodb-session

Sonra server.js’te veritabanı bağlantılarının yapıldığı yerde bu pakete ait gerekli kodları yazalım:

const MongoDBStore = require('connect-mongodb-session')(session);

const sessionStore = new MongoDBStore(

{

    uri: process.env.MONGODB\_CONNECTION\_STRING,

    databaseName: 'connect\_mongodb\_session\_test',

    collection: ' allSessions'

});

Şimdi user model’imizi oluşturalım. Models dizini altında user\_model.js oluşturduk:

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const UserSchema = new Schema({ // aşağıda model bazlı validation sağladık

    firsName: { type: String, require: true, trim: true, minlength: 2, maxlength: 30 },

    lastName: { type: String, require: true, trim: true, minlength: 2, maxlength: 30 },

    email: { type: String, required: true, trim: true, unique: true, lowercase: true },

    password: { type: String, required: true, trim: true },

}, {collection: 'users', timestamp: true});

const User = mongoose.model('User', UserSchema);

module.exports = User;

Oluşturduğumuz User modeli auth\_controller’da import edelim:

const User = require('../models/user\_model');

auth\_controller’daki register metodunda request’in body’sinden gelen bilgileri, eğer validation hataları yoksa veritabanına kayıt edeceğiz. User’ın girdiği maile göre de mail bilgisi unique olduğundan dolayı db’de bu mail ile kayıtlı user var mı yok mu diye kontrol sağladık:

else {    // validation error(s) yoksa

        try {

            const \_user = await User.findOne({email: req.body.email});

            if (\_user) { // eğer böyle bir user bulunduysa

                req.flash('validation\_error', [{msg: "Someone else is already using this email!"}]);

            } else {    // veritabanında bu maili kullanan bir user yoksa

                const newUser = new User({

                    firstName: req.body.firstName, lastName: req.body.lastName,

                    email: req.body.email, password: req.body.password

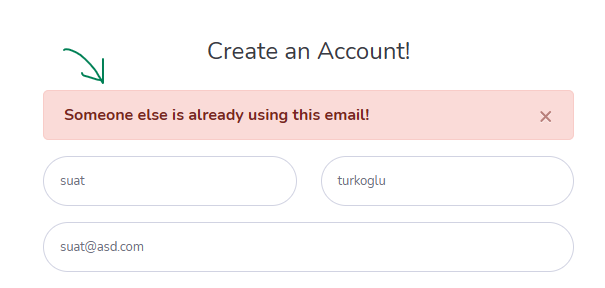
                });

                await newUser.save();   // yeni kullanıcıyı db'ye kaydet

            }

        } catch (error) { console.log(error) }

    }

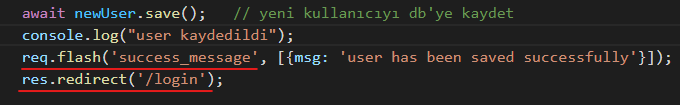


Daha önce veritabanında \_user nesnesinin tuttuğu email ile kayıtlı kullanıcı varsa soldaki gibi bir validation mesajı bastırmasını sağladık. Bunu da validation paketine ait msg nesnesi ile gönderdik.



Ve validation işlemleri tamamen doğru yapıldığında da newUser nesnesinin tutmuş olduğu kullanıcımız veritabanına kayıt edilmiş oldu.

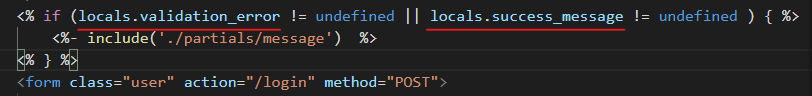
User veritabanına kaydedildikten sonra success mesajı ile login sayfasına yönlendirmemiz gerekiyor.

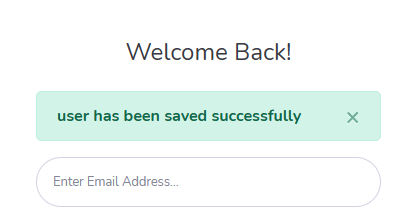


Flash’ta yeni bir nesne ile (success\_message) bu kayıt başarılı mesajını flash middleware’ine gönderelim ve orada response’un locals alanına bu nesneyi kaydedelim:

res.locals.success\_message = req.flash('success\_message');

Artık bu mesajı login.ejs dosyamızda kullanabiliriz:





Görüldüğü üzere register sayfasında başarılı şekilde kayıt olan kullanıcı, login sayfasına yönlendirildi ve başarı mesajı bastırıldı.

Kayıt olan kullanıcının bilgileri veritabanımıza kayıt edildi fakat authentication işlemleri hep passport js kullanılacak, projenin başında kurduğumuz paketleri artık kullanacağız.

passport\_local paketini kullanmaya başlayalım. config dizini altında passport\_local.js dosyası oluşturalım. Biz email ve şifre ile giriş yapmayı kullanacağız.

Passport.js’in dökümantasyonundan aldığımız yapıları kendi alanlarımıza uydurarak passport\_local.js dosyamıza yazdık ve auth\_controller’da login kısmında passport’u çağırıyoruz ki authentication sağlansın:

const login = (req, res, next) => {

    passport.authenticate('local', {

        successRedirect: '/',   // başarılı olursa yönlendir

        failureRedirect: '/login',  // hatalı olursa buraya yönlendir

        failureFlash: true          // mesajları aç

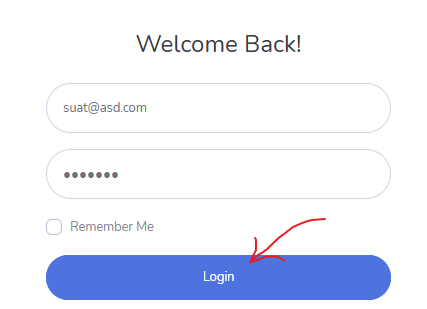
    })(req, res, next);

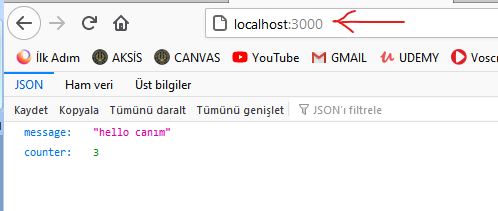
};

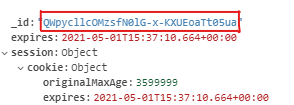
Passport yapısını projeye aktif edebilmek için bunu server.js’te middleware olarak çağırmalıyız. Tabi bunu flash ve session middleware’lerinden sonra eklemeliyiz:

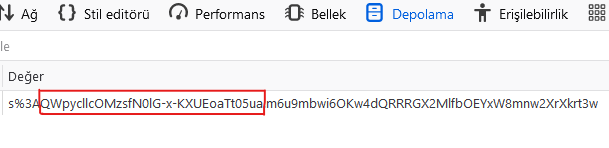
app.use(passport.initialize());

app.use(passport.session());



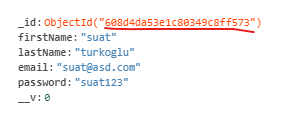
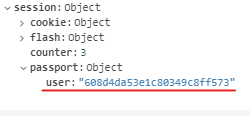


 Login sayfasında veritabanına kayıtlı bir user ile giriş yaptığımızda, user’ın artık ana sayfamıza yönlendirildiğini gördük.



Browser’daki session değeri ile veritabanındaki mySessions koleksiyonunda bulunan session’un id’sinin eşleştiğini gördük.

Aynı zamanda mySessions koleksiyonundaki oturum açan user’ın passport alanındaki user bilgisi ile users koleksiyonundaki giriş yaptığım user’ın id’si eşleşti:



User’ın id’sini session’a kaydetme işini passport’un serializeUser metodu halleder.

deserializeUser metodu da session’a kaydedilmiş olan user id’yi alır ve veritabanında aranır ve bulunur.

Bu noktada authentaication işlemleri tamamlandı. Kullanıcıya başarılı bir login işleminden sonra yönetici panelini göstermek için gerekli route işlemlerini yazdık. Öncelikle temamızın panel ile ilgili frontend kodlarını alıp views’teki layout kısmında yeni bir management\_layout adında bir ejs dosyası oluşturarak oraya yapıştırdık. Sonra management\_router adlı bir dosya oluşturup rotayı hazırladık:

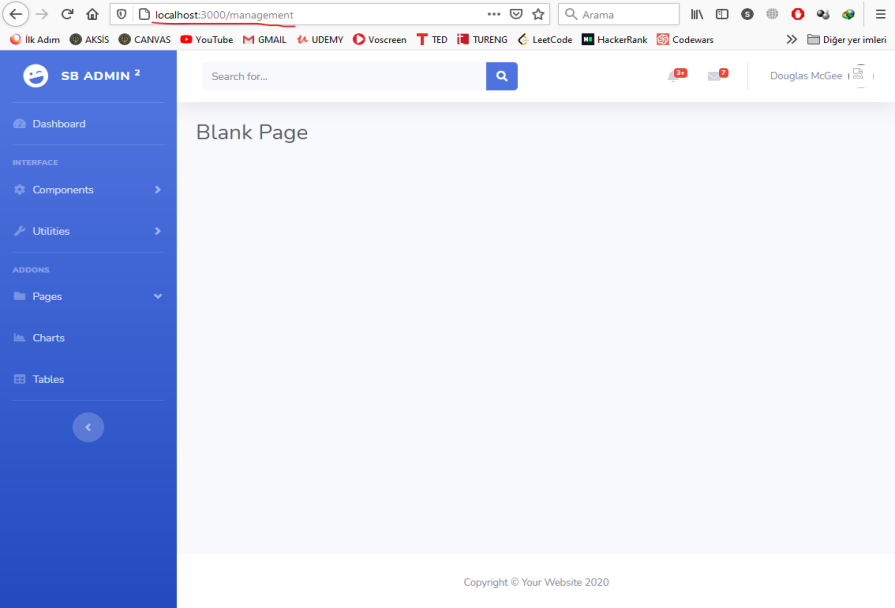
router.get('/', authMiddleware.notLoggedIn, managementController.showHomePage);

management\_controller dosyasını oluşturduktan sonra showHomePage metodunu yazdık:

const showHomePage = (req, res, next) => {

    res.render('index', { layout: './layout/management\_layout.ejs' });

};



Şimdi sisteme başarılı şekilde giriş yapan kullanıcıyı yönetici paneline yönlendirmek için Css ve js dosyaları için dosya yolunu doğru şekilde ayarladıktan sonra başarılı bir login işleminden sonra kullanıcıyı yönetim paneli karşıladı:

Kullanıcı sağ üstte yer alan logout’a basıp çıkış yapmak istediğinde session’un kapanıp cookie’larin temizlenmesi için auth\_controller’da logout metodunu yazdık:

const logout = (req, res, next) => {

    req.logout();   // db'deki session kısmındaki passport alanındaki id silinir.

    req.session.destroy((error) => {    // session'ı da silmeliyiz

        res.clearCookie('connect.sid'); // mevcut cookie'yi temizle

        res.redirect('/login');

    });

};

Sonra ilgili router’ı auth\_router’da yazdık:

router.get('/logout', authMiddleware.notLoggedIn, authController.logout);

const loggedInUser = (req, res, next) => {

    if (!req.isAuthenticated()) {

        return next();

    } else {

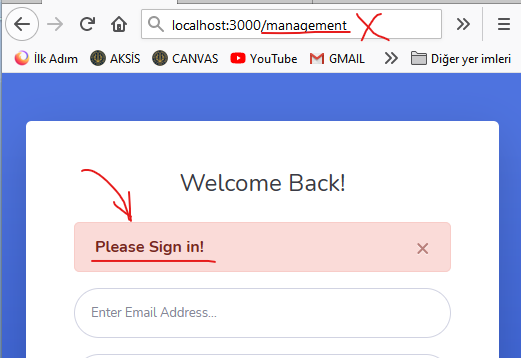
        res.redirect('/management');    // eğer oturum açılmışsa login veya register sayfalarına

    }                                   // gidilmek istendiğinde kullanıcı management sayfasına yönlensin

};

Bu metot sayesinde kullanıcı login olmuş halde iken /register ve /login rotalarına gitmek istediklerinde yine /management rotasına yönlenmesini sağladık, böylece register ve login sayfalarına erişmek istediğinde mutlaka logout ile çıkış yapmak zorunda kalacak.

Kullanıcı çıkış yaptıktan sonra /management rotasına erişmek istediğinde ise yönetim paneline asla erişemeyecek ve bir uyarı mesajı görecek:

 Bunu sağlayan metot:

const notLoggedIn = (req, res, next) => {

    if (req.isAuthenticated()) {

        return next();

    } else {

        req.flash('error', ['Please Sign in!']);

        res.redirect('/login');

    }

};