Create ASP.NET Core MVC React-Redux APPLIcation With Token Authentication (without typescript) (using VS Code)

This series of tutorial teaches you to create a new ASP.NET Core MVC web application with react-redux and token base authentication in Visual Studio Code and use it as a template or boilerplate for our future use.

# Contents

* Why not use already existing React-Redux Template provided by Microsoft?
* Prerequisites
* Create ASP.NET Core With React Project
* Setup Webpack

## Why not use already existing REACT-REDUX Template provided by Microsoft?

There is a built-in template for asp.net 2 application with react-redux. To create this project type, run dotnet new reactredux instead of dotnet new mvc in a terminal window. But there are few points that I was persuaded to create my own template.

* **Existing Template comes with lots of features that are already setup.**

When you create a new project using *donet new reactredux* command. It creates a new dotnet core project with react and redux added to it. It also setup babel and webpack for you and add some default libraries like bootstrap.js for you. All these things need setup and connections to work. If you use this template you may not understand how they really connected to each other and how they work. So, creating your own project will help you to understand how setup everything related to your project and also prevent unnecessary libraries to be included with your project.

* **It uses Typescript instead of JavaScript and I’m not fun of using typescript.**

As you know [TypeScript](http://www.typescriptlang.org/) is a superset of JavaScript which primarily provides optional static typing, classes and interfaces. One of the big benefits is to enable IDEs to provide a richer environment for spotting common errors *as you type the code*.

Actually I’m a C# developer and I really liked using typescript at first but there is few problem using it with react:

* Most of learning course of react and redux aren’t using typescript.
* Most of the time when you search in internet for an issue related to react you will find it in JavaScript.
* Most of open source packages and component that you may use are not using typescript.

After dealing with these problems I decided to not using typescript. But it is totally up on you to use it or not. In fact it brings lots of benefit with it but it also have lots of headaches. So I will not use it in this tutorial.

* **We mostly need authentication with our project.**

We mostly wrote asp.net core applications that requires individual authentication so it was good to have a template with authentication already setup. (Token base authentication with openIdConnect)

## Prerequisites

Before we get started, you’ll need to make sure you have the following installed pc (or mac).

* Node JS
* Visual Studio Code
* ASP.NET Core SDK
* Omni Sharp C# extension

If we have everything set up, let’s start by creating a new ASP.NET Core Web Application.

## Creating ASP.NET Core With React Project

We’re going to create a new project from command prompt using the command dotnet new mvc of ASP.NET Core CLI.

First open a Command Prompt window and navigate to desired folder then run below commands to create a new project.

mkdir MyApp

cd MyApp

dotnet new mvc

Run below command to open newly created project in Visual Studio Code.

Code .

If you already installed c# extensions add-on it will prompt you to create required assets for this project press yes.

To start working with npm packages for this project run below command in terminal

npm init

It will ask you few questions, you may leave them blank if you don’t know what to fill.

After finishing this command, **packages.json** file will be added to project’s root directory

### Setting up webpack and babel

Webpack is an open-source JavaScript module bundler. Its main purpose is to bundle JavaScript files for usage in a browser, yet it is also capable of transforming, bundling, or packaging just about any resource or asset. Webpack takes modules with dependencies and generates static assets representing those modules.

As we want to use react.js and it is a JavaScript library, we need webpack to bundle our JavaScript code for browsers.

If you didn’t already install webpack globally run below:

npm install webpack –g

Then run below to add webpack to your project:

npm i webpack –-save-dev

npm i webpack-cli --save-dev

open package.json file and add below

{

"name": "myapp",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

+ "build": "webpack",

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^4.20.2",

"webpack-cli": "^3.1.2"

}

}

We will write our react app with ES6. **ES6** refers to version 6 of the ECMA Script programming language. ECMA Script is the standardized name for JavaScript, and version 6 is the next version after version 5, which was released in 2011. Not all Browsers support es6, so we need a translator for this purpose.

**Babel** is a JavaScript compiler that includes the ability to compile JSX into regular JavaScript.

Run below commands in CMD to add required Babel libraries to our project:

npm install --save-dev babel-loader @babel/core

npm install @babel/preset-env --save-dev

npm install @babel/preset-react –-save-dev

Create .**babelrc** configuration file in root folder of project with below content.

{

"presets": ["@babel/preset-env", "@babel/preset-react"]

}

Now we should setup webpack to use babel as loader of JavaScript files.

Create webpack.config.js file in root directory of project and write below code in it.

const path = require('path');

module.exports = {

entry: "./src/index.js",

mode: "production", // "production" | "development" | "none"

output: {

path: path.resolve(\_\_dirname, "./wwwroot/dist"), // string

// the target directory for all output files

// must be an absolute path (use the Node.js path module)

filename: "bundle.js", // the filename template for entry chunks

publicPath: "dist/", // string

// the url to the output directory resolved relative to the HTML page

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

}

]

}

}

### Add ReactJs to Project

Run below command to add react to our project.

npm i react react-dom --save-dev

Create src folder in root directory of project then create src/index.js file.

import \* as ReactDOM from 'react-dom';

import \* as React from 'react';

ReactDOM.render(

<h1>Hello, world!</h1>,

document.getElementById('root')

);

Edit Views/Home/Index.cshtml file and remove extra content from it to be like below.

@{

ViewData["Title"] = "Home Page";

}

<div id="root">Loading...</div>

@section scripts {

<script src="~/dist/bundle.js"></script>

}

Now that we add react to our project we don’t need other razor views and controllers came with ASP.NET Core CLI Template. So let get rid of them first.

First Delete About.cshtml and Cantact.cshtml files from View folder. We only need Index.cshtml file. Then remove extra content from layout.cshtml file

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>@ViewData["Title"] - MyApp</title>

<base href="~/" />

</head>

<body>

@RenderBody()

@RenderSection("scripts", required: false)

</body>

</html>

Rename Models folder to ViewModels. Then open ErroViewModel file and change namespace of it.

Open ViewImports.cshtml and Change namespace there

@using MyApp.ViewModels

And also change using of this namespace in HomeController.cs file.

### Run Project

Before testing our application first run below command to create our bundle.js file

Webpack --config webpack.config.js

Then run

dotnet run

You should see “Hello World!” text in your browser.

Now try to change Hello World text and save the file and refresh your browser, but as you can see nothing happened. This happens because of that we didn’t run our webpack command to bundle our JavaScript again. There is a solution for this to instantly see changes as you save your file and we will explain it in next topic.

### Adding Hot Module Replacement to project

What we need is [Webpack middleware](https://github.com/aspnet/JavaScriptServices/tree/master/src/Microsoft.AspNetCore.SpaServices) and Hot Module replacement:

[**Webpack middleware**](https://github.com/aspnet/JavaScriptServices/tree/master/src/Microsoft.AspNetCore.SpaServices) so that, during development, any webpack-built resources will be generated on demand, without you having to run webpack manually or compile files to disk

[**Hot module replacement**](https://github.com/aspnet/JavaScriptServices/tree/master/src/Microsoft.AspNetCore.SpaServices) so that, during development, your code and markup changes will be pushed to your browser and updated in the running application automatically, without even needing to reload the page.

* **Adding webpack middleware**

Open Startup.cs file and add this using:

+ using Microsoft.AspNetCore.SpaServices.Webpack;

Go to configure method and add Webpack middleware to it as below:

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {

HotModuleReplacement = true,

ReactHotModuleReplacement = true

});

}

else

{

app.UseExceptionHandler("/Home/Error");

}

Open \_ViewImports.cshtml file and add below to it:

@addTagHelper \*, Microsoft.AspNetCore.SpaServices

* **Adding hot module replacement:**

We need two packages to enable hot module replacement, so run below command:

npm install -D aspnet-webpack aspnet-webpack-react

our webpack configuration file need some modifications:

* Change entry point from string of “./src/index.js” to an object with main property like this { main: "./src/index.js"}.
* Add react-hot-module/babel plugin to babel-loader

So open webpack.config.js file and change it as below:

const path = require('path');

module.exports = {

entry: { main: "./src/index.js"},

mode: "development", // "production" | "development" | "none"

output: {

path: path.resolve(\_\_dirname, "./wwwroot/dist"), // string

// the target directory for all output files

// must be an absolute path (use the Node.js path module)

filename: "bundle.js", // the filename template for entry chunks

publicPath: "dist/", // string

// the url to the output directory resolved relative to the HTML page

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader",

options: {

plugins: [

"react-hot-loader/babel"

]

}

}

}

]

}

}

Add the end open index.js and add this to end of the file.

// Allow Hot Module Replacement

if (module.hot) {

module.hot.accept();

}

Now run the project and try to change hello world! Text one more time and save the file. You should instantly see changes in your browser.

### Create Required React Components and navigate between them

Not that we initialized our ASP.NET Application, let create our react app to register and login users. For simplify reasons we will going to create 4 components. We are not going to create real implementation of them. We will just return simple text in render method of each component.

* App.js : our main component (Home Page)
* Login.js
* Signup.js
* Feature.js: our special component that should be shown only when user is logged in.

#### Adding simplified Version of Components

First create our “components” folder under “src” folder:

…\src>mkdir components

Then let’s create our first component named app.js:

…\src>cd components

…\src\components> echo.>App.js

Modify app.js file as below:

import React, { Component} from "react";

class App extends Component {

constructor(props) {

super(props);

}

render()

{

return (

<div>This is App.js</div>

);

}

}

export default App;

Then let’s create Feature component:

…\src\components> echo.>Feature.js

Modify Feature.js file as below:

import React, { Component} from "react";

class Feature extends Component {

constructor(props) {

super(props);

}

render()

{

return (

<div>This is Feature.js. Only authenticated Users allowed.</div>

);

}

}

export default Feature;

For better organizing our components we will create a sub-folder for related components.

Create a new folder named “auth”, under components folder and add “Signup.js” and ”Signin.js” files to it

…\src\components>mkdir auth

…\src\components\auth>echo.>Signin.js

…\src\components\auth>echo.>Signin.js

Create signin component as below:

import React, { Component } from 'react';

class Signin extends Component {

constructor(props) {

super(props);

}

render() {

return(

<div>This is signin page!</div>

)

}

}

export default Signin;

Next let’s create our signup component

import React, { Component } from 'react';

class Signup extends Component {

constructor(props) {

super(props);

}

render() {

return(

<div>This is signup page!</div>

)

}

}

export default Signup;

Add Navigation Between Components

Next step is to create a navigation between this 4 components

Add react router to project

npm install react-router-dom

modify index.js file to add the ability of navigating between components

import \* as ReactDOM from 'react-dom';

import \* as React from 'react';

import { BrowserRouter as Router, Route, Link,Switch } from "react-router-dom";

import App from './components/App';

import Signup from './components/auth/Signup';

import Signin from './components/auth/Signin';

import Feature from './components/Feature';

ReactDOM.render(

<Router>

<div>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/Signin">Signin</Link>

</li>

<li>

<Link to="/Signup">Signup</Link>

</li>

<li>

<Link to="/Feature">Feature</Link>

</li>

</ul>

<hr />

<Route exact path="/" component={App} />

<Route path="/Signin" component={Signin} />

<Route path="/Signup" component={Signup} />

<Route path="/Feature" component={Feature} />

</div>

</Router>

,

document.getElementById('root')

);

// Allow Hot Module Replacement

if (module.hot) {

module.hot.accept();

}

Run the application and if everything is set as said you should be able to navigate between these components using provided links inside home page. Now try to type desired page (Ex. http://localhost:5000/Signin) on browsers address bar. As you can see it will not load the page. So why we access our pages from links inside pages but we can access them when typing directly on browsers address bar?

### Server-side vs Client-side routing

Now that React-router is added to our project, we have 2 place where the URL is interpreted. The very first request will always be to the server. That will then return a page that contains the needed script tags to load React and React Router etc. Only when those scripts have loaded does phase 2 start.

So in our case when we type directly on browsers address bar no react-router is running yet. So it will make a server request and that is where our problem starts.

To fix this we should add single page application fallback rout to our ASP.NET application routes. Open startup.cs file and modify below section in Configure method:

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

routes.MapSpaFallbackRoute(

name: "spa-fallback",

defaults: new { controller = "Home", action = "Index" });

});

Run the application once again and now if we type our desired page like <http://localhost:5000/Signin> on browser’s address bar, we will navigate to sign in page.

# Setup Authentincation in ASP.NET Core Web Application

If you are already added entity framework to project you can skip “Enable entity framework and add dbcontext to project” section.

## Enable entity framework and add dbcontext to project

Entity Framework (EF) Core is a lightweight, extensible version of Entity framework data access technology. To add EF Core to project, we should install Microsoft.EntityFrameworkCore package from nuget package manager. But if we are already added Microsoft.AspNetCore.All package (check our “.csproj” file) as below, there is no need to install EF Core separately.

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />

</ItemGroup>

Otherwise run below to install EF Core:

dotnet add package Microsoft.EntityFrameworkCore --version 2.2.0

dotnet restore

Let create our first entity to keep users information named “ApplicationUser”. It is “Best Practice” to create our entities under Core\Models folder.

…> mkdir Core

…> cd Core

…/Core> mkdir Models

…/Core> cd Models

…/Core/Models> echo.> ApplicationUser.cs

Open “ApplicationUser.cs” file and modify it as below:

using Microsoft.AspNetCore.Identity;

namespace MyApp.Core.Models

{

public class ApplicationUser : IdentityUser

{

}

}

It is “Best Practice” to create “Dbcontext” in separate folder from Models so let’s create “Persistence” folder and add “ApplicationDbContext.cs” file to it:

…>mkdir Persistence

…>cd Persistence

../Persistence> echo.> ApplicationDbContext.cs

Modify “ApplicationDbContext.cs” file as below:

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using MyApp.Core.Models;

namespace MyApp.Persistence

{

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>

{

public ApplicationDbContext(DbContextOptions options)

: base(options)

{

}

}

}

Open appsettings.json file and add ConnectionStrings to it (I have SQL Server express localdb 2014 installed on my pc, feel free to change connection sting if you have another version of SQL Server installed)

{

"ConnectionStrings": {

"Default": "server=(LocalDb)\\MSSQLLocalDB; database=MyAppDbContext; Integrated Security=SSPI"

},

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Warning"

}

}

}

Tobe able to run EF Core Commands we need one more package named “Entity Framework Core .NET Command Line Tools”. Open “.csproj” file and add below to project node:

<ItemGroup>

<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1" />

</ItemGroup>

Run below commands:

dotnet restore

Now that EF Core is installed, let Configure EF Core in “ConfigureServices” of “Startup.cs”. so open startup file and add below code to it:

using MyApp.Persistence;

using Microsoft.EntityFrameworkCore;

.

.

.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddDbContext<ApplicationDbContext>(o =>

{

o.UseSqlServer(Configuration.GetConnectionString("Default"));

});

}

Now we should create our first migration

dotnet ef migrations add init

If you check you could see migration folder and its files in your root directory. To update database run below command:

dotnet ef database update

## Add OpenIddict to project

Open MyApp.csproj and add below to it then restore dotnet project

<ItemGroup>

<PackageReference Include="OpenIddict" Version="2.0.0-\*" />

<PackageReference Include="OpenIddict.EntityFrameworkCore" Version="2.0.0-\*" />

<PackageReference Include="OpenIddict.Mvc" Version="2.0.0-\*" />

</ItemGroup>

Then run below commad:

dotnet restore

## Create authentication Controllers

create “AuthController.cs” file under Controllers folder as below

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Threading.Tasks;

using AspNet.Security.OpenIdConnect.Extensions;

using AspNet.Security.OpenIdConnect.Primitives;

using AspNet.Security.OpenIdConnect.Server;

using Microsoft.AspNetCore.Authentication;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Options;

using MyApp.Core.Models;

using OpenIddict.Abstractions;

using OpenIddict.Mvc.Internal;

namespace MyApp.Controllers

{

public class AuthController : Controller

{

private readonly IOptions<IdentityOptions> \_identityOptions;

private readonly SignInManager<ApplicationUser> \_signInManager;

private readonly UserManager<ApplicationUser> \_userManager;

public AuthController(

IOptions<IdentityOptions> identityOptions,

SignInManager<ApplicationUser> signInManager,

UserManager<ApplicationUser> userManager)

{

\_identityOptions = identityOptions;

\_signInManager = signInManager;

\_userManager = userManager;

}

[HttpPost("~/connect/token"), Produces("application/json")]

public async Task<IActionResult> Exchange([ModelBinder(typeof(OpenIddictMvcBinder))] OpenIdConnectRequest request)

{

Debug.Assert(request.IsTokenRequest(),

"The OpenIddict binder for ASP.NET Core MVC is not registered. " +

"Make sure services.AddOpenIddict().AddMvcBinders() is correctly called.");

if (request.IsPasswordGrantType())

{

var user = await \_userManager.FindByNameAsync(request.Username);

if (user == null)

{

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.InvalidGrant,

ErrorDescription = "The username/password couple is invalid."

});

}

// Ensure the user is allowed to sign in.

if (!await \_signInManager.CanSignInAsync(user))

{

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.InvalidGrant,

ErrorDescription = "The specified user is not allowed to sign in."

});

}

// Reject the token request if two-factor authentication has been enabled by the user.

if (\_userManager.SupportsUserTwoFactor && await \_userManager.GetTwoFactorEnabledAsync(user))

{

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.InvalidGrant,

ErrorDescription = "The specified user is not allowed to sign in."

});

}

// Ensure the user is not already locked out.

if (\_userManager.SupportsUserLockout && await \_userManager.IsLockedOutAsync(user))

{

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.InvalidGrant,

ErrorDescription = "The username/password couple is invalid."

});

}

// Ensure the password is valid.

if (!await \_userManager.CheckPasswordAsync(user, request.Password))

{

if (\_userManager.SupportsUserLockout)

{

await \_userManager.AccessFailedAsync(user);

}

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.InvalidGrant,

ErrorDescription = "The username/password couple is invalid."

});

}

if (\_userManager.SupportsUserLockout)

{

await \_userManager.ResetAccessFailedCountAsync(user);

}

// Create a new authentication ticket.

var ticket = await CreateTicketAsync(request, user);

var result = SignIn(ticket.Principal, ticket.Properties, ticket.AuthenticationScheme);

return result;

}

else if (request.IsRefreshTokenGrantType())

{

// Retrieve the claims principal stored in the refresh token.

var info = await HttpContext.AuthenticateAsync(OpenIdConnectServerDefaults.AuthenticationScheme);

// Retrieve the user profile corresponding to the refresh token.

// Note: if you want to automatically invalidate the refresh token

// when the user password/roles change, use the following line instead:

// var user = \_signInManager.ValidateSecurityStampAsync(info.Principal);

var user = await \_userManager.GetUserAsync(info.Principal);

if (user == null)

{

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.InvalidGrant,

ErrorDescription = "The refresh token is no longer valid."

});

}

// Ensure the user is still allowed to sign in.

if (!await \_signInManager.CanSignInAsync(user))

{

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.InvalidGrant,

ErrorDescription = "The user is no longer allowed to sign in."

});

}

// Create a new authentication ticket, but reuse the properties stored

// in the refresh token, including the scopes originally granted.

var ticket = await CreateTicketAsync(request, user, info.Properties);

return SignIn(ticket.Principal, ticket.Properties, ticket.AuthenticationScheme);

}

return BadRequest(new OpenIdConnectResponse

{

Error = OpenIdConnectConstants.Errors.UnsupportedGrantType,

ErrorDescription = "The specified grant type is not supported."

});

}

private async Task<AuthenticationTicket> CreateTicketAsync(OpenIdConnectRequest request,

ApplicationUser user,

AuthenticationProperties properties = null)

{

// Create a new ClaimsPrincipal containing the claims that

// will be used to create an id\_token, a token or a code.

var principal = await \_signInManager.CreateUserPrincipalAsync(user);

// Create a new authentication ticket holding the user identity.

var ticket = new AuthenticationTicket(principal,

properties,

OpenIdConnectServerDefaults.AuthenticationScheme);

if (!request.IsRefreshTokenGrantType())

{

// Set the list of scopes granted to the client application.

ticket.SetScopes(new[]

{

OpenIdConnectConstants.Scopes.OpenId,

OpenIdConnectConstants.Scopes.Email,

OpenIdConnectConstants.Scopes.Profile,

OpenIdConnectConstants.Scopes.OfflineAccess,

OpenIddictConstants.Scopes.Roles

}.Intersect(request.GetScopes()));

}

ticket.SetResources("http://localhost:5000");

// Note: by default, claims are NOT automatically included in the access and identity tokens.

// To allow OpenIddict to serialize them, you must attach them a destination, that specifies

// whether they should be included in access tokens, in identity tokens or in both.

// principal.Claims.Append(new Claim ("test", "narbe", OpenIdConnectConstants.Destinations.AccessToken));

foreach (var claim in principal.Claims)

{

// Never include the security stamp in the access and identity tokens, as it's a secret value.

if (claim.Type == \_identityOptions.Value.ClaimsIdentity.SecurityStampClaimType)

{

continue;

}

var destinations = new List<string>

{

OpenIdConnectConstants.Destinations.AccessToken

};

// Only add the iterated claim to the id\_token if the corresponding scope was granted to the client application.

// The other claims will only be added to the access\_token, which is encrypted when using the default format.

if ((claim.Type == OpenIdConnectConstants.Claims.Name && ticket.HasScope(OpenIdConnectConstants.Scopes.Profile)) ||

(claim.Type == OpenIdConnectConstants.Claims.Email && ticket.HasScope(OpenIdConnectConstants.Scopes.Email)) ||

(claim.Type == OpenIdConnectConstants.Claims.Role && ticket.HasScope(OpenIddictConstants.Claims.Roles)))

{

destinations.Add(OpenIdConnectConstants.Destinations.IdentityToken);

}

claim.SetDestinations(destinations);

}

return ticket;

}

}

}

Create RegisterViewModel:

…> cd ViewModels

…/ViewModels> echo.> RegisterViewModel.cs

Open “RegistorViewModel.cs” file and modify it as below:

using System.ComponentModel.DataAnnotations;

namespace MyApp.ViewModels

{

public class RegisterViewModel

{

[Required]

[EmailAddress]

[Display(Name = "Email")]

public string Email { get; set; }

[Required]

[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]

[DataType(DataType.Password)]

[Display(Name = "Password")]

public string Password { get; set; }

[Required]

[StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]

[DataType(DataType.Password)]

[Display(Name = "ConfirmPassword")]

public string ConfirmPassword { get; set; }

}

}

Create AccountController.cs as below:

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Options;

using MyApp.Core.Models;

using MyApp.Persistence;

using MyApp.ViewModels;

namespace MyApp.Controllers

{

public class AccountController : Controller

{

private readonly UserManager<ApplicationUser> \_userManager;

private readonly SignInManager<ApplicationUser> \_signInManager;

private readonly IOptions<IdentityOptions> \_identityOptions;

private readonly ApplicationDbContext \_applicationDbContext;

private static bool \_databaseChecked;

public AccountController(

UserManager<ApplicationUser> userManager,

IOptions<IdentityOptions> identityOptions,

SignInManager<ApplicationUser> signInManager,

ApplicationDbContext applicationDbContext

)

{

\_userManager = userManager;

\_identityOptions = identityOptions;

\_signInManager = signInManager;

\_applicationDbContext = applicationDbContext;

}

[AllowAnonymous]

[HttpPost("~/api/auth/register")]

public async Task<IActionResult> Register([FromBody] RegisterViewModel model)

{

EnsureDatabaseCreated(\_applicationDbContext);

if (ModelState.IsValid)

{

if(model.Password != model.ConfirmPassword)

{

return BadRequest(new { general = new[] {"Password and Confirm Password not match"} });

}

var user = new ApplicationUser

{

UserName = model.Email,

Email = model.Email,

FullName = model.FullName,

};

var result = await \_userManager.CreateAsync(user, model.Password);

if (result.Succeeded)

{

// we may need to send confirmation email after resigter succeed

// Send an email with this link

//var code = await \_userManager.GenerateEmailConfirmationTokenAsync(user);

//var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = code }, protocol: Context.Request.Scheme);

//await \_emailSender.SendEmailAsync(model.Email, "Confirm your account",

// "Please confirm your account by clicking this link: <a href=\"" + callbackUrl + "\">link</a>");

//===========================================================================

//but we do nothing now just return ok

return Ok();

}

else

{

return BadRequest(new { general = result.Errors.Select(x => x.Description).ToArray() });

}

}

else

{

return BadRequest(new { general = ModelState.SelectMany(x => x.Value.Errors)

.Select(x => x.ErrorMessage).ToArray() });

}

// If we got this far, something failed, redisplay form

//return BadRequest(new { general = "Bad Request".ToArray() });

}

// The following code creates the database and schema if they don't exist.

// This is a temporary workaround since deploying database through EF migrations is

// not yet supported in this release.

// Please see this http://go.microsoft.com/fwlink/?LinkID=615859 for more information on how to do deploy the database

// when publishing your application.

private static void EnsureDatabaseCreated(ApplicationDbContext context)

{

if (!\_databaseChecked)

{

\_databaseChecked = true;

context.Database.EnsureCreated();

}

}

}

}

Now we have our API endpoints for register and login users. Next step is to configure services to enable authentication. Open Startup.cs file and change *ConfigureServices* method as below:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddDbContext<ApplicationDbContext>(o =>

{

o.UseSqlServer(Configuration.GetConnectionString("Default"));

o.UseOpenIddict();

});

services.AddIdentity<ApplicationUser, IdentityRole>(

options => {

options.Password.RequiredLength = 4;

options.Password.RequireUppercase = false;

options.Password.RequireDigit = false;

options.Password.RequireLowercase =false;

options.Password.RequireNonAlphanumeric = false;

options.Password.RequiredUniqueChars = 0;

}

)

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

services.Configure<IdentityOptions>(options =>

{

options.ClaimsIdentity.UserNameClaimType = OpenIdConnectConstants.Claims.Name;

options.ClaimsIdentity.UserIdClaimType = OpenIdConnectConstants.Claims.Subject;

options.ClaimsIdentity.RoleClaimType = OpenIdConnectConstants.Claims.Role;

});

// Register the OpenIddict services.

// Note: use the generic overload if you need

// to replace the default OpenIddict entities.

services.AddOpenIddict()

// Register the OpenIddict core services.

.AddCore(options =>

{

//options.

// Register the Entity Framework stores and models.

options.UseEntityFrameworkCore()

.UseDbContext<ApplicationDbContext>();

})

.AddServer

(options =>

{

options.UseMvc();

// Enable the token endpoint (required to use the password flow).

options.EnableTokenEndpoint("/connect/token");

// Allow client applications to use the grant\_type=password flow.

options.AllowPasswordFlow()

.AllowRefreshTokenFlow();

// Accept token requests that don't specify a client\_id.

options.AcceptAnonymousClients();

// During development, you can disable the HTTPS requirement.

options.DisableHttpsRequirement();

options.RegisterScopes(

OpenIdConnectConstants.Scopes.OpenId,

OpenIdConnectConstants.Scopes.Profile,

OpenIdConnectConstants.Scopes.Email

);

// Note: to use JWT access tokens instead of the default

// encrypted format, the following lines are required:

//

// options.UseJsonWebTokens();

// options.AddEphemeralSigningKey();

});

services.AddAuthentication()

.AddOAuthValidation();

}

Add below add *app.UserAuthentication()* to *Configure* method.

*Note: you must add it before app.UseMvc()*

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {

HotModuleReplacement = true,

ReactHotModuleReplacement = true

});

}

else

{

app.UseExceptionHandler("/Home/Error");

}

app.UseStaticFiles();

app.UseAuthentication();

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

}

Don’t foget to add required namespaces to startup.cs file

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.SpaServices.Webpack;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using MyApp.Persistence;

using Microsoft.EntityFrameworkCore;

using MyApp.Core.Models;

using Microsoft.AspNetCore.Identity;

using AspNet.Security.OpenIdConnect.Primitives;

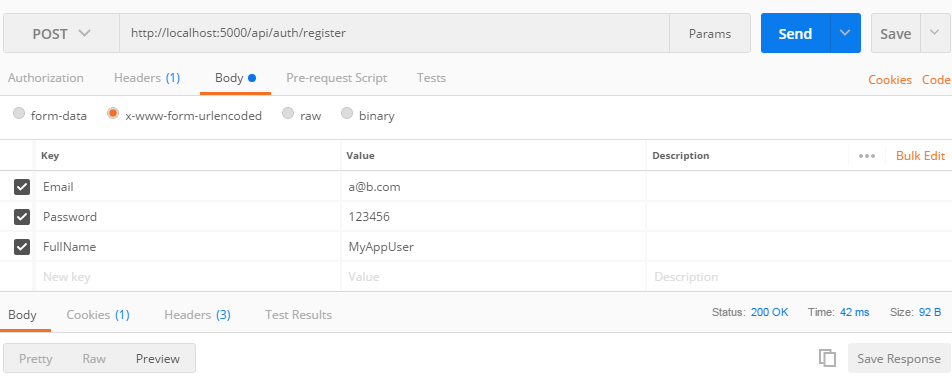
lets test our app,

first run the application

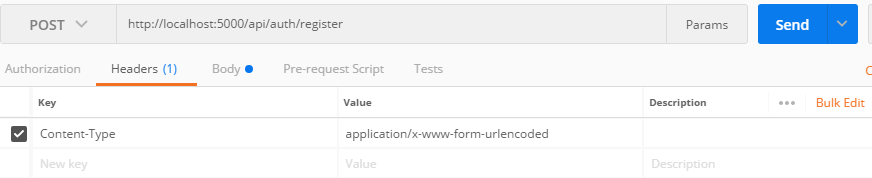
dotnet run

the open postman and create new post request to <http://localhost:5000/api/auth/register>

and send Email, Password, FullName in body of the form.



Don’t remember to set content-type of the request to application/x-www-form-urlencoded



If everything goes fine you should see status 200 OK after sending the request.

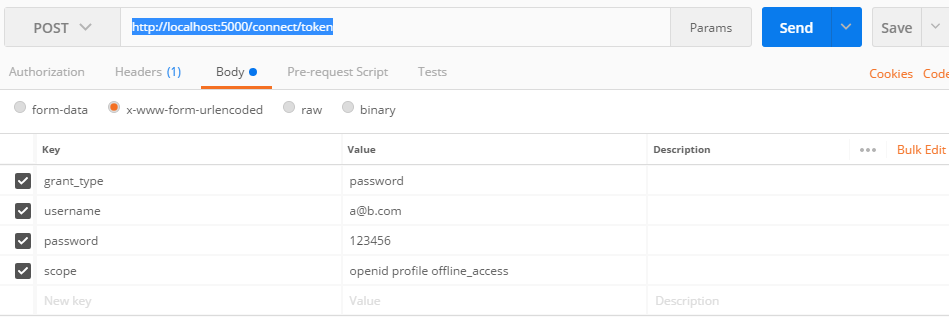
If we try to login user with our api, it will fail with error that says: Invalid object name 'OpenIddictAuthorizations'.

So Before we check login, we forgot to update our database to for openiddict tables. So let create a new migration and update the database.

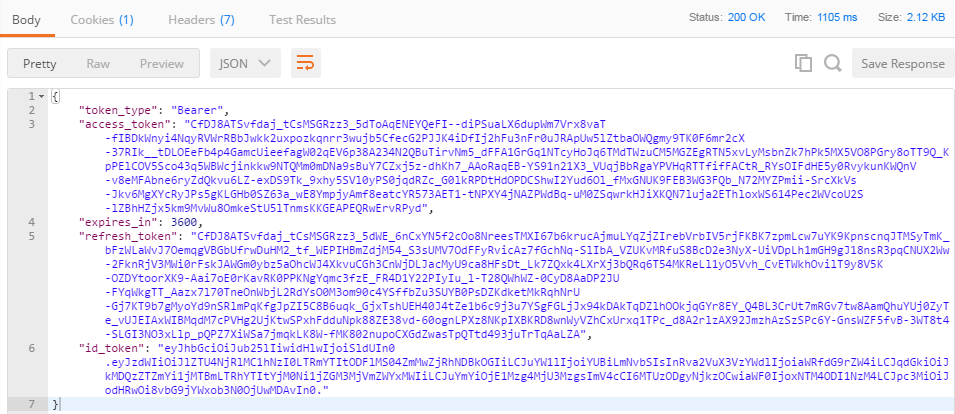
dotnet ef migrations add addOpenIddictTables

dotnet ef database update

Now we are good to go. So create new request in postman with below parameters and send it to <http://localhost:5000/connect/token>



After sending the request, you should see the response as below:



# Create React-Redux Application

## Add Redux to React Application

**Redux** is a predictable state container for JavaScript apps. From the very beginning, we need to stress that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

**Redux** is **used** mostly for application state management. To summarize it, **Redux** maintains the state of an entire application in a single immutable state tree (object), which can't be changed directly. When something changes, a new object is created (using actions and reducers).

Let add required packages to our project

npm install redux --save

npm install react-redux --save

npm install redux-thunk --save

Next create primary folders for a react redux application

cd src

mkdir actions

mkdir reducers

Next we should create our Combine reducers function. Combine reducers takes a hash of reducers and returns a single reducer. The resulting reducer calls every child reducer, and gathers their results into a single state object. **The state produced by** combineReducers() **namespaces the states of each reducer under their keys as passed to** combineReducers()

We normally put our combineReducers() function in index.js file under reducers folder.

cd reducers

echo.>Index.js

our combineReduce() function is empty now but we will add our reducers in our next steps.

import { combineReducers } from 'redux';

import { reducer as form } from 'redux-form';

const rootReducer = combineReducers({

form,

});

export default rootReducer;

now that we setup our redux, let add it to our application’s entry point. So open Index.js file and modify it as below:

import \* as ReactDOM from 'react-dom';

import \* as React from 'react';

import { BrowserRouter as Router, Route, Link,Switch } from "react-router-dom";

import { createStore, applyMiddleware } from 'redux';

import { Provider } from 'react-redux';

import reduxThunk from 'redux-thunk';

import App from './components/App';

import Signup from './components/auth/Signup';

import Signin from './components/auth/Signin';

import Feature from './components/Feature';

import reducers from './reducers/index';

const createStoreWithMiddleware = applyMiddleware(reduxThunk)(createStore);

ReactDOM.render(

<Provider store={createStoreWithMiddleware(reducers)}>

<Router>

<div>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/Signin">Signin</Link>

</li>

<li>

<Link to="/Signup">Signup</Link>

</li>

<li>

<Link to="/Feature">Feature</Link>

</li>

</ul>

<hr />

<Route exact path="/" component={App} />

<Route path="/Signin" component={Signin} />

<Route path="/Signup" component={Signup} />

<Route path="/Feature" component={Feature} />

</div>

</Router>

</Provider>

,

document.getElementById('root')

);

// Allow Hot Module Replacement

if (module.hot) {

module.hot.accept();

}

Now if you check your application it must run as previously.

## Add Signup to project

for adding signup to our react-redux application, we need to create 3 files

* Signup Component (We already create it but it only display a text instead of signup form)
* Signup Action creators
* Signup Reducer

Let first create our action types and action creator:

Create a new file named Types.js under actions folder.

cd actions

echo.>Types.js

we need 3 action types for register succeed and register failed.

export const REGISTER\_USER\_START = 'register\_user\_start';

export const REGISTER\_USER\_SUCCESS = 'register\_user\_success';

export const REGISTER\_USER\_FAIL = 'register\_user\_fail';

next create AuthActions.js file

cd actions

echo.>AuthActions.js

Our first action is registerUser. This action will call our dotnet application’s Register API we already wrote in previous section, so we need to install axios (or other package) to be able to call this API.

npm install axios

next let write our register action

import {

REGISTER\_USER\_START,

REGISTER\_USER\_FAIL,

REGISTER\_USER\_SUCCESS,

} from './Types';

import axios from 'axios';

const AUTH\_SERVER\_URL = '';

export const registerUser = ({email, password, confirmPassword}, callback) => {

return async (dispatch) => {

dispatch({

type: REGISTER\_USER\_START,

})

var params = {

email, password,confirmPassword

};

try {

const response = await axios.post(`${AUTH\_SERVER\_URL}/api/auth/register`, params)

dispatch({

type: REGISTER\_USER\_SUCCESS,

})

if(callback)

callback();

} catch (error) {

let errorMessages = [];

if(error.response.data.general) {

Array.prototype.push.apply(errorMessages, error.response.data.general);

}

else {

errorMessages.push('Registration Failed. Something went wrong!')

}

dispatch({

type: REGISTER\_USER\_FAIL,

payload: errorMessages

});

}

}

}

next create our authentication reducer

cd reducers

echo.>AuthReducer.js

There is nothing specific we need to do in our reducer for now. Our Authentication State needs only one property to show registration error. So we only prepare our “AuthReducer” to receive required action types for future enhancements.

import {

REGISTER\_USER\_START,

REGISTER\_USER\_FAIL,

REGISTER\_USER\_SUCCESS

} from "../actions/Types";

const INITIAL\_STATE = {

register\_error: '',

isLoading: false

}

export default(state = INITIAL\_STATE, action) => {

switch(action.type) {

case REGISTER\_USER\_START:

return { ...state, isLoading: true, register\_error: ''};

case REGISTER\_USER\_SUCCESS:

return { ...state, isLoading: false, register\_error: ''};

case REGISTER\_USER\_FAIL:

return { ...state, isLoading: false, register\_error: action.payload};

default: return state;

}

}

We should add this reducer to combineReducers function so open reducers\index.js file and modify it as below.

import { combineReducers } from 'redux';

import { reducer as form } from 'redux-form';

import AuthReducer from './AuthReducer';

const rootReducer = combineReducers({

auth: AuthReducer

});

export default rootReducer;

For creating our signup form we should install redux-form package to our project.

npm install redux-form

we also need to add redux-form reducer to combineReducers function

import { combineReducers } from 'redux';

import { reducer as form } from 'redux-form';

import AuthReducer from './AuthReducer';

const rootReducer = combineReducers({

form,

auth: AuthReducer,

});

export default rootReducer;

## Create Signup Component

Let first create our component using redux-form and register new user on form submit. After completing submit we will navigate user to login page. To do this we will add “withRouter” to our component. With adding it, you will be able to programmatically navigate to another page after submitting the form.

import React, { Component } from 'react';

import { Field, reduxForm } from 'redux-form';

import { withRouter } from "react-router-dom";

import { connect } from 'react-redux';

import { registerUser } from "../../actions/AuthActions";

class Signup extends Component {

constructor(props) {

super(props);

}

handleFormSubmit(data) {

this.props.registerUser(data, this.onSubmitComplete.bind(this));

}

onSubmitComplete() {

console.log('register succeed');

this.props.history.push("/signin");

}

render() {

const { handleSubmit } = this.props;

console.log(this.props.auth.isLoading);

return (

<form onSubmit={handleSubmit(this.handleFormSubmit.bind(this))}>

<h1>Sign up</h1>

<br/>

<div>

<label>email</label>

<div>

<Field

name="email"

component="input"

type="email"

placeholder="Email"

/>

</div>

</div>

<div>

<label>Password</label>

<div>

<Field

name="password"

component="input"

type="password"

placeholder="Password"

/>

</div>

</div>

<div>

<label>Confirm Password</label>

<div>

<Field

name="confirmPassword"

component="input"

type="password"

placeholder="Confirm Password"

/>

</div>

</div>

<div>

<span style={{color: 'red'}}>

{this.props.auth.register\_error}

</span>

</div>

<div>

<button type="submit" disabled={this.props.auth.isLoading ? 'disabled' : ''}>

Signup

</button>

</div>

</form>

)

}

}

Signup = reduxForm({

form: 'signup' // a unique identifier for this form

})(Signup);

function mapStateToProp(state) {

return {

auth: state.auth

};

}

Signup = connect(mapStateToProp, { registerUser })(withRouter(Signup));

export default Signup;

Now run the application, navigate to signup page and try to submit it. Below error will occur :

Uncaught ReferenceError: regeneratorRuntime is not defined

This error occurs because we wrote async actions and babel does know how to translate it. To solve this problem we should add below packages:

npm install @babel/plugin-transform-runtime --save-dev

npm install --save @babel/runtime

then open .babelrc file and add this package as a plugin :

{

"presets": ["@babel/preset-env", "@babel/preset-react"],

"plugins": [

["@babel/plugin-transform-runtime"]

]

}

For more information about this babel plugin check <https://babeljs.io/docs/en/babel-plugin-transform-runtime>.

Now run the application once more, you should be able to submit new user and check email and password in console of browser.

## Add Signin to Project

we need 3 action types for login succeed and login failed and fetch token.

export const LOGIN\_USER\_START = 'login\_user\_start';

export const LOGIN\_USER\_SUCCESS = 'login\_user\_success';

export const LOGIN\_USER\_FAIL = 'login\_user\_fail';

export const FETCH\_TOKEN = 'fetch\_token';

add this types to authactions

import {

REGISTER\_USER\_START,

REGISTER\_USER\_FAIL,

REGISTER\_USER\_SUCCESS,

LOGIN\_USER\_START,

LOGIN\_USER\_SUCCESS,

LOGIN\_USER\_FAIL,

FETCH\_TOKEN,

} from './Types';

Our next action is loginUser. This action will call /connect/token API that we already wrote in previous section.

…/src> npm i jwt-decode

Import it to authactions

import jwt\_decode from 'jwt-decode';

create “loginuser” action :

export const loginUser = ({email, password}, callback) => {

return async (dispatch) => {

dispatch({

type: LOGIN\_USER\_START,

})

//OpenIddict post parameters should be string as below (Json not allowed)

var params = `username=${email}&password=${password}&grant\_type=password&scope=openid offline\_access profile`;

try {

const response = await axios.post(`${AUTH\_SERVER\_URL}/connect/token`

, encodeURI(params), header );

//Save Tokens local storage.

localStorage.setItem('accessToken', response.data.access\_token);

localStorage.setItem('refreshToken', response.data.refresh\_token);

localStorage.setItem('idToken', response.data.id\_token);

const now = new Date();

let expirationDate = new Date(now.getTime() + response.data.expires\_in \* 1000)

.getTime().toString();

localStorage.setItem('expiresAt', expirationDate);

dispatch({

type: LOGIN\_USER\_SUCCESS,

});

dispatch(

{

type: FETCH\_TOKEN,

payload: getUser(response.data.id\_token)

});

if(callback)

callback();

} catch (arg) {

let errorMessage = "Something went wrong";

if(arg.response && arg.response.data) {

errorMessage = arg.response.data.error\_description;

}

dispatch({

type: LOGIN\_USER\_FAIL,

payload: errorMessage

});

}

}

}

const getUser = (idToken) => {

let user = {

username:'\*\*\*'

};

if (idToken) {

var decoded = jwt\_decode(idToken);

user.username = decoded.name;

}

return user;

}

Next step is add sign in to “authreducer”. First we should update our initial state by adding few properties.

1. User : to keep login user’s info
2. isLogin: to determine if user is logged in or not (actually it is 3 state option. Null value means that we are checking login state.
3. Login\_error: to display error message of login form.

After preparing initial state, we should create our action creators as below:

import {

REGISTER\_USER\_FAIL,

REGISTER\_USER\_SUCCESS,

LOGIN\_USER\_START,

FETCH\_TOKEN,

LOGIN\_USER\_FAIL,

LOGIN\_USER\_SUCCESS

} from "../actions/Types";

const INITIAL\_STATE = {

register\_error: '',

user: null,

isLogin: null,

login\_error: ''

}

export default(state = INITIAL\_STATE, action) => {

switch(action.type) {

case REGISTER\_USER\_START:

return { ...state, isLoading: true, register\_error: ''};

case REGISTER\_USER\_SUCCESS:

return { ...state, isLoading: false, register\_error: ''};

case REGISTER\_USER\_FAIL:

return { ...state, isLoading: false, register\_error: action.payload};

case LOGIN\_USER\_START:

return { ...state, isLoading: true, login\_error: ''};

case FETCH\_TOKEN:

return { ...state,

isLogin: true,

user: action.payload

};

case LOGIN\_USER\_SUCCESS:

return { ...state, isLoading: false, isLogin:true, login\_error: ''};

case LOGIN\_USER\_FAIL:

return { ...state, login\_error:action.payload , isLoading: false, isLogin: false, password:'' }

default: return state;

}

}

## Create Signin Component

Let first create our component using redux-form and try to log form data when user click on submit button.

import React, { Component } from 'react';

import { Field, reduxForm } from 'redux-form';

import { withRouter } from "react-router-dom";

import { connect } from 'react-redux';

import { loginUser } from "../../actions/AuthActions";

class Signin extends Component {

constructor(props) {

super(props);

}

handleFormSubmit(data) {

this.props.loginUser(data, this.onSigninComplete.bind(this));

}

onSigninComplete() {

console.log('sigin succeed');

this.props.history.push("/");

}

render() {

const { handleSubmit } = this.props

return (

<form onSubmit={handleSubmit(this.handleFormSubmit.bind(this))}>

<div>

<label>email</label>

<div>

<Field

name="email"

component="input"

type="email"

placeholder="Email"

/>

</div>

</div>

<div>

<label>Password</label>

<div>

<Field

name="password"

component="input"

type="password"

placeholder="Password"

/>

</div>

</div>

<div>

<span style={{color: 'red'}}>

{this.props.auth.login\_error}

</span>

</div>

<div>

<button type="submit">

Signin

</button>

</div>

</form>

)

}

}

Signin = reduxForm({

form: 'Signin' // a unique identifier for this form

})(Signin);

function mapStateToProp(state) {

return {

auth: state.auth

};

}

Signin = connect(mapStateToProp, { loginUser })(withRouter(Signin));

export default Signin;

Now run your application and try to register a user and then login in with it.

Next we will try to restrict unauthorized users to access feature page. For this purpose we will create a base component to load authentication needed components in it. This component is going to check if user is logged in (isLogin = true), and then display its child component to user. Otherwise (isLogin = false) it will redirect user to login page. We also going to handle delay that will occur when our application is checking the authentication state (isLogin = null).

lets create “RequiredAuth” component:

…> cd src/components

…/src/components> echo.> RequiredAuth.js

And modify it as below:

import React, { Component } from 'react';

import { connect } from 'react-redux';

import PropTypes from 'prop-types';

import { withRouter } from "react-router-dom";

export default function(ComposedComponent) {

class Authentication extends Component {

static contextTypes = {

router: PropTypes.object

}

componentWillMount() {

const {isLogin} = this.props.auth;

if (isLogin != null && !isLogin) {

this.props.history.push('/signin');

}

}

componentWillReceiveProps(nextProps) {

if (nextProps.auth.isLogin === false) {

this.props.history.push('/signin');

}

}

render() {

const {isLogin} = this.props.auth;

return isLogin != null ? (<ComposedComponent {...this.props} />)

: (<div style={{

display: 'flex',

flex: 1,

height: '100%',

flexDirection:'column',

alignItems:'center',

justifyContent:'center'

}}>

<h3 style={{color:'rgba(0, 0, 0, 0.54)', marginTop:20}}>Authorizing...</h3>

</div>)

}

}

function mapStateToProps(state) {

return { auth: state.auth };

}

return withRouter(connect(mapStateToProps)(Authentication));

}

Next we should add this component to index.js file and change Feature route as below:

import RequiredAuth from './components/RequiredAuth';

.

.

.

<Route path="/Feature" component={RequiredAuth(Feature)} />

.

.

Now if we run the project, below error will be shown:

… Support for the experimental syntax 'classProperties' isn't currently enabled.

Add @babel/plugin-proposal-class-properties (https://git.io/vb4SL) to the 'plugins' section of your Babel config to enable transformation.

To fix this let first add @babel/plugin-proposal-class-properties package to project:

npm i @babel/plugin-proposal-class-properties

then modify .babelrc file as below :

{

"presets": [["es2015", { "modules": false }], "react", "stage-2"],

"plugins": [

"@babel/plugin-proposal-class-properties"

]

}

Now run the project once again and try to navigate to feature page, you should see “Authorizing…” message. Go to login page and try to login with your user then navigate to Feature one more time, now you will see “This is Feature.js. Only authenticated Users allowed.” Message.

Just one thing left to do, we should automatically authenticate users that already get the token. For this purpose first we need an authentication action to load things related to authentication. So open authAction.js file and add below:

export function loadThings() {

return (dispatch, getState) => {

const { auth, isLoading } = getState();

if(auth.isLogin == null) {

dispatch({ type: LOAD\_FETCHING, fetching: true })

let accessToken = localStorage.getItem('accessToken');

let expiresAt = localStorage.getItem('expiresAt');

if (!isExpired(expiresAt)) {

let idToken = localStorage.getItem('idToken');

dispatch(

{

type: FETCH\_TOKEN, payload: getUser(idToken)

}

)

}

else {

refreshToken(dispatch);

}

}

};

}

const refreshToken = async (dispatch) => {

let refreshToken = localStorage.getItem('refreshToken');

if(refreshToken) {

var params = `refresh\_token=${refreshToken}&grant\_type=refresh\_token&scope=openid offline\_access fullname profile`;

try {

let response = await axios.post(`${AUTH\_SERVER\_URL}connect/token`, encodeURI(params), header);

localStorage.setItem('accessToken', response.data.access\_token);

const now = new Date();

let expirationDate = new Date(now.getTime() + response.data.expires\_in \* 1000).getTime().toString();

localStorage.setItem('refreshToken', response.data.refresh\_token);

localStorage.setItem('expiresAt', expirationDate);

localStorage.setItem('idToken', response.data.id\_token);

dispatch(

{

type: FETCH\_TOKEN,

payload: user: getUser(response.data.id\_token)

}

)

} catch (error) {

dispatch({

type: LOGIN\_USER\_FAIL

})

}

}

else {

dispatch({

type: LOGIN\_USER\_FAIL

})

}

}

const isExpired = (expiresAt) => {

if (Date.now() >= expiresAt) {

return true;

}

else {

return false;

}

}

Need a component to check authentication token before other components are shown to users. First create startup component:

…> cd src/components

…/src/components> echo.> Startup.js

Modify this file as below:

import React, { Component } from 'react';

import { connect } from 'react-redux';

import { loadThings } from './actions/AuthActions';

import { withRouter } from "react-router-dom";

function mapStateToProps(state) {

return {

};

}

class Startup extends Component {

componentDidMount() {

this.props.loadThings();

}

render() {

return (

<div>

{this.props.children}

</div>

);

}

}

export default withRouter(connect(

mapStateToProps, {loadThings}

)(Startup));

Now add this component in index.js file

import Startup from './Startup';

…

<Startup>

<div>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/Signin">Signin</Link>

</li>

<li>

<Link to="/Signup">Signup</Link>

</li>

<li>

<Link to="/Feature">Feature</Link>

</li>

</ul>

<hr />

<Route exact path="/" component={App} />

<Route path="/Signin" component={Signin} />

<Route path="/Signup" component={Signup} />

<Route path="/Feature" component={RequiredAuth(Feature)} />

</div>

</Startup>

If we run the application, it will check the token at startup and if no token found, application will redirect us to login page. Otherwise if it found a valid token, we will be navigated to home page.