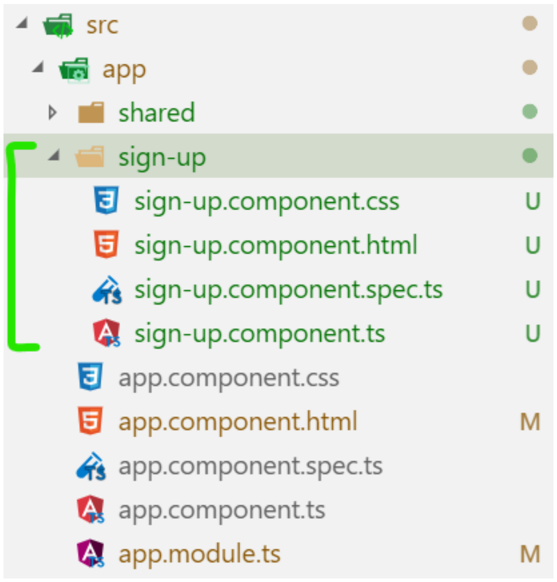
- Create a model class namely, user, which will be used during registration.

```
// path: /src/app/shared/user.model.ts
export class User {
    userName: string;
    password: string;
    email: string;
    firstName: string;
    lastName: string;
}
```

- Create sign-up component.



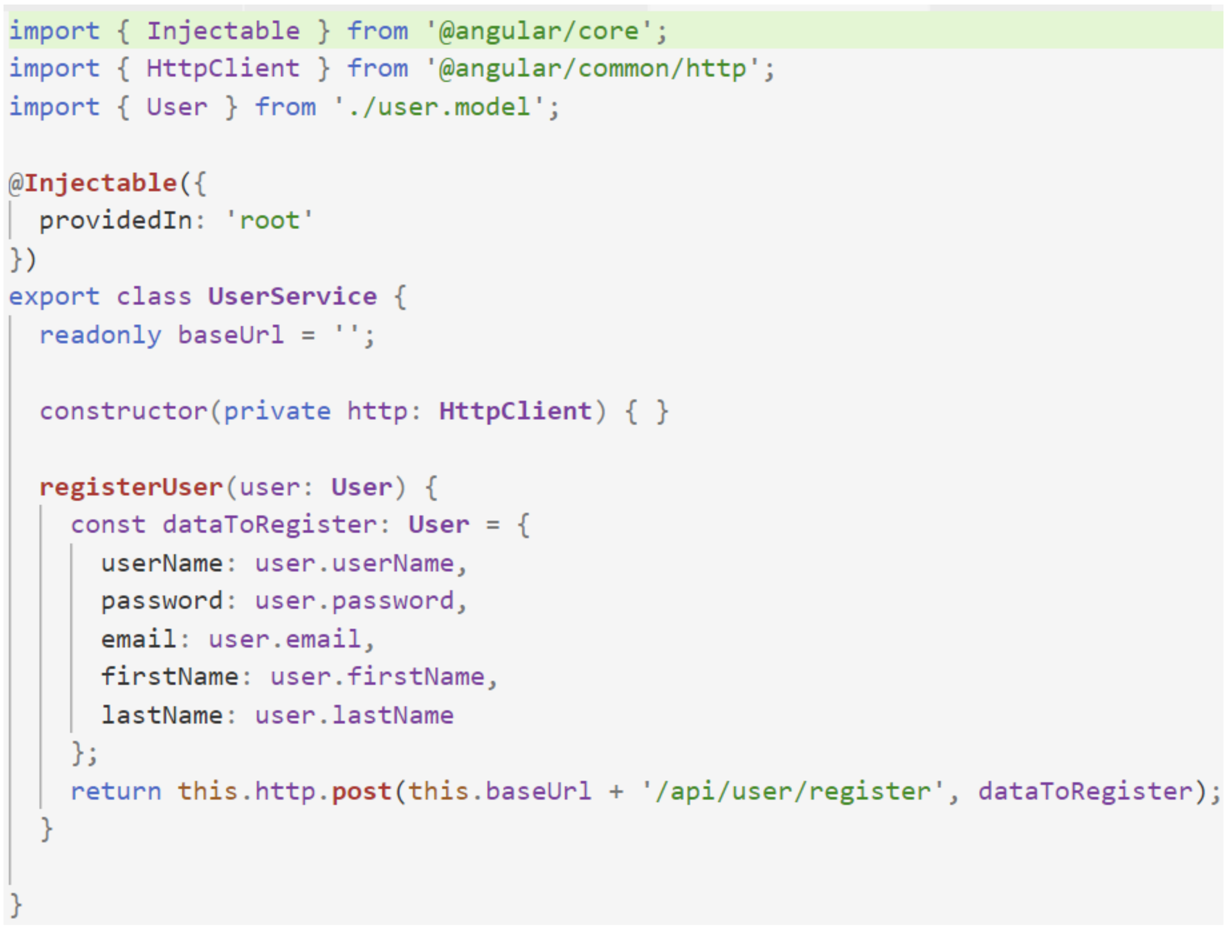
```
src
  app
    shared
    sign-up
      sign-up.component.css        U
      sign-up.component.html       U
      sign-up.component.spec.ts    U
      sign-up.component.ts         U
    app.component.css
    app.component.html             M
    app.component.spec.ts
    app.component.ts
    app.module.ts                  M
```

- Update default component html i.e. app.component.html.

```
<div class="container">
  <app-sign-up></app-sign-up>
</div>
```

app-sign-up tag will be replaced by the sign-up.component.html content.

- Inside service “UserService” define the registerUser method.



```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { User } from './user.model';

@Injectable({
  providedIn: 'root'
})
export class UserService {
  readonly baseUrl = '';

  constructor(private http: HttpClient) { }

  registerUser(user: User) {
    const dataToRegister: User = {
      userName: user.userName,
      password: user.password,
      email: user.email,
      firstName: user.firstName,
      lastName: user.lastName
    };
    return this.http.post(this.baseUrl + '/api/user/register', dataToRegister);
  }

}
```

- Install toastr.
  - npm install ngx-toastr --save
- Write below code in sign-up.component.ts file, to set up basic registration process.

```
import { Component, OnInit } from '@angular/core';
import { User } from '../shared/user.model';
import { UserService } from '../shared/user.service';
import { ToastrService } from 'ngx-toastr';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-sign-up',
  templateUrl: './sign-up.component.html',
  styleUrls: ['./sign-up.component.css']
})
export class SignUpComponent implements OnInit {
  user: User;
  emailPattern = '^[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$';

  constructor(private userService: UserService, private toastr:
ToastrService) { }
```

```
  ngOnInit() {
    this.resetForm();
  }

  resetForm(form?: NgForm) {
    if (form != null) {
      form.reset();
    }
    this.user = {
      userName: '',
      password: '',
      email: '',
      firstName: '',
      lastName: ''
    };
  }

  registerUser(form: NgForm) {
    this.userService.registerUser(form.value)
      .subscribe((data: any) => {
        if (data.Succeeded) {
          this.resetForm(form);
          this.toastr.success('Registered successfully!!');
        } else {
          this.toastr.error(data.Errors[0]);
        }
      });
  }

}
```

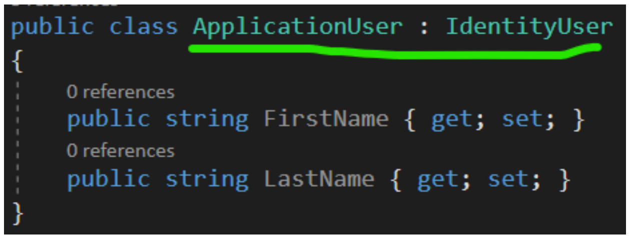
- Prepare the sign-up.component.html file to hold registration form.

```
<div class="row">
  <div class="col s8 offset-2">
    <div class="card">

      <div class="sm-jumbotron center-align">
        <h2>User Registration</h2>
      </div>

      <form class="col s12 white" #registrationForm="ngForm"
(ngSubmit)="registerUser(registrationForm)">
        <div class="row">
          <div class="input-field col s6">
```

```
            <input type="text" class="validate" name="userName"
#userName="ngModel" [(ngModel)]="user.userName"
              required>
            <label data-error="Required field!">UserName</label>
          </div>
          <div class="input-field col s6">
            <input class="validate" type="password" name="password"
#password="ngModel" [(ngModel)]="user.password"
              required minlength="3">
            <label
              [attr.data-
error]="password.errors!=null?(password.errors.required?'Required
field!':'Minimum 3 characters needed'):''">Password</label>
          </div>

          <div class="row">
            <div class="input-field col s12">
              <input class="validate" type="text" name="email"
#email="ngModel" [(ngModel)]="user.email"
                [pattern]="emailPattern">
              <label data-error="Invalid email!">Email</label>
            </div>
          </div>

          <div class="row">
            <div class="input-field col s6">
              <input type="text" name="firstName" #firstName="ngModel"
[(ngModel)]="user.firstName">
              <label>First Name</label>
            </div>
            <div class="input-field col s6">
              <input type="text" name="lastName" #lastName="ngModel"
[(ngModel)]="user.lastName">
              <label>Last Name</label>
            </div>
          </div>

          <div class="row">
            <div class="input-field col s12">
              <button [disabled]="!registrationForm.valid" class="btn-
large btn-submit"
                type="submit">Submit</button>
            </div>
          </div>
        </div>
```

```
      </form>

    </div>
  </div>
</div>
```

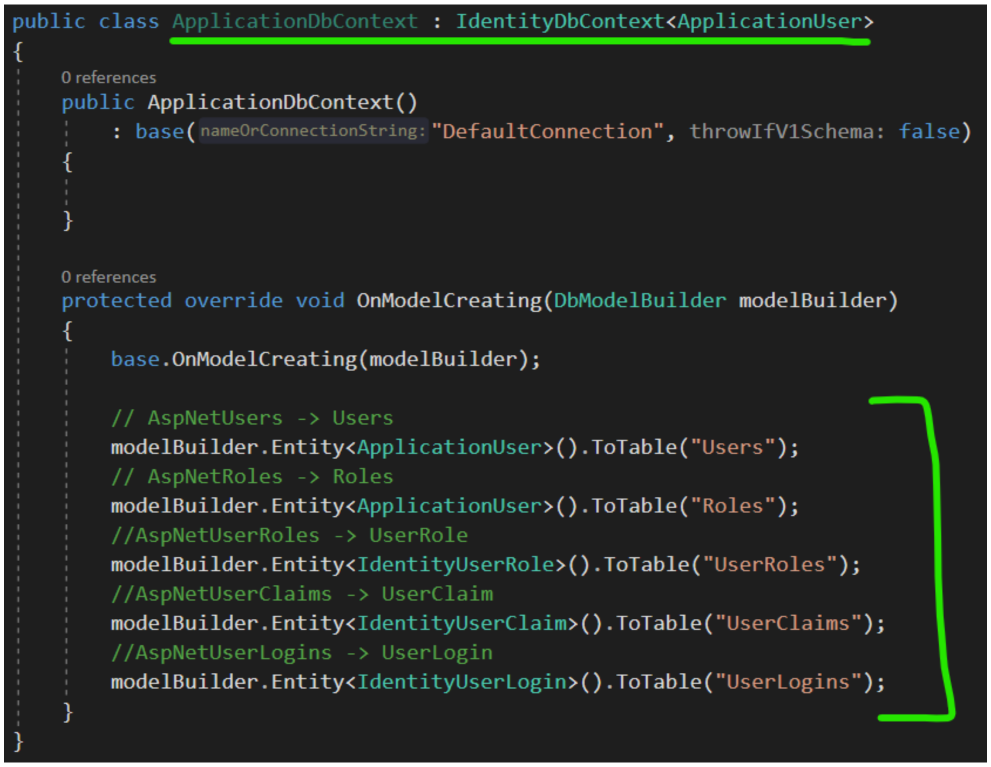
- Add ASP.NET Identity to ASP.NET Application
  - ASP.NET Identity is a membership system for ASP.NET Applications.
  - It provide lots of features like, User Management, Role Management, User Authentication and Authorization, social logins etc.
  - Install Microsoft.ASPNET.Identity.EntityFramework.
  - As ASP.NET Identity used Code First approach, so table structure inside the membership system is already defined. But, we will customize the same.



```
public class ApplicationUser : IdentityUser
{
    0 references
    public string FirstName { get; set; }
    0 references
    public string LastName { get; set; }
}
```

In IdentityUser class, ASP.NET Identity defines a set of default properties including Username, PasswordHash, and Email etc.

ApplicationUser class is inheriting form IdentityUser hence has access to default properties provided by IdentityUser class. Hence we add two extra properties of FirstName and LastName.

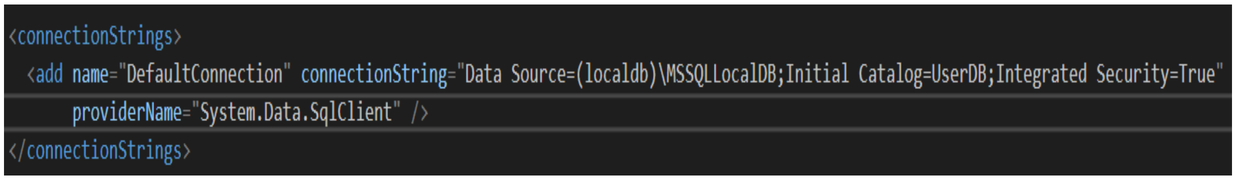


```csharp
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    0 references
    public ApplicationDbContext()
        : base(nameOrConnectionString: "DefaultConnection", throwIfV1Schema: false)
    {

    }

    0 references
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // AspNetUsers -> Users
        modelBuilder.Entity<ApplicationUser>().ToTable("Users");
        // AspNetRoles -> Roles
        modelBuilder.Entity<ApplicationUser>().ToTable("Roles");
        //AspNetUserRoles -> UserRole
        modelBuilder.Entity<IdentityUserRole>().ToTable("UserRoles");
        //AspNetUserClaims -> UserClaim
        modelBuilder.Entity<IdentityUserClaim>().ToTable("UserClaims");
        //AspNetUserLogins -> UserLogin
        modelBuilder.Entity<IdentityUserLogin>().ToTable("UserLogins");
    }
}
```

Here we are overriding the default names of ASP.NET Identity tables with our custom tables.

- Now we need to add and run migration for the above created custom classes after adding connection string to web.config file.

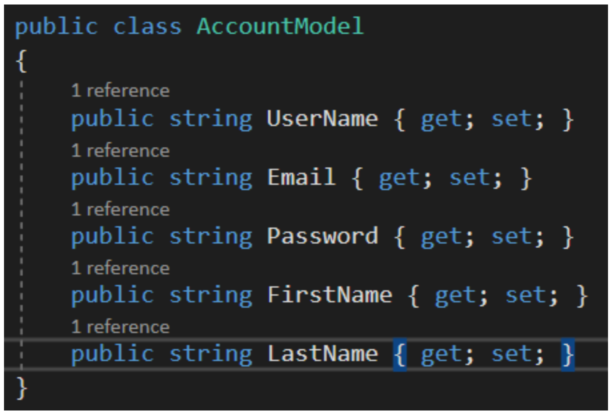
  Web.config



```xml
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=UserDB;Integrated Security=True"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

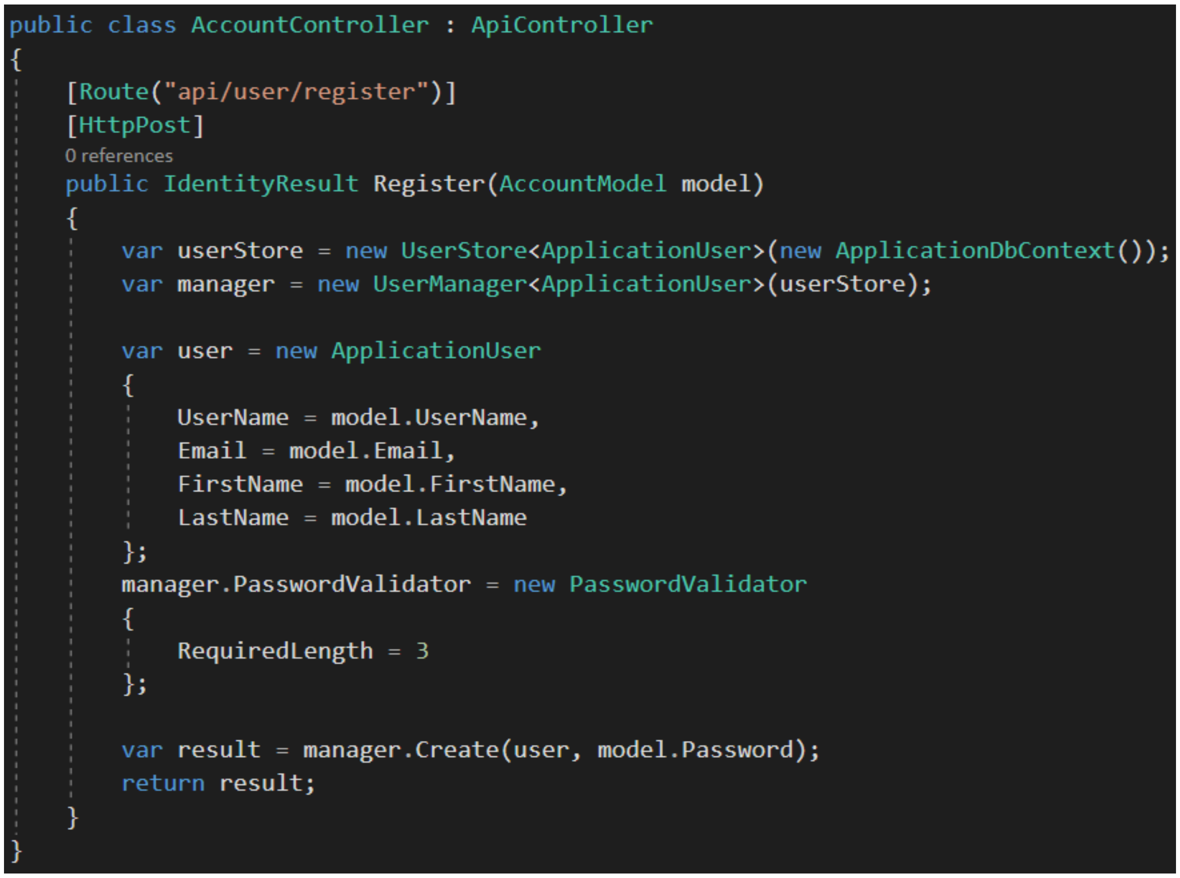
  - ✓ enable-migrations
  - ✓ add-migrations <migration_name>
  - ✓ update-database

- Add an Account model.



```
public class AccountModel
{
    1 reference
    public string UserName { get; set; }
    1 reference
    public string Email { get; set; }
    1 reference
    public string Password { get; set; }
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }
}
```

- Add Web API Controller for User Registration.



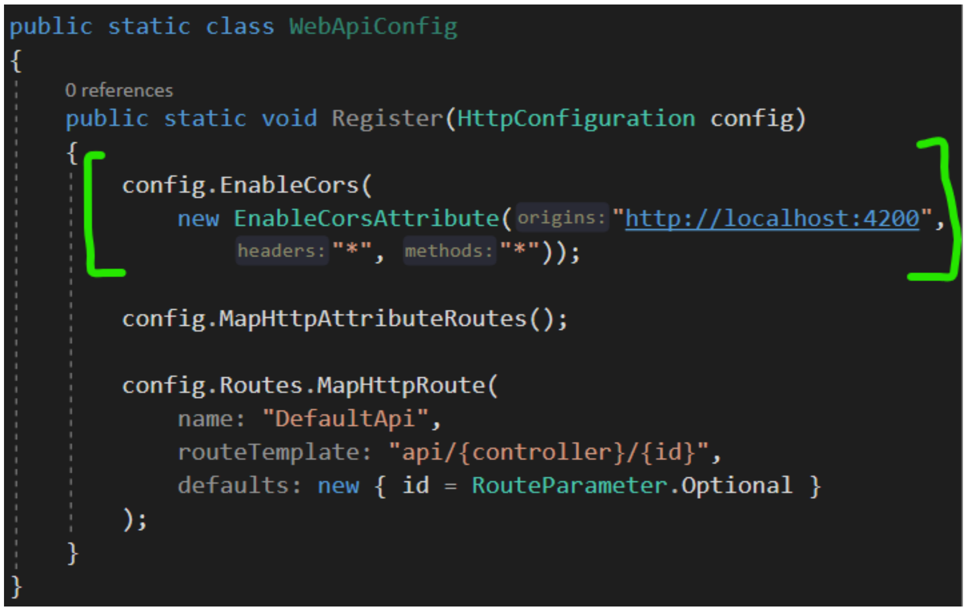
```
public class AccountController : ApiController
{
    [Route("api/user/register")]
    [HttpPost]
    0 references
    public IdentityResult Register(AccountModel model)
    {
        var userStore = new UserStore<ApplicationUser>(new ApplicationDbContext());
        var manager = new UserManager<ApplicationUser>(userStore);

        var user = new ApplicationUser
        {
            UserName = model.UserName,
            Email = model.Email,
            FirstName = model.FirstName,
            LastName = model.LastName
        };
        manager.PasswordValidator = new PasswordValidator
        {
            RequiredLength = 3
        };

        var result = manager.Create(user, model.Password);
        return result;
    }
}
```

UserStore is something which actually interacts with DB and stores and fetches the User details.

UserManager needs user store to be created and just manages the user store.

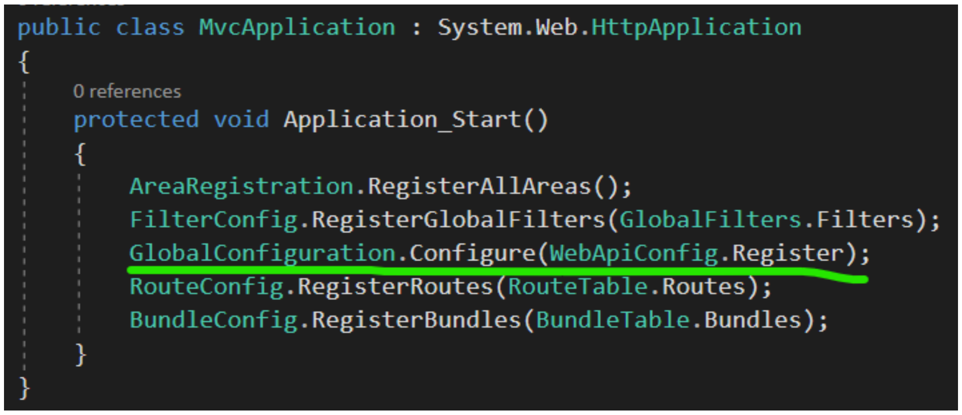
By default, ASP.NET Identity requires Password to be at least 6 characters long. Here, we have reduced the same to 3, as we have set minimum password length in Angular code as 3.

- **CORS:** Cross-Origin Resource Sharing
  - ✓ Install-Package Microsoft.AspNet.WebApi.Cors
  - ✓ Below is how we can use the mentioned package in our application.



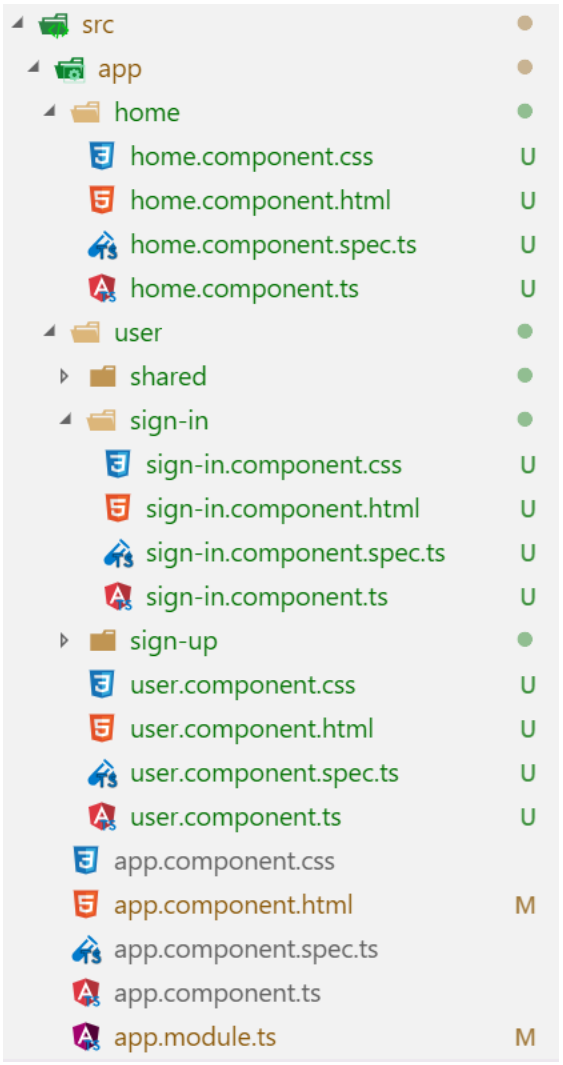
```
public static class WebApiConfig
{
    0 references
    public static void Register(HttpConfiguration config)
    {
        config.EnableCors(
            new EnableCorsAttribute(origins: "http://localhost:4200",
                headers: "*", methods: "*"));

        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

  - ✓ Apart from this, for enabling Attribute routing we need to add **config.MapHttpAttributeRoutes()** as shown in above image below CORS.
  - ✓ To register Web API routing we need to add below code in Global.asax file.



```
public class MvcApplication : System.Web.HttpApplication
{
    0 references
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        GlobalConfiguration.Configure(WebApiConfig.Register);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

- Login and Logout with Web API using Token Based Authentication
  - Create home, user component.
  - Move the already existing shared and sign-up folder under user folder; and create sign-in component under user folder.
    (Application's folder structure is depicted below)



```
src
  app
    home
      home.component.css          U
      home.component.html         U
      home.component.spec.ts      U
      home.component.ts           U
    user
      shared
      sign-in
        sign-in.component.css     U
        sign-in.component.html    U
        sign-in.component.spec.ts U
        sign-in.component.ts      U
      sign-up
      user.component.css          U
      user.component.html         U
      user.component.spec.ts      U
      user.component.ts           U
    app.component.css
    app.component.html            M
    app.component.spec.ts
    app.component.ts
    app.module.ts                 M
```

- Update the app.component.html as below.

```
<router-outlet></router-outlet>
```

- Make changes to sign-up.component.html, i.e. remove the card div and just keep the form.

```
<form class="col s12 white" #registrationForm="ngForm"
(ngSubmit)="registerUser(registrationForm)">
  <div class="row">
    <div class="input-field col s6">
      <input type="text" class="validate" name="userName"
#userName="ngModel" [(ngModel)]="user.userName" required>
      <label data-error="Required field!">UserName</label>
    </div>
    <div class="input-field col s6">
      <input class="validate" type="password" name="password"
#password="ngModel" [(ngModel)]="user.password" required
        minlength="3">
      <label
        [attr.data-
error]="password.errors!=null?(password.errors.required?'Required
field!':'Minimum 3 characters needed'):''">Password</label>
    </div>

    <div class="row">
      <div class="input-field col s12">
        <input class="validate" type="text" name="email"
#email="ngModel" [(ngModel)]="user.email"
          [pattern]="emailPattern">
        <label data-error="Invalid email!">Email</label>
      </div>
    </div>

    <div class="row">
      <div class="input-field col s6">
        <input type="text" name="firstName" #firstName="ngModel"
[(ngModel)]="user.firstName">
        <label>First Name</label>
      </div>
      <div class="input-field col s6">
        <input type="text" name="lastName" #lastName="ngModel"
[(ngModel)]="user.lastName">
        <label>Last Name</label>
      </div>
    </div>

    <div class="row">
      <div class="input-field col s12">
        <button [disabled]="!registrationForm.valid" class="btn-
large btn-submit" type="submit">Submit</button>
      </div>
    </div>
  </div>
</form>
```

- Complete user.component.html file.

```
<div class="container">
  <div class="row">
    <div class="col s8 offset-s2">
      <div class="card grey lighten-2">

        <div class="card-tabs">
          <ul class="tabs tabs-fixed-width tabs-transparent">
```

```
            <li class="tab">
              <a [routerLink]="['/login']"
routerLinkActive="active">Sign In</a>
            </li>
            <li class="tab">
              <a [routerLink]="['/signup']"
routerLinkActive="active">Sign Up</a>
            </li>
          </ul>
        </div>

        <div class="card-content white">
          <div class="row">
            <router-outlet></router-outlet>
          </div>
        </div>

      </div>
    </div>
  </div>
</div>
```

- Create app.routes.ts file and write code as mentioned.

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { UserComponent } from './user/user.component';
import { SignUpComponent } from './user/sign-up/sign-up.component';
import { SignInComponent } from './user/sign-in/sign-in.component';

export const routes: Routes = [
    {
        path: 'home',
        component: HomeComponent
    },
    {
        path: 'signup',
        component: UserComponent,
        children: [
            {
                path: '',
                component: SignUpComponent
            }
        ]
    },
    {
        path: 'login',
        component: UserComponent,
        children: [
            {
                path: '',
                component: SignInComponent
            }
        ]
    },
    {
        path: '',
        redirectTo: '/login',
        pathMatch: 'full'
    }
];
```

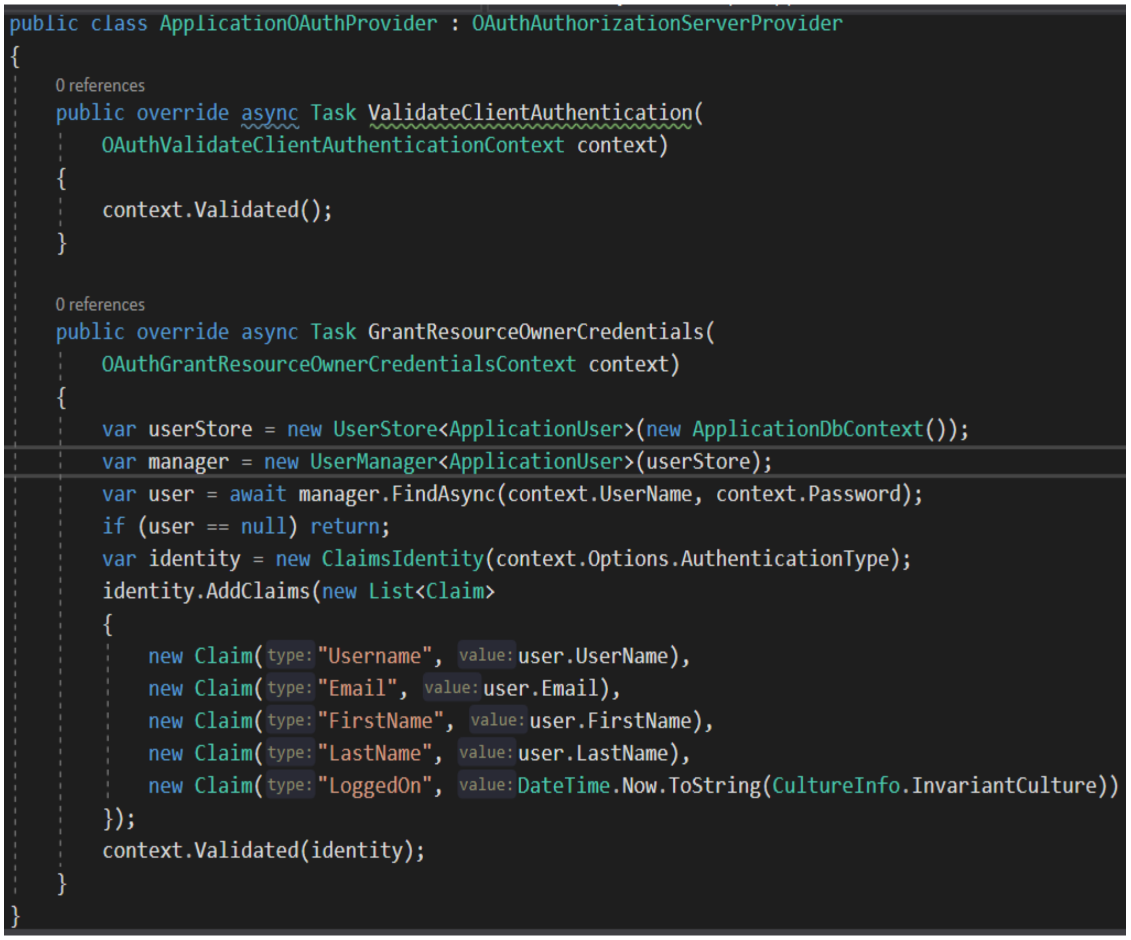
Routes will be modified once we will as guards to our application.

- Design the login form.

```
<div *ngIf="isLoginError" class="red-text center error-message">
  <i class="material-icons">error</i> Incorrect username or password
</div>

<form class="col s12 white" #loginForm="ngForm"
(ngSubmit)="login(userName.value, password.value)">
  <div class="row">
    <div class="input-field col s12">
      <i class="material-icons prefix">account_circle</i>
      <input type="text" name="userName" #userName ngModel
placeholder="Username" required>
    </div>
  </div>

  <div class="row">
    <div class="input-field col s12">
      <i class="material-icons prefix">vpn_key</i>
      <input type="password" name="password" #password ngModel
placeholder="password" required>
    </div>
  </div>

  <div class="row">
    <div class="input-field col s12">
      <button [disabled]="!loginForm.valid" class="btn-large btn-
submit" type="submit">Login</button>
```

```
    </div>
  </div>
</form>
```

- Token based user Authentication in Web API
  - ✓ In order to implement user authentication, OWIN is required.
  - ✓ **OWIN:** Open Web Interface for .NET Applications
  - ✓ OWIN acts as a middleware between .NET Application and IIS Server.
  - ✓ Install below packages from nugget:
    - ❖ **Microsoft.AspNet.Identity.Owin:** This provides OWIN implementation for ASP.NET Identity.
    - ❖ **Microsoft.Owin.Host.SystemWeb:** OWIN server that enables OWIN-based applications to run on IIS using the ASP.NET request pipeline.
    - ❖ **Microsoft.Owin.Cors:** Contains the components to enable CORS in OWIN middleware.
  - ✓ Create ApplicationOAuthProvider class to validate the username and password.

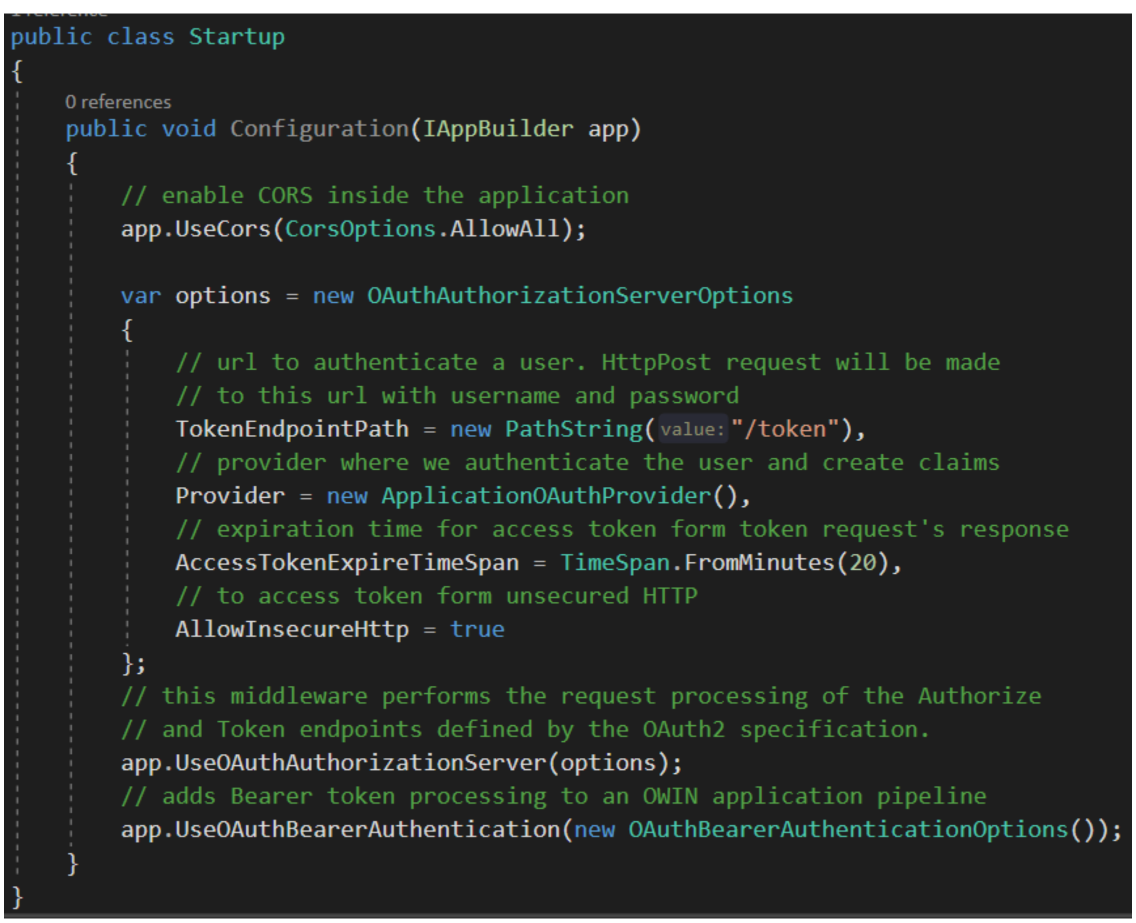


```
public class ApplicationOAuthProvider : OAuthAuthorizationServerProvider
{
    0 references
    public override async Task ValidateClientAuthentication(
        OAuthValidateClientAuthenticationContext context)
    {
        context.Validated();
    }

    0 references
    public override async Task GrantResourceOwnerCredentials(
        OAuthGrantResourceOwnerCredentialsContext context)
    {
        var userStore = new UserStore<ApplicationUser>(new ApplicationDbContext());
        var manager = new UserManager<ApplicationUser>(userStore);
        var user = await manager.FindAsync(context.UserName, context.Password);
        if (user == null) return;
        var identity = new ClaimsIdentity(context.Options.AuthenticationType);
        identity.AddClaims(new List<Claim>
        {
            new Claim(type: "Username", value: user.UserName),
            new Claim(type: "Email", value: user.Email),
            new Claim(type: "FirstName", value: user.FirstName),
            new Claim(type: "LastName", value: user.LastName),
            new Claim(type: "LoggedOn", value: DateTime.Now.ToString(CultureInfo.InvariantCulture))
        });
        context.Validated(identity);
    }
}
```

ValidateClientAuthentication(): this method is used to validate client device based on Client Id and secret code. (In our application we are not using this method).

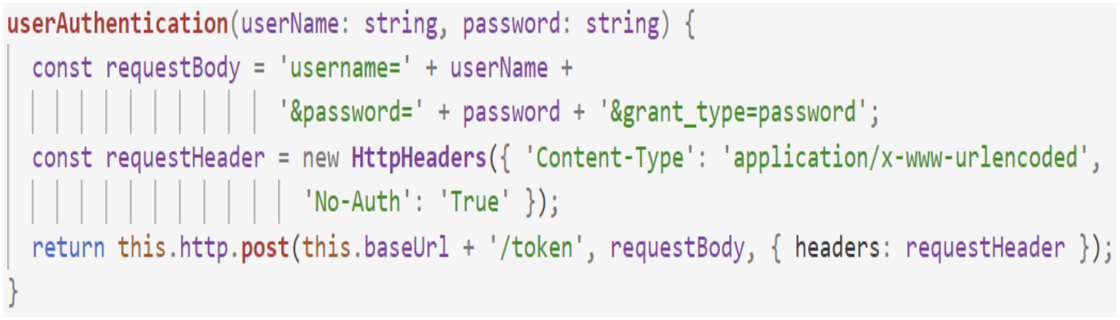
GrantResourceOwnerCredentials(): This method is used to authenticate a user with the given credentials. If authentication is successful, we save user details as Claims Identity. From Claims user details can be accessed without DB interaction.

- ✓ Create OWIN Startup class to work with OWIN.
  A template for adding OWIN Startup Class is available in VS.



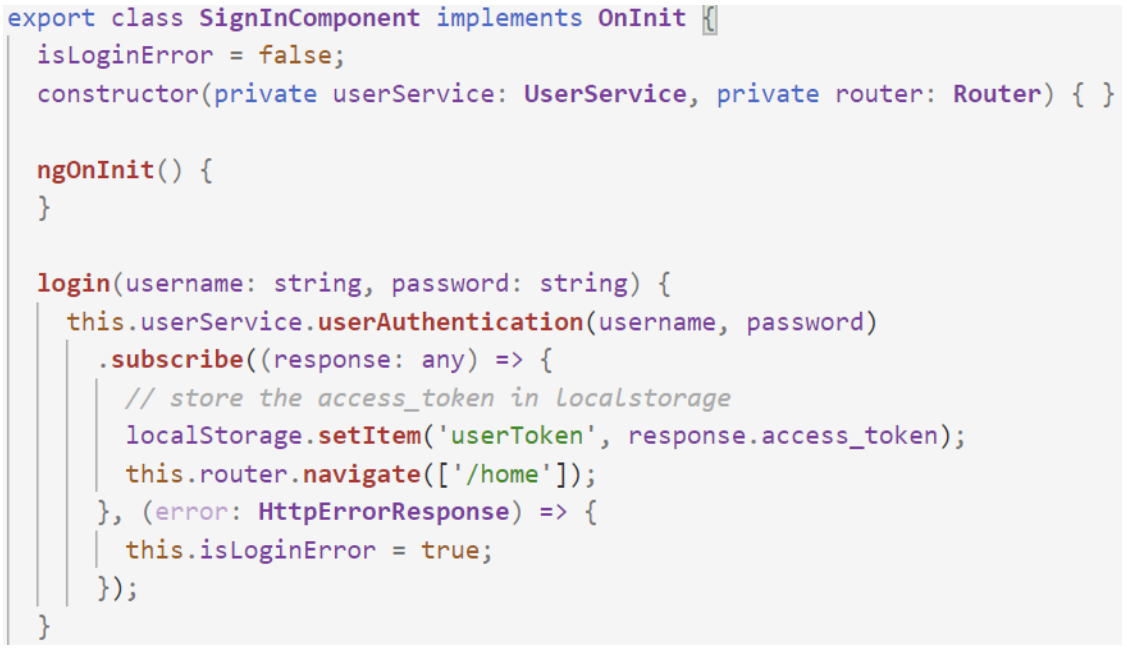
```
public class Startup
{
    0 references
    public void Configuration(IAppBuilder app)
    {
        // enable CORS inside the application
        app.UseCors(CorsOptions.AllowAll);

        var options = new OAuthAuthorizationServerOptions
        {
            // url to authenticate a user. HttpPost request will be made
            // to this url with username and password
            TokenEndpointPath = new PathString(value: "/token"),
            // provider where we authenticate the user and create claims
            Provider = new ApplicationOAuthProvider(),
            // expiration time for access token form token request's response
            AccessTokenExpireTimeSpan = TimeSpan.FromMinutes(20),
            // to access token form unsecured HTTP
            AllowInsecureHttp = true
        };
        // this middleware performs the request processing of the Authorize
        // and Token endpoints defined by the OAuth2 specification.
        app.UseOAuthAuthorizationServer(options);
        // adds Bearer token processing to an OWIN application pipeline
        app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());
    }
}
```

- Requesting Token from Angular
  - ✓ Inside user.service.ts file add one more method to send authentication request.



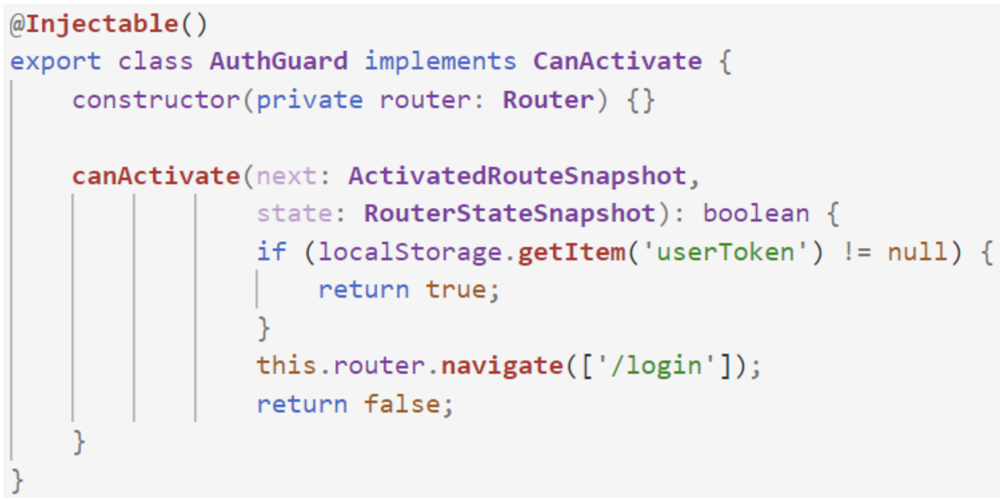
```
userAuthentication(userName: string, password: string) {
  const requestBody = 'username=' + userName +
                      '&password=' + password + '&grant_type=password';
  const requestHeader = new HttpHeaders({ 'Content-Type': 'application/x-www-urlencoded',
                        'No-Auth': 'True' });
  return this.http.post(this.baseUrl + '/token', requestBody, { headers: requestHeader });
}
```

✓ Use the userAuthentication method in sign-in.component.ts file to login



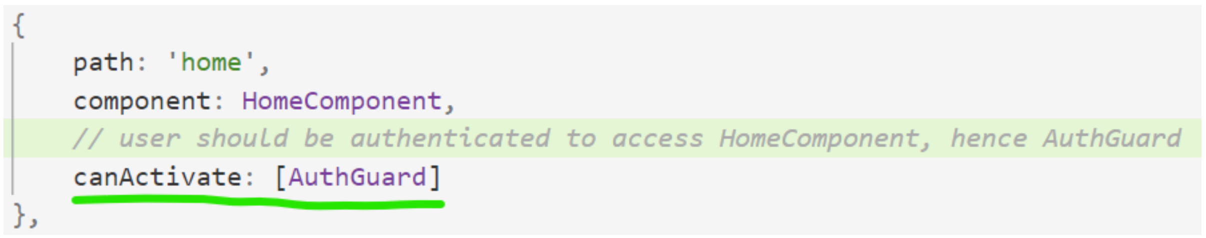
```
export class SignInComponent implements OnInit {
  isLoginError = false;
  constructor(private userService: UserService, private router: Router) { }

  ngOnInit() {
  }

  login(username: string, password: string) {
    this.userService.userAuthentication(username, password)
      .subscribe((response: any) => {
        // store the access_token in localstorage
        localStorage.setItem('userToken', response.access_token);
        this.router.navigate(['/home']);
      }, (error: HttpErrorResponse) => {
        this.isLoginError = true;
      });
  }
```

- User Authentication using Token



```
@Injectable()
export class AuthGuard implements CanActivate {
    constructor(private router: Router) {}

    canActivate(next: ActivatedRouteSnapshot,
                state: RouterStateSnapshot): boolean {
                if (localStorage.getItem('userToken') != null) {
                    return true;
                }
                this.router.navigate(['/login']);
                return false;
    }
}
```

With canActivate() function, we will check whether user is authenticated or not using userToken present in local storage.

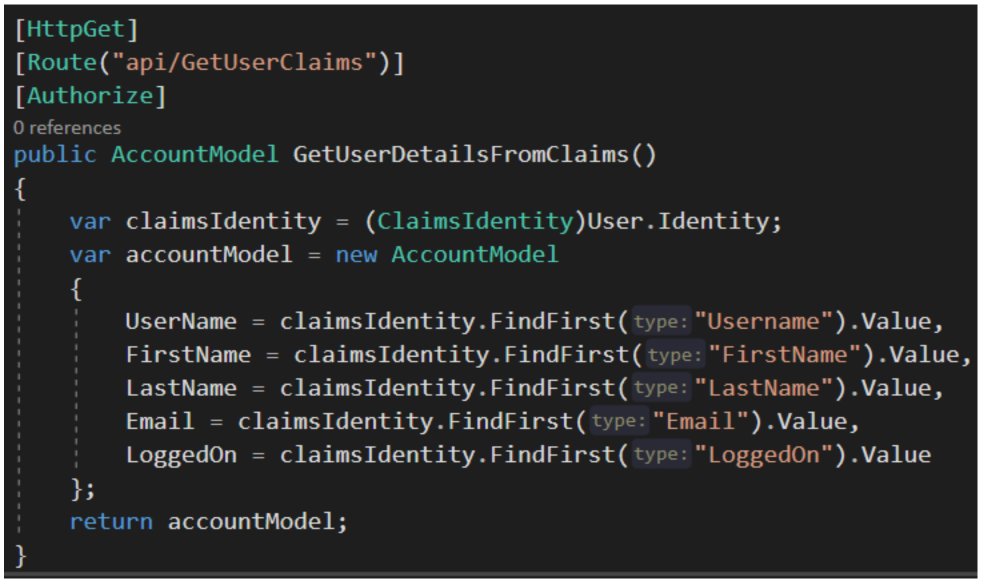


```
{
    path: 'home',
    component: HomeComponent,
    // user should be authenticated to access HomeComponent, hence AuthGuard
    canActivate: [AuthGuard]
},
```

Update routes for home as above.

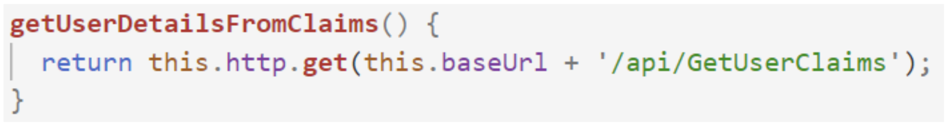
- Consume Web API methods with Authorization

Fetch claims stored during authentication, inside AccountController, to be shown in home component.



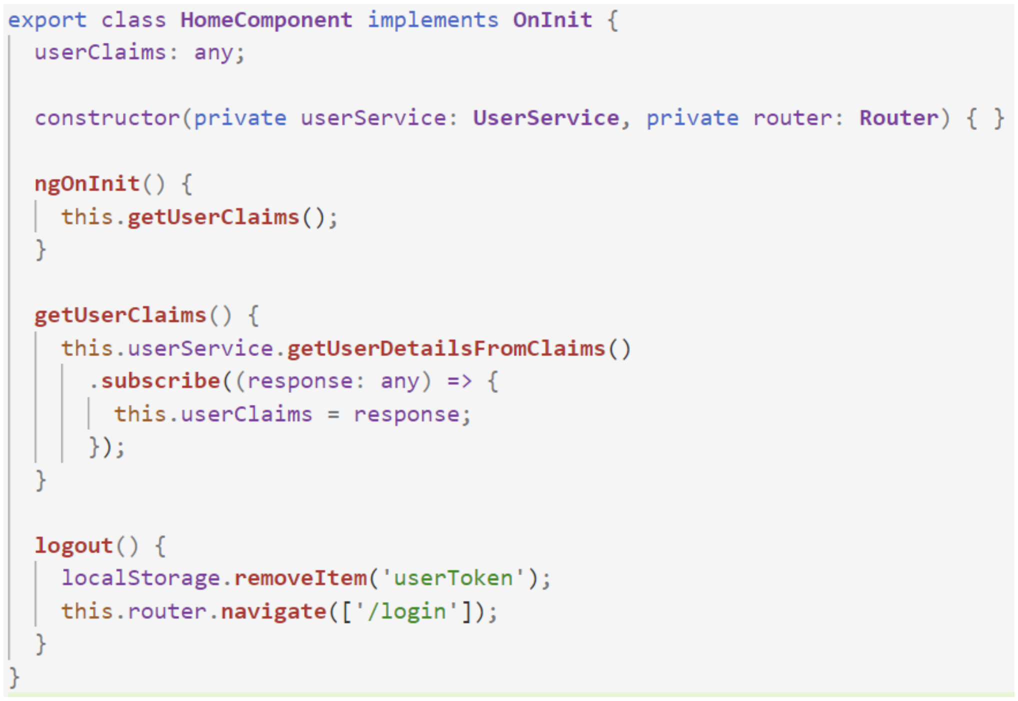
```
[HttpGet]
[Route("api/GetUserClaims")]
[Authorize]
0 references
public AccountModel GetUserDetailsFromClaims()
{
    var claimsIdentity = (ClaimsIdentity)User.Identity;
    var accountModel = new AccountModel
    {
        UserName = claimsIdentity.FindFirst(type: "Username").Value,
        FirstName = claimsIdentity.FindFirst(type: "FirstName").Value,
        LastName = claimsIdentity.FindFirst(type: "LastName").Value,
        Email = claimsIdentity.FindFirst(type: "Email").Value,
        LoggedOn = claimsIdentity.FindFirst(type: "LoggedOn").Value
    };
    return accountModel;
}
```

Update UserService in angular.



```
getUserDetailsFromClaims() {
  return this.http.get(this.baseUrl + '/api/GetUserClaims');
}
```

Prepare home.component.ts file for getting user claims and also for logout.



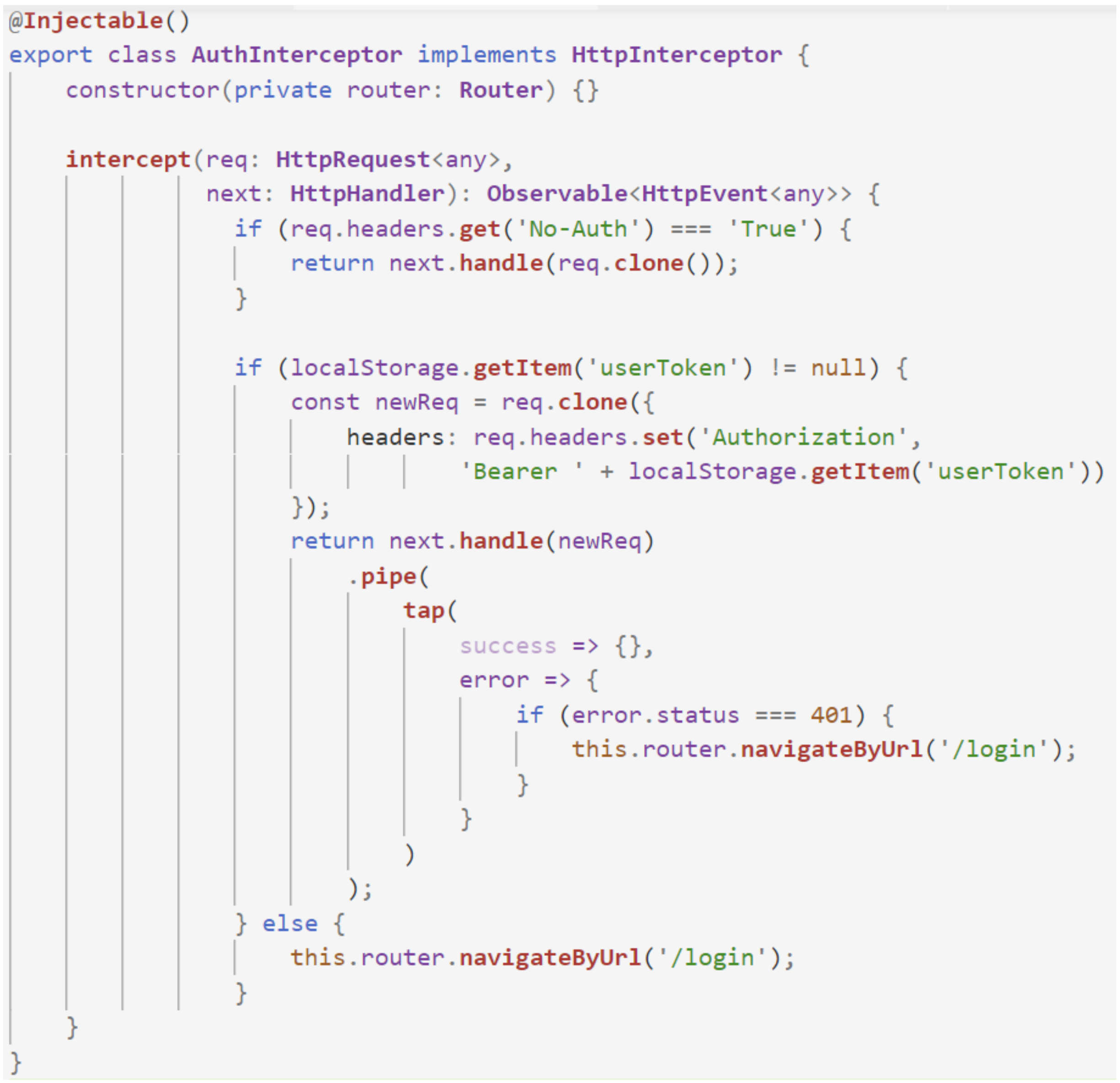
```
export class HomeComponent implements OnInit {
  userClaims: any;

  constructor(private userService: UserService, private router: Router) { }

  ngOnInit() {
    this.getUserClaims();
  }

  getUserClaims() {
    this.userService.getUserDetailsFromClaims()
      .subscribe((response: any) => {
        this.userClaims = response;
      });
  }

  logout() {
    localStorage.removeItem('userToken');
    this.router.navigate(['/login']);
  }
}
```

home.component.html

```html
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo center">
      <i class="material-icons">cloud</i>Dotnet Mob App</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <li>
        <a (click)="logout()">Logout</a>
      </li>
    </ul>
  </div>
</nav>
<div class="row" *ngIf="userClaims">
  <div class="col s12 m7">
    <div class="card">
      <div class="card-content">
        <span>Username :{{userClaims.UserName}}</span>
        <br>
        <span>Email : {{userClaims.Email}}</span>
        <br>
        <span>Full Name : {{userClaims.FirstName}}
{{userClaims.LastName}}</span>
        <br>
        <span>Logged On : {{userClaims.LoggedOn}}</span>
      </div>
    </div>
  </div>
</div>
```

- Using HTTP Interceptors
  While calling getUserDetailsFromClaims() method we have not appended the access token. Here, it is one method. In real world, there will be multiple methods. So, it will be difficult to append access token to all the methods separately.

  Hence we use HTTP Interceptors.

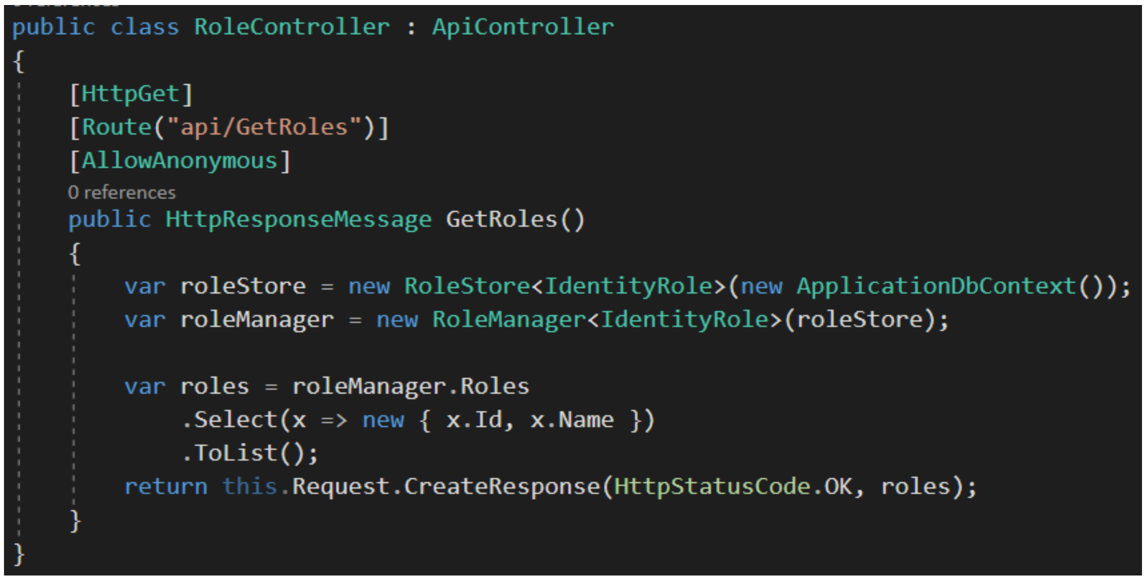


```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
    constructor(private router: Router) {}

    intercept(req: HttpRequest<any>,
              next: HttpHandler): Observable<HttpEvent<any>> {
        if (req.headers.get('No-Auth') === 'True') {
            return next.handle(req.clone());
        }

        if (localStorage.getItem('userToken') != null) {
            const newReq = req.clone({
                headers: req.headers.set('Authorization',
                    'Bearer ' + localStorage.getItem('userToken'))
            });
            return next.handle(newReq)
                .pipe(
                    tap(
                        success => {},
                        error => {
                            if (error.status === 401) {
                                this.router.navigateByUrl('/login');
                            }
                        }
                    )
                );
        } else {
            this.router.navigateByUrl('/login');
        }
    }
}
```

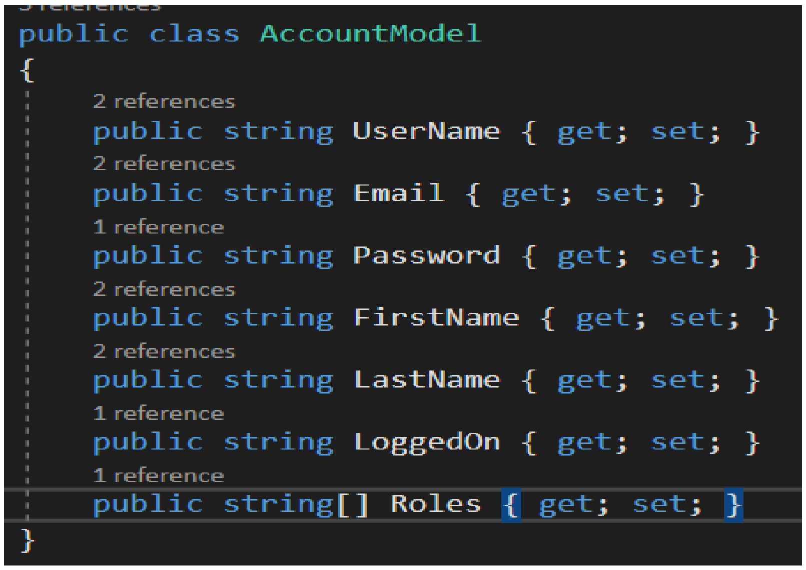
With first if statement, we check whether No-Auth is True/False. If it is true, means we don’t require to append the access token with the request. This implies we can set No-Auth as true for Web API calls which do not require authorization.

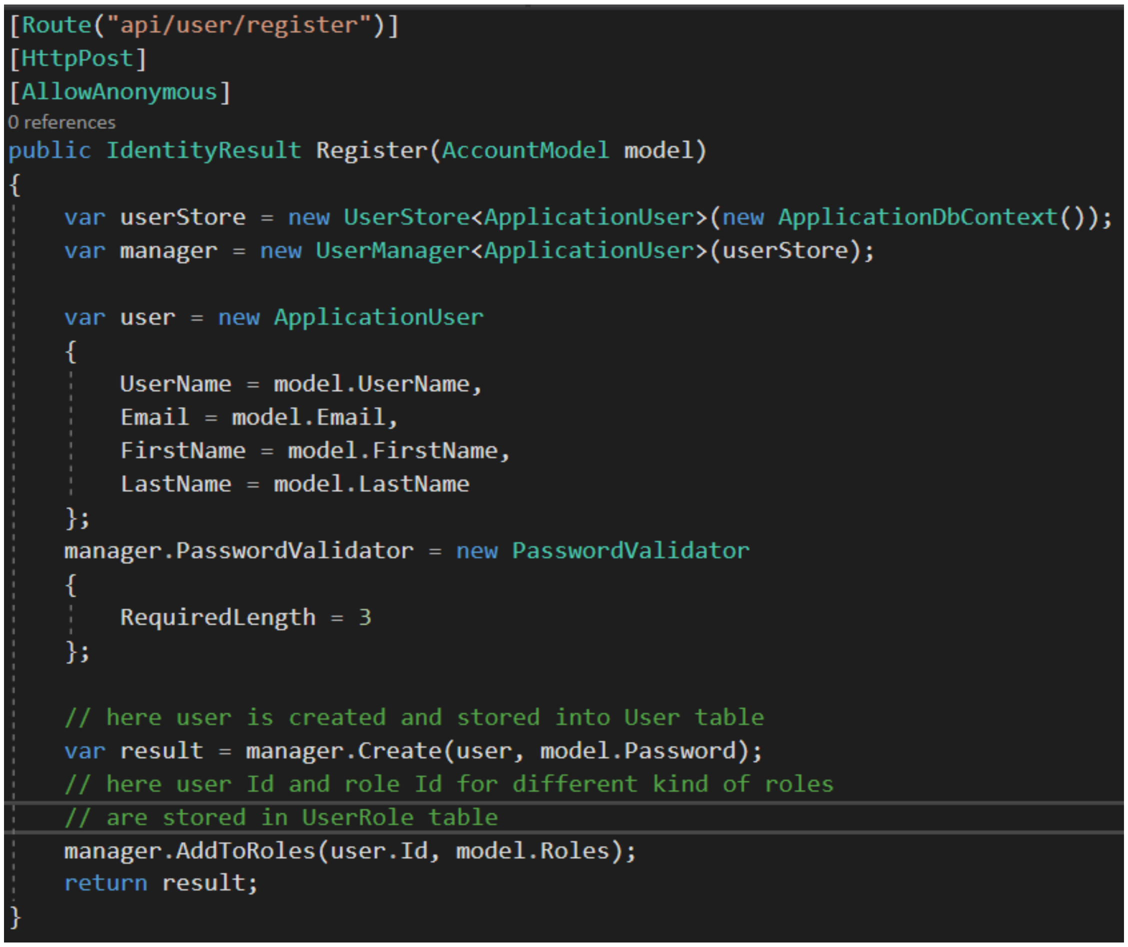
If user is authenticated, i.e. token exist, we append the access token in the request header.

- Role based Authorization
    - Create Role Controller, to get all the roles from Role table in the db.


```
public class RoleController : ApiController
{
    [HttpGet]
    [Route("api/GetRoles")]
    [AllowAnonymous]
    0 references
    public HttpResponseMessage GetRoles()
    {
        var roleStore = new RoleStore<IdentityRole>(new ApplicationDbContext());
        var roleManager = new RoleManager<IdentityRole>(roleStore);

        var roles = roleManager.Roles
            .Select(x => new { x.Id, x.Name })
            .ToList();
        return this.Request.CreateResponse(HttpStatusCode.OK, roles);
    }
}
```

- Update AccountController, register method as we have additional item as Role.

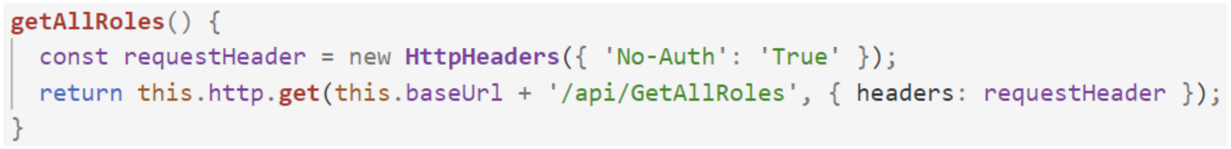

```
public class AccountModel
{
    2 references
    public string UserName { get; set; }
    2 references
    public string Email { get; set; }
    1 reference
    public string Password { get; set; }
    2 references
    public string FirstName { get; set; }
    2 references
    public string LastName { get; set; }
    1 reference
    public string LoggedOn { get; set; }
    1 reference
    public string[] Roles { get; set; }
}
```



```
[Route("api/user/register")]
[HttpPost]
[AllowAnonymous]
0 references
public IdentityResult Register(AccountModel model)
{
    var userStore = new UserStore<ApplicationUser>(new ApplicationDbContext());
    var manager = new UserManager<ApplicationUser>(userStore);

    var user = new ApplicationUser
    {
        UserName = model.UserName,
        Email = model.Email,
        FirstName = model.FirstName,
        LastName = model.LastName
    };
    manager.PasswordValidator = new PasswordValidator
    {
        RequiredLength = 3
    };

    // here user is created and stored into User table
    var result = manager.Create(user, model.Password);
    // here user Id and role Id for different kind of roles
    // are stored in UserRole table
    manager.AddToRoles(user.Id, model.Roles);
    return result;
}
```

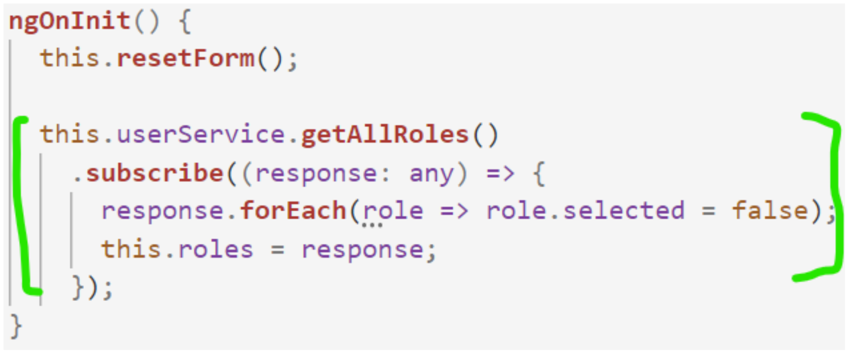
The new line added above will insert selected role for the user in **UserRole** table.

- Add method getAllRoles() in UserService class in angular.



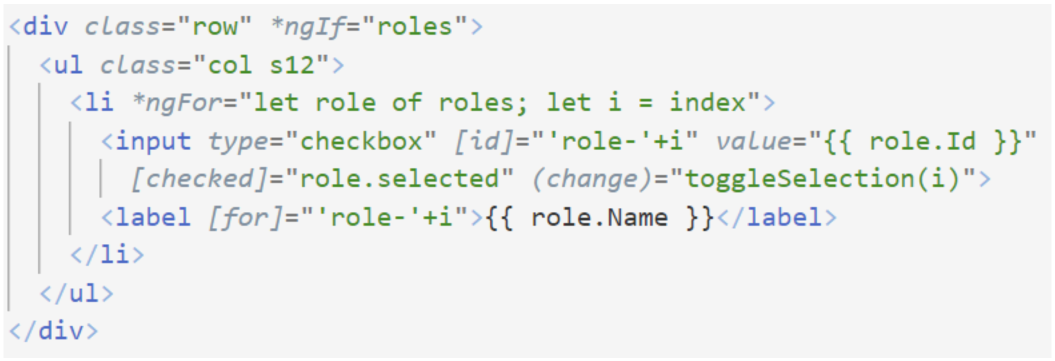
```
getAllRoles() {
  const requestHeader = new HttpHeaders({ 'No-Auth': 'True' });
  return this.http.get(this.baseUrl + '/api/GetAllRoles', { headers: requestHeader });
}
```

- Update ngOnInit() of sign-up.component.ts as below.



```
ngOnInit() {
  this.resetForm();

  this.userService.getAllRoles()
    .subscribe((response: any) => {
      response.forEach(role => role.selected = false);
      this.roles = response;
    });
}
```

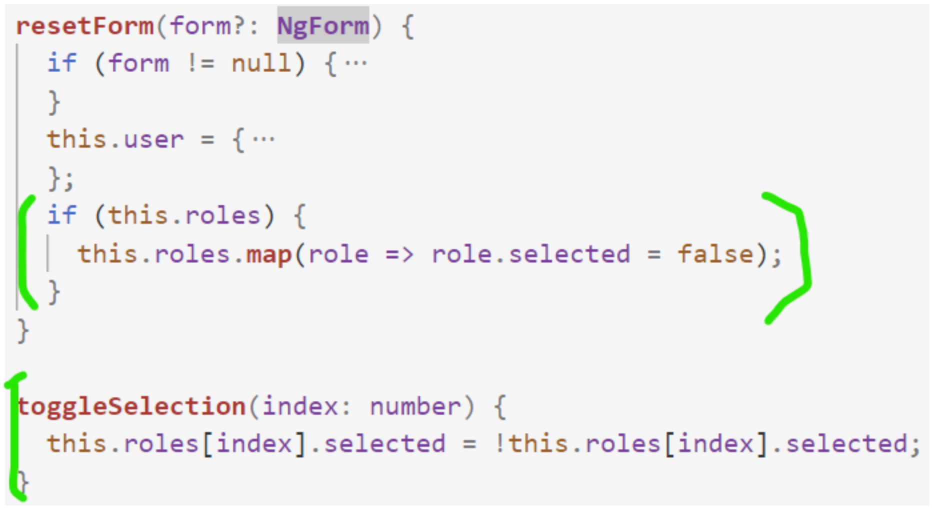
- Update Sign up Form in Angular



```html
<div class="row" *ngIf="roles">
  <ul class="col s12">
    <li *ngFor="let role of roles; let i = index">
      <input type="checkbox" [id]="'role-'+i" value="{{ role.Id }}"
        [checked]="role.selected" (change)="toggleSelection(i)">
      <label [for]="'role-'+i">{{ role.Name }}</label>
    </li>
  </ul>
</div>
```

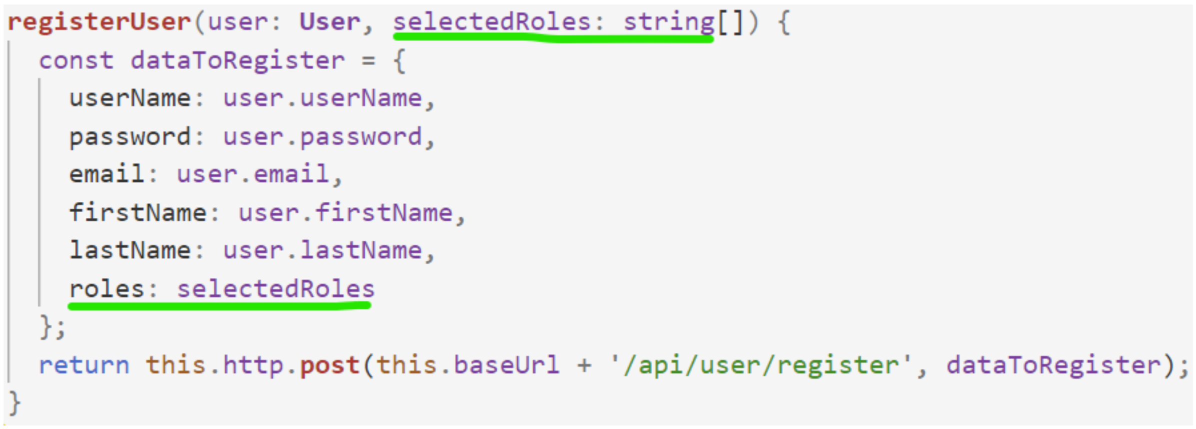
Add checkboxes above submit button.

- Update sign-up.component.ts as below.



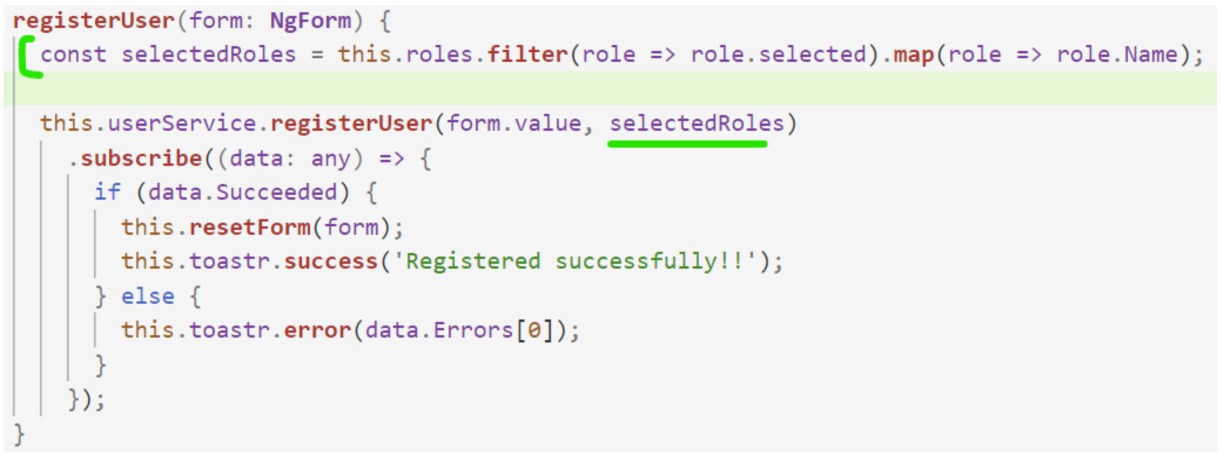
```typescript
resetForm(form?: NgForm) {
  if (form != null) {…
  }
  this.user = {…
  };
  if (this.roles) {
    this.roles.map(role => role.selected = false);
  }
}

toggleSelection(index: number) {
  this.roles[index].selected = !this.roles[index].selected;
}
```

- Update registerUser method in UserService class in angular



```typescript
registerUser(user: User, selectedRoles: string[]) {
  const dataToRegister = {
    userName: user.userName,
    password: user.password,
    email: user.email,
    firstName: user.firstName,
    lastName: user.lastName,
    roles: selectedRoles
  };
  return this.http.post(this.baseUrl + '/api/user/register', dataToRegister);
}
```

- Update registerUser method in sign-up.component.ts



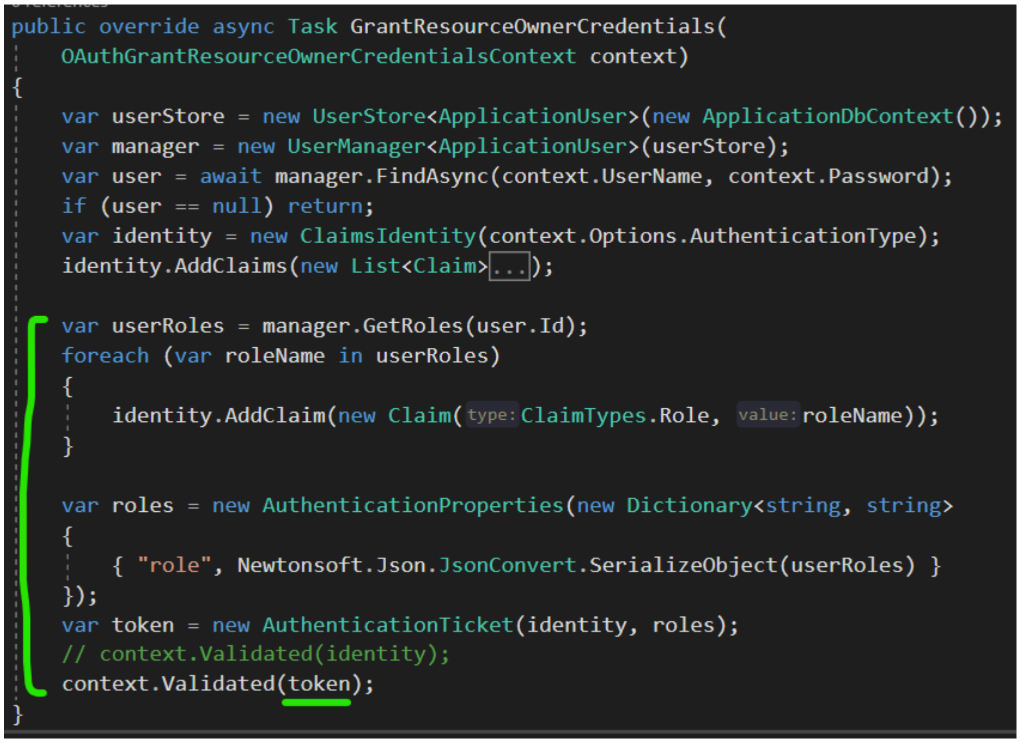
```
registerUser(form: NgForm) {
  const selectedRoles = this.roles.filter(role => role.selected).map(role => role.Name);

  this.userService.registerUser(form.value, selectedRoles)
    .subscribe((data: any) => {
      if (data.Succeeded) {
        this.resetForm(form);
        this.toastr.success('Registered successfully!!');
      } else {
        this.toastr.error(data.Errors[0]);
      }
    });
}
```

Till here we can try to register a user with our new role changes. Before registering a user at this stage, we need to apply **[AllowAnonymous] attribute** on Register method of AccountController.

After successful registration, user Id and role Id mapping will be established in UserRole table.

- Implement Role based Authorization

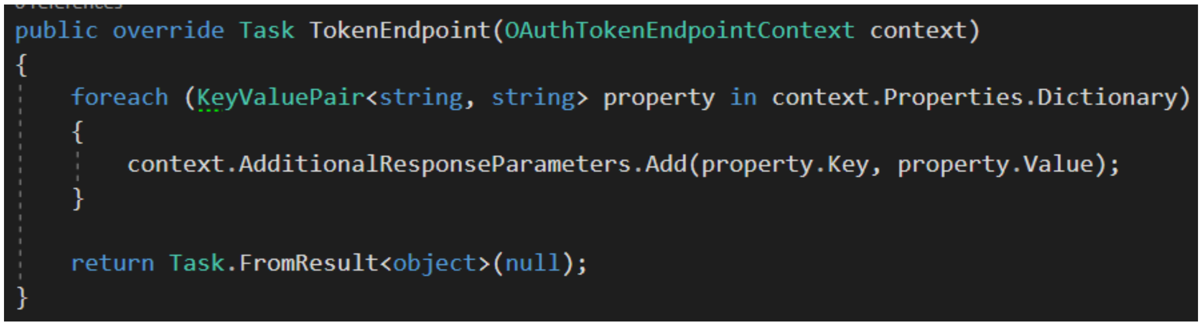
Firstly, we need to store roles assigned to a user in Claims during authentication or login. We will do the same in GrantResourceOwnerCredentials method.



```
public override async Task GrantResourceOwnerCredentials(
    OAuthGrantResourceOwnerCredentialsContext context)
{
    var userStore = new UserStore<ApplicationUser>(new ApplicationDbContext());
    var manager = new UserManager<ApplicationUser>(userStore);
    var user = await manager.FindAsync(context.UserName, context.Password);
    if (user == null) return;
    var identity = new ClaimsIdentity(context.Options.AuthenticationType);
    identity.AddClaims(new List<Claim>...);

    var userRoles = manager.GetRoles(user.Id);
    foreach (var roleName in userRoles)
    {
        identity.AddClaim(new Claim(type: ClaimTypes.Role, value: roleName));
    }

    var roles = new AuthenticationProperties(new Dictionary<string, string>
    {
        { "role", Newtonsoft.Json.JsonConvert.SerializeObject(userRoles) }
    });
    var token = new AuthenticationTicket(identity, roles);
    // context.Validated(identity);
    context.Validated(token);
}
```

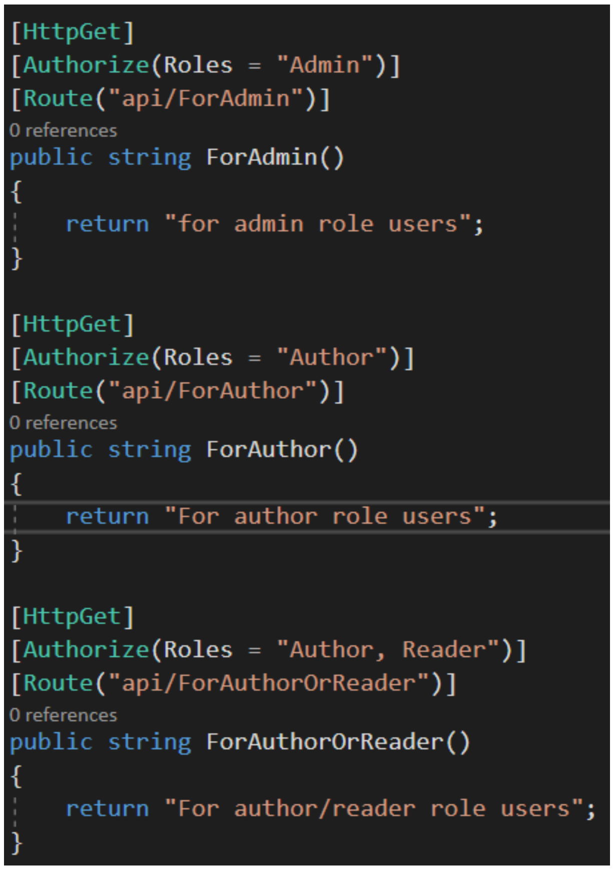
Here, we are providing roles as additional data through AuthenticationTicket, which can be consumed by the user or runtime.

To send the above created data at client side, proper formatting is required, as below.



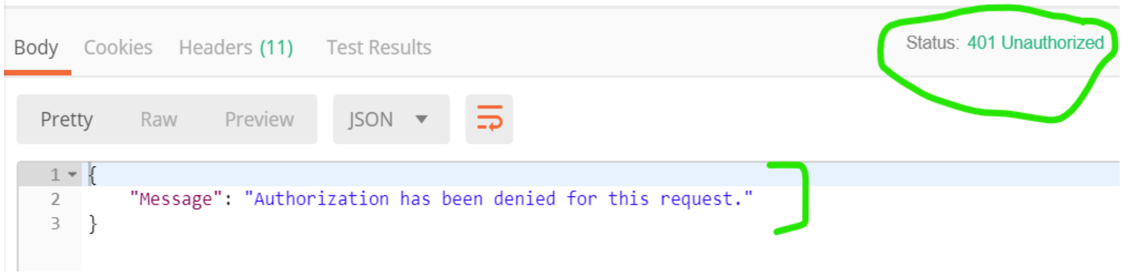
```
public override Task TokenEndpoint(OAuthTokenEndpointContext context)
{
    foreach (KeyValuePair<string, string> property in context.Properties.Dictionary)
    {
        context.AdditionalResponseParameters.Add(property.Key, property.Value);
    }

    return Task.FromResult<object>(null);
}
```

- Just for testing (in Postman), we will add three dummy methods in AccountController class.



```
[HttpGet]
[Authorize(Roles = "Admin")]
[Route("api/ForAdmin")]
0 references
public string ForAdmin()
{
    return "for admin role users";
}

[HttpGet]
[Authorize(Roles = "Author")]
[Route("api/ForAuthor")]
0 references
public string ForAuthor()
{
    return "For author role users";
}

[HttpGet]
[Authorize(Roles = "Author, Reader")]
[Route("api/ForAuthorOrReader")]
0 references
public string ForAuthorOrReader()
{
    return "For author/reader role users";
}
```

When we test in postman, we find that if an unauthorized user for a particular method tries to access that method, we get 401 Unauthorized.

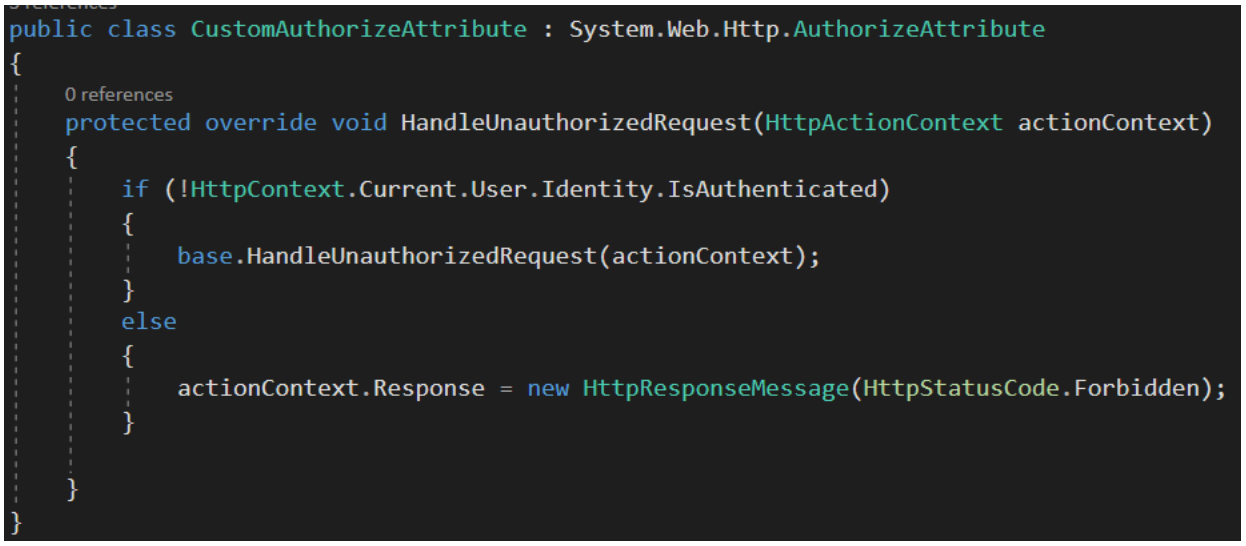


Status: 401 Unauthorized

```
{
    "Message": "Authorization has been denied for this request."
}
```

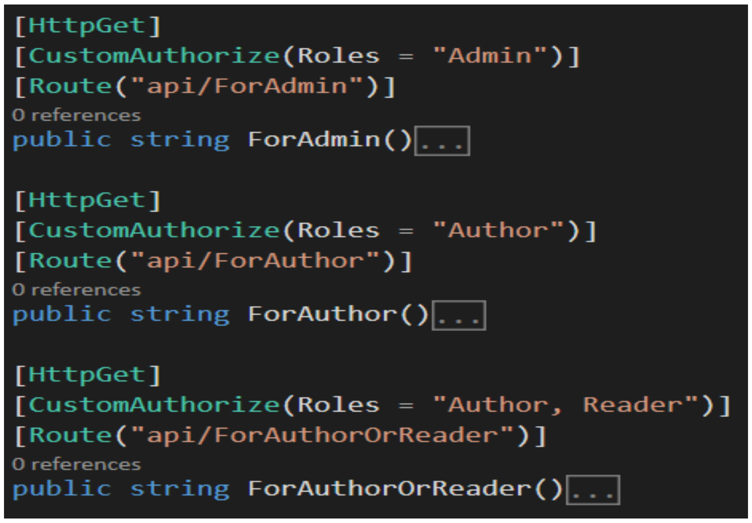
401 Unauthorized states that we are not authenticated to access the application, which is misleading. For this request, we should get **403 Forbidden**.

In order to achieve this, we will override this default behavior in a custom Authorize attribute.



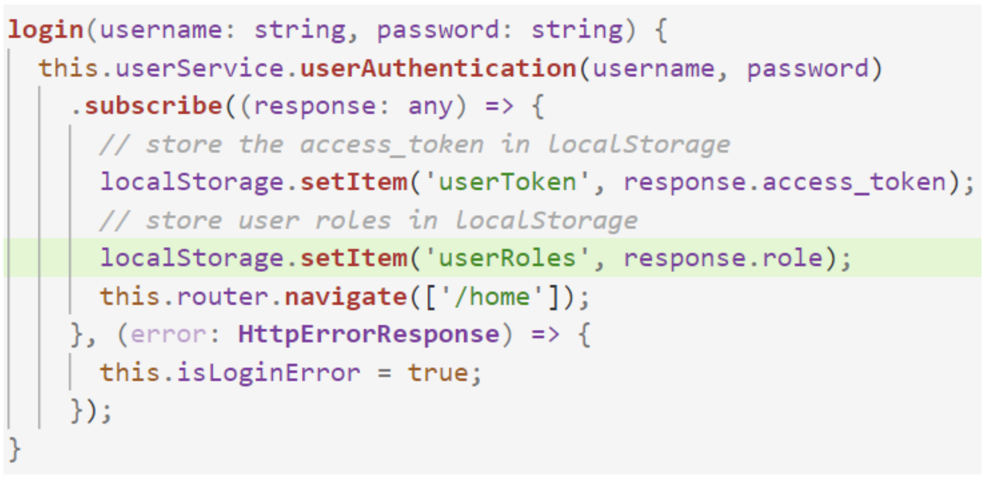
```
public class CustomAuthorizeAttribute : System.Web.Http.AuthorizeAttribute
{
    0 references
    protected override void HandleUnauthorizedRequest(HttpActionContext actionContext)
    {
        if (!HttpContext.Current.User.Identity.IsAuthenticated)
        {
            base.HandleUnauthorizedRequest(actionContext);
        }
        else
        {
            actionContext.Response = new HttpResponseMessage(HttpStatusCode.Forbidden);
        }

    }
}
```

And post this, we can add CustomAttribute on the dummy methods which we created.



```
[HttpGet]
[CustomAuthorize(Roles = "Admin")]
[Route("api/ForAdmin")]
0 references
public string ForAdmin()...

[HttpGet]
[CustomAuthorize(Roles = "Author")]
[Route("api/ForAuthor")]
0 references
public string ForAuthor()...

[HttpGet]
[CustomAuthorize(Roles = "Author, Reader")]
[Route("api/ForAuthorOrReader")]
0 references
public string ForAuthorOrReader()...
```

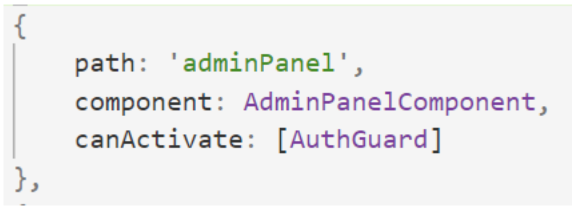
- Role Based Authorization at client end.
  Store user roles also in localStorage.



```
login(username: string, password: string) {
  this.userService.userAuthentication(username, password)
    .subscribe((response: any) => {
      // store the access_token in localStorage
      localStorage.setItem('userToken', response.access_token);
      // store user roles in localStorage
      localStorage.setItem('userRoles', response.role);
      this.router.navigate(['/home']);
    }, (error: HttpErrorResponse) => {
      this.isLoginError = true;
    });
}
```

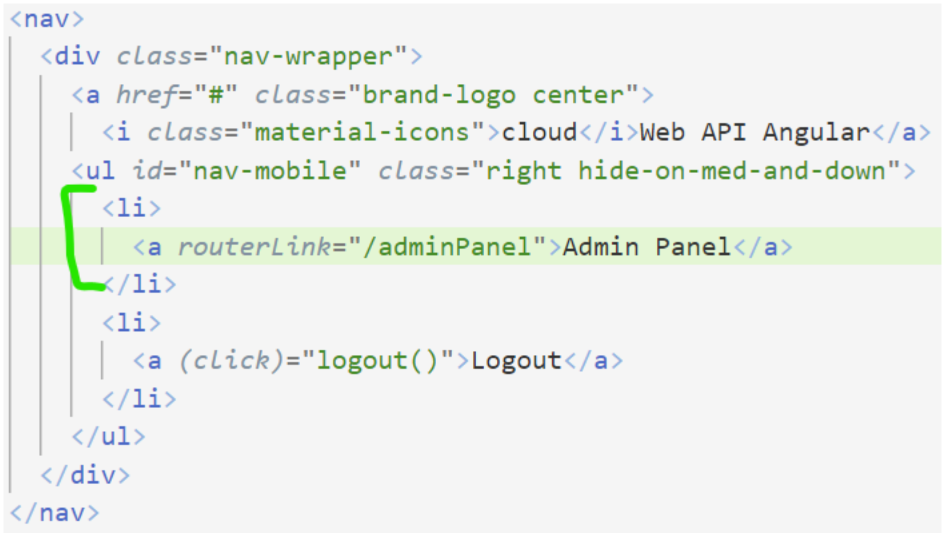
- Role Based Menu

  Create admin panel component and add it to route.



```
{
    path: 'adminPanel',
    component: AdminPanelComponent,
    canActivate: [AuthGuard]
},
```

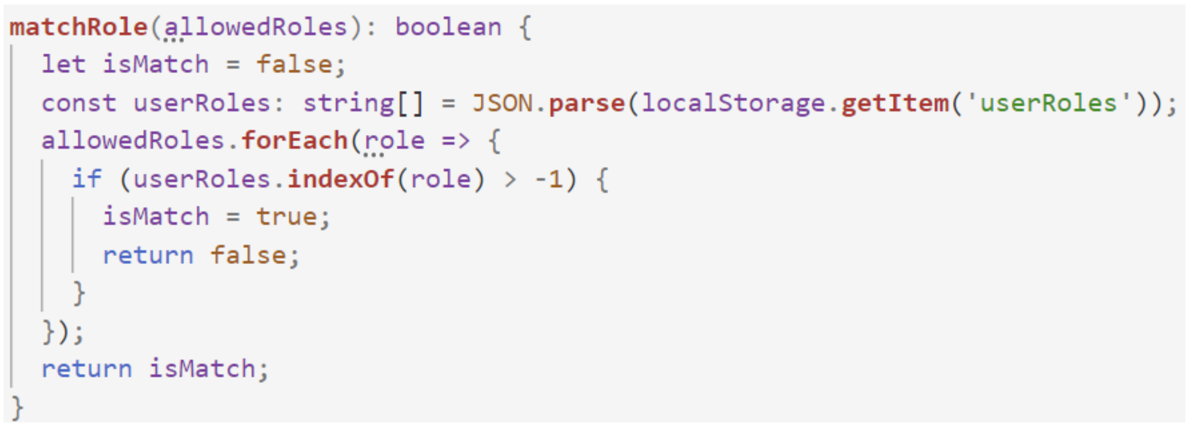
Add router link for admin panel in home.component.html



```
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo center">
      <i class="material-icons">cloud</i>Web API Angular</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <li>
        <a routerLink="/adminPanel">Admin Panel</a>
      </li>
      <li>
        <a (click)="logout()">Logout</a>
      </li>
    </ul>
  </div>
</nav>
```

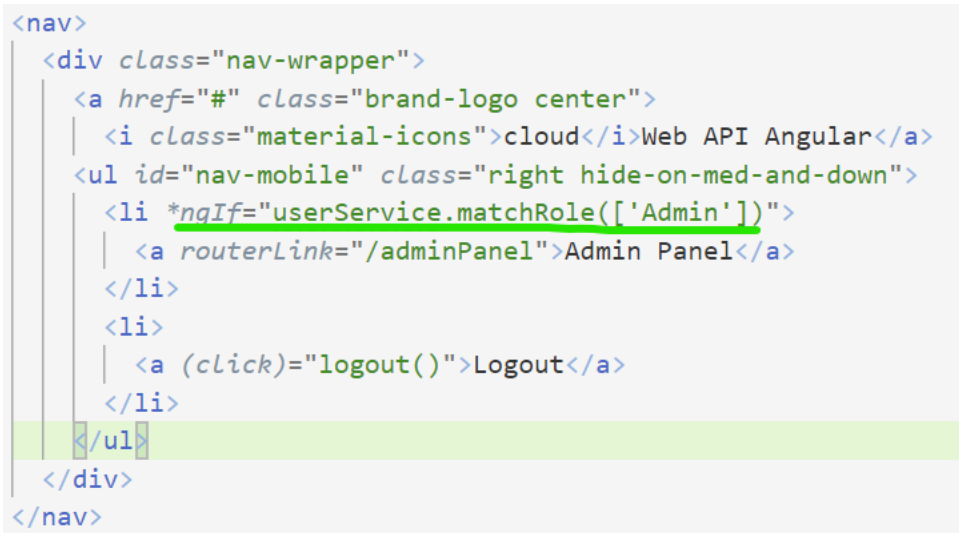
To make above Admin Panel role based, we need to put condition in ngIf on the li, stating true only if user is Admin.

For that we need to find whether role is getting satisfied. So, create a matchRole method in UserService class in angular.



```
matchRole(allowedRoles): boolean {
  let isMatch = false;
  const userRoles: string[] = JSON.parse(localStorage.getItem('userRoles'));
  allowedRoles.forEach(role => {
    if (userRoles.indexOf(role) > -1) {
      isMatch = true;
      return false;
    }
  });
  return isMatch;
}
```

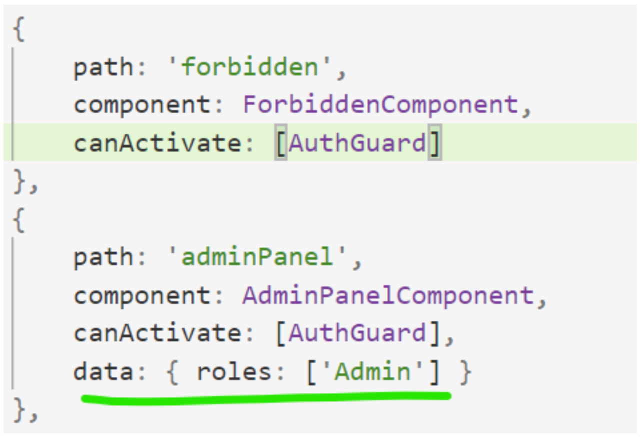
Now,



```
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo center">
      <i class="material-icons">cloud</i>Web API Angular</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <li *ngIf="userService.matchRole(['Admin'])">
        <a routerLink="/adminPanel">Admin Panel</a>
      </li>
      <li>
        <a (click)="logout()">Logout</a>
      </li>
    </ul>
  </div>
</nav>
```

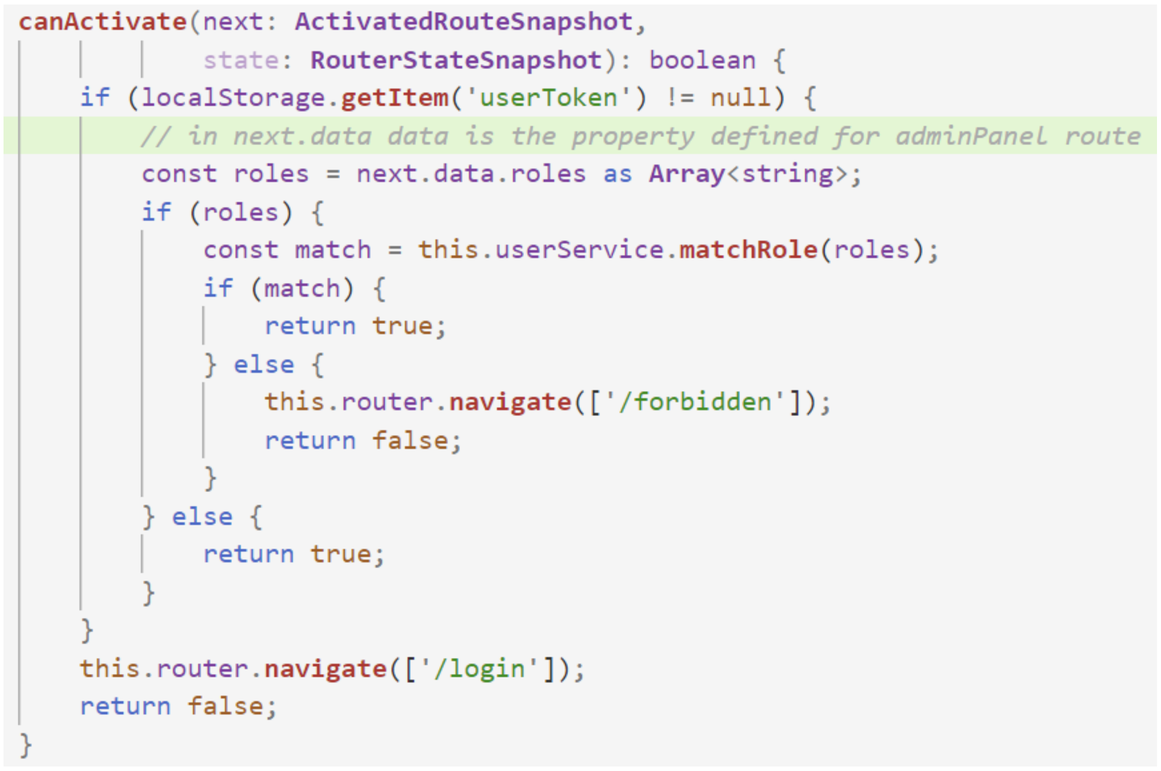
This will hide the Admin Panel menu from users who are not authorized to see Admin panel menu based on their role.

But, here we have one issue. The user who is not authorized to see Admin Panel, if provides direct path in url, the person can navigate to it. So, we need to have role based routing.



```
{
    path: 'forbidden',
    component: ForbiddenComponent,
    canActivate: [AuthGuard]
},
{
    path: 'adminPanel',
    component: AdminPanelComponent,
    canActivate: [AuthGuard],
    data: { roles: ['Admin'] }
},
```

Update the AuthGuard as below.



```
canActivate(next: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean {
    if (localStorage.getItem('userToken') != null) {
        // in next.data data is the property defined for adminPanel route
        const roles = next.data.roles as Array<string>;
        if (roles) {
            const match = this.userService.matchRole(roles);
            if (match) {
                return true;
            } else {
                this.router.navigate(['/forbidden']);
                return false;
            }
        } else {
            return true;
        }
    }
    this.router.navigate(['/login']);
    return false;
}
```