Token-based Authentication in Angular 6 with ASP.NET Core 2.1

Jignesh Trivedi Source Code Print 17 min read

14 Aug 2021

Advanced

33K Views

JSON Web Token (JWT) is the most popular and open standard interface that allows communication & data transmitting between parties as JSON. JWT is digitally signed, so the information is trusted and verified.

It consists of three parts: Header, Payload, and Signature; all the parts are separated by the dot (.). It looks like "hhhh.ppppp.sssss" and these parts are Base64Url encoded. The header part of JWT consists of two parts: the hash algorithm that used like RAS or HMAC SHA256, the second part is the type of token (it is always JWT). The Payload part of JWT contains the claims (user information such as Username, email, etc.) and other metadata. The claim can be either Registered, Public or Private.

The signature part of JWT is created by using the base64 encoded value of header and payload, a secret key that used by the algorithm (defined in the Header part of JWT) and signs that. This part is being used to verify the message is same as original i.e. wasn't change in transition.

The jwt.io debugger (https://jwt.io/) is enable us to decode the JWT (not decode signature part) and we can verify or generate new token.

Why Use JWT (JSON Web Token)?

One of the common answers to this question is, JWT is convenient, compact, and secure. The token is simply a based64 encoded string that contains the few header fields and payloads, so it usually contains fewer bytes compare to other tokens. It can be signed using either RSA public/private key-pair encryption or HMAC encryption using a shared secret, so they are more secure. It is open source libraries and readily available libraries with various technologies such as .net, Python, Java, Ruby, Swift etc.

There are many XML based tokens are available such as SAML (Security Assertion Markup Language) token, SWT (Simple Web Token), etc. The JWT is JSON based. The size of encoded JSON smaller compare to the size of encoded XML, so JSON Web Token is more compact than the other token. It is more secure because it uses public/private key in the form of an X.509 certificate for signing.

In this post, I will explain how to create application using Angular 6 app with ASP.net core 2.1 and do the token-based authentication using JWT.

Prerequisites

The following software needs to be installed in our system before starting the work.

.NET Core framework 2.1or above

TypeScript 2.5 and above.

Latest version of Node JS and npm (node package manager)

Editor such as VS 2017 or VS Code

ASP.NET Core comes with many built-in template such as Angular, React, etc. I am using Angular template to demonstrate the concept. We can create an application by using Angular template of Visual Studio (2017 or higher) or using CLI (Command Line Interface).

Following .net core CLI command will create an ASP.NET Web API project with Angular with project name "TokenbasedAuthentication" in the current folder.

<dotnet new angular –n TokenbasedAuthentication

In this template, Angular client App and Web API are shipped together in one project, however we can create two separate project: one for Angular and other for web API.

Configure JWT Token based Authentication

To Configure, JWT based authentication, we need to register a JWT authentication schema by using "AddAuthentication" method and specifying JwtBearerDefaults.AuthenticationScheme. Here, we configure the JWT bearer options for authentication schema using AddJwtBearer method.

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = Configuration["Jwt:Issuer"],

ValidAudience = Configuration["Jwt:Issuer"],

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["Jwt:Key"]))

};

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

// In production, the Angular files will be served from this directory

services.AddSpaStaticFiles(configuration =>

{

configuration.RootPath = "ClientApp/dist";

});

}

In this example, we have specified which parameters must be taken into account to consider JWT as valid. As per code mention in above section, the following items are considered to validate the token

Validate the server information that generates the JWT ((ValidateIssuer = true)

Validate the receiver of JWT (client) is authorized to receive (ValidateAudience = true)

Check JWT expiration (ValidateLifetime = true)

Check signature of the JWT (ValidateIssuerSigningKey = true)

In this example, I have specify the values for the issuer, audience, signing key that are stored in appsettings.json file.

AppSetting.Json

{

"Jwt": {

"Key": "ThisismyPrivateSecretKey",

"Issuer": "abc.com"

}

}

After configuring JWT based authentication service, the next step is to make it is available to the application. Using app.UseAuthentication() method, we can add the service to the application. This method defines in the "Configure" method of startup class. The UseAuthentication method must called before UseMvc method otherwise .net framework does not recognize authentication service.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

….

app.UseAuthentication();

app.UseMvc();

}

Generate JWT

To Generate JSON Web Token, I have created Login method under the LoginController that is responsible to generate the JWT. To bypass the authentication, I have marked this controller with "AllowAnonymous" attribute. Using UserModel, I have passed the username and password to this method. The AuthenticateUser method is responsible to validate User credential and returns to the UserModel. In this method, we have to write our custom code to validate user credential. For demonstration, I have only allowed “Jignesh” user i.e. I am checking hardcode value for username, if username is "Jignesh" then this method return user model and Web API generates the token by using GenerateJSONWebToken method.

With the help of JwtSecurityToken class object, we can create JWT. Here I have pass some parameters to constructor of the class while creating object of this class such as issuer, audience, expiration, claims, and signature. Finally using JwtSecurityTokenHandler.WriteToken method, we can generate the token based on the JwtSecurityToken class.

using System;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Configuration;

using Microsoft.IdentityModel.Tokens;

using TokenbasedAuthentication.Model;

namespace TokenbasedAuthentication.Controllers

{

[Route("api/[controller]")]

[ApiController]

[AllowAnonymous]

public class LoginController : ControllerBase

{

private IConfiguration \_config;

public LoginController(IConfiguration config)

{

\_config = config;

}

[HttpPost]

public IActionResult Login([FromBody]UserModel login)

{

IActionResult response = Unauthorized();

var user = AuthenticateUser(login);

if (user != null)

{

var tokenString = GenerateJSONWebToken(user);

response = Ok(new { token = tokenString });

}

return response;

}

private string GenerateJSONWebToken(UserModel userInfo)

{

var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_config["Jwt:Key"]));

var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

var claims = new[] {

new Claim(JwtRegisteredClaimNames.Sub, userInfo.Username),

new Claim(JwtRegisteredClaimNames.Email, userInfo.EmailAddress),

new Claim("DateOfJoing", userInfo.DateOfJoing.ToString("yyyy-MM-dd")),

new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())

};

var token = new JwtSecurityToken(\_config["Jwt:Issuer"],

\_config["Jwt:Issuer"],

claims,

expires: DateTime.Now.AddMinutes(120),

signingCredentials: credentials);

return new JwtSecurityTokenHandler().WriteToken(token);

}

private UserModel AuthenticateUser(UserModel login)

{

UserModel user = null;

//Validate the User Credentials

//Demo Purpose, I have Passed HardCoded User Information

if (login.Username == "Jignesh")

{

user = new UserModel { Username = "Jignesh Trivedi", EmailAddress = "test.btest@gmail.com" };

}

return user;

}

}

}

Once, we have enabled the JWT based authentication, we need to pass the token with request header which marked with authorize attribute. If we call a method (that mark with authorize attribute) without token or invalid token, we will get 401 (UnAuthorizedAccess) - HTTP status code as a response. The AllowAnonymous attribute help us to bypass the authentication for any action method or controller(for all action methods within controller).

Client App in Angular 6

Asp.net core Angular Template creates an Angular app in our project under “ClientApp” folder. Alternatively, we can create an Angular App using Angular CLI (https://cli.angular.io/). To configure SPA (Single Page Application) with angular and ASP.net core web API, we need to register an SPA service in configuring method of startup class and supply source path. Also, during the development, we need to run the angular command to start the angular application. The commands that are used during start app can be defined in the package.json file. Using Following code, we can configure SPA.

Startup.cs

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

…..

…..

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

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

});

app.UseSpa(spa =>

{

// To learn more about options for serving an Angular SPA from ASP.NET Core,

// see https://go.microsoft.com/fwlink/?linkid=864501

spa.Options.SourcePath = "ClientApp";

if (env.IsDevelopment())

{

spa.UseAngularCliServer(npmScript: "start");

}

});

}

I have created following code structure for Angular application. Here, I have created all client code under “scr\app” folder. I have created Login component that take user credential and with help of authorization service, it stored JWT in HTML 5 local storage.

Authentication Service

Path \ClientApp\src\app\Services\authentication.service.ts

Authentication Service is responsible to provide login and logout feature to the application. This service post user credential to the web API for Login and check the response. If the response is successful, so response contains the JSON web token and token details are stored in local storage. If the response is not successful, it returns the error which is display by the login component. The local storage shared the data between browser sessions and data also preserve in local storage event if refresh the browser. Alternatively, we can store token information within windows session storage. The session storage cannot share the information between browser sessions. For Logout, it simply removes the token from local storage

import { Injectable } from '@angular/core';

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

import { map } from 'rxjs/operators';

@Injectable()

export class AuthenticationService {

constructor(private http: HttpClient) { }

login(username: string, password: string) {

return this.http.post<any>('/api/login', { username, password })

.pipe(map(user => {

// login successful if there's a jwt token in the response

if (user && user.token) {

// store user details and jwt token in local storage to keep user logged in between page refreshes

localStorage.setItem('TokenInfo', JSON.stringify(user));

}

return user;

}));

}

logout() {

// remove user from local storage to log user out

localStorage.removeItem('TokenInfo');

}

}

Login Template and component

Path \ClientApp\src\app\Login\

The login component template contains username and password fields. It also shows the validation for invalid fields when you will click submit button. The submit event of the form is bound to "OnLogin" method of login component. The "OnLogin" method uses the authentication service’s “login” method to validate user credential and generate token.

Login Template

<h2>Login</h2>

<form [formGroup]="loginForm" (ngSubmit)="onLogin()">

<div class="form-group">

<label for="username">Username</label>

<input type="text" formControlName="username" class="form-control" [ngClass]="{ 'is-invalid': submitted && formData.username.errors }" />

<div \*ngIf="submitted && formData.username.errors" class="invalid-feedback">

<div \*ngIf="formData.username.errors.required">Username is required</div>

</div>

</div>

<div class="form-group">

<label for="password">Password</label>

<input type="password" formControlName="password" class="form-control" [ngClass]="{ 'is-invalid': submitted && formData.password.errors }" />

<div \*ngIf="submitted && formData.password.errors" class="invalid-feedback">

<div \*ngIf="formData.password.errors.required">Password is required</div>

</div>

</div>

<div class="form-group">

<button [disabled]="submitClick" class="btn btn-primary">Login</button>

</div>

<div \*ngIf="error" class="alert alert-danger"></div>

</form>

Login Component

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

import { Router, ActivatedRoute } from '@angular/router';

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

import { first } from 'rxjs/operators';

import { AuthenticationService } from '../Services/authentication.service';

@Component({

templateUrl: 'login.html'

})

export class LoginComponent implements OnInit {

loginForm: FormGroup;

submitClick = false;

submitted = false;

returnUrl: string;

error = '';

constructor(

private formBuilder: FormBuilder,

private route: ActivatedRoute,

private router: Router,

private authenticationService: AuthenticationService) { }

ngOnInit() {

this.loginForm = this.formBuilder.group({

username: ['', Validators.required],

password: ['', Validators.required]

});

// reset login status

this.authenticationService.logout();

// get return url from route parameters or default to '/'

this.returnUrl = this.route.snapshot.queryParams['returnUrl'] || '/';

}

// convenience getter for easy access to form fields

get formData() { return this.loginForm.controls; }

onLogin() {

this.submitted = true;

// stop here if form is invalid

if (this.loginForm.invalid) {

return;

}

this.submitClick = true;

this.authenticationService.login(this.formData.username.value, this.formData.password.value)

.pipe(first())

.subscribe(

data => {

this.router.navigate([this.returnUrl]);

},

error => {

this.error = error;

this.submitClick = false;

});

}

}

Restrict Unauthenticated User to access the Application

I have created authorization check that prevent unauthenticated users from accessing restricted routes. Angular route provides the feature called Navigation Guards that allow us to write code before routing occurred. There are four different guard types available in Angular: CanActivate, CanActivateChild, CanDeactivate and CanLoad. Here I have check whether local storage contains the token information if yes than allow to navigate else navigate to login page.

Path:ClientApp\src\app\Services\authorizationCheck.ts

import { Injectable } from '@angular/core';

import { Router, CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable()

export class AuthorizationCheck implements CanActivate {

constructor(private router: Router) { }

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

//If token data exist, user may login to application

if (localStorage.getItem('TokenInfo')) {

return true;

}

// otherwise redirect to login page with the return url

this.router.navigate(['/login'], { queryParams: { returnUrl: state.url } });

return false;

}

}

Http Interceptor

After successful login to application, we need to pass token with every http request header. For the same we can write common code using Http Interceptor. It intercepts the http request and add the JWT (auth token) to the header of request if user is logged in. The TypeScript has allowed us to extend any predefined class, so we can create custom http interceptor by extending the HttpInterceptor class. This allow us to modify http request before it sent to the server. To register this Http interceptor, we need to add it to request pipeline by configure in the provider section of the app.module.ts file.

Path: ClientApp\src\app\Interceptor\httpInterceptor.ts

import { Injectable } from '@angular/core';

import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class httpInterceptor implements HttpInterceptor {

intercept(request: HttpRequest<any>, newRequest: HttpHandler): Observable<HttpEvent<any>> {

// add authorization header to request

//Get Token data from local storage

let tokenInfo = JSON.parse(localStorage.getItem('TokenInfo'));

if (tokenInfo && tokenInfo.token) {

request = request.clone({

setHeaders: {

Authorization: `Bearer ${tokenInfo.token}`,

'Content-Type': 'application/x-www-form-urlencoded;charset=utf-8'

}

});

}

return newRequest.handle(request);

}

}

Error Interceptor

I have added one more interceptor that check the errors from the http responses and re-throw the error. This interceptor analyses http response error and if the status code is a (401 Unauthorized response), user get logout automatically.

Path: ClientApp\src\app\Interceptor\errorInterceptor.ts

import { Injectable } from '@angular/core';

import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';

import { Observable } from 'rxjs/Rx';

import { catchError } from 'rxjs/operators';

import { AuthenticationService } from '../Services/authentication.service';

import { Router } from '@angular/router';

@Injectable()

export class ErrorInterceptor implements HttpInterceptor {

constructor(private authenticationService: AuthenticationService, private router: Router) { }

intercept(request: HttpRequest<any>, newRequest: HttpHandler): Observable<HttpEvent<any>> {

return newRequest.handle(request).pipe(catchError(err =>{

if (err.status === 401) {

//if 401 response returned from api, logout from application & redirect to login page.

this.authenticationService.logout();

}

const error = err.error.message || err.statusText;

return Observable.throw(error);

}));

}

}

App Module and Routing

The app module defines the root module of the application that contains the metadata about the module. We need to register all the interceptors, services, components and other modules within this app module. We can also define routing with-in app module.

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

import { RouterModule } from '@angular/router';

import { AppComponent } from './app.component';

import { NavMenuComponent } from './nav-menu/nav-menu.component';

import { HomeComponent } from './home/home.component';

import { CounterComponent } from './counter/counter.component';

import { FetchDataComponent } from './fetch-data/fetch-data.component';

import { LoginComponent } from './Login/login.component'

import { httpInterceptor } from './Interceptor/httpInterceptor';

import { ErrorInterceptor } from './Interceptor/errorInterceptor';

import { AuthorizationCheck } from './Services/authorizationCheck'

import { AuthenticationService } from './Services/authentication.service'

@NgModule({

declarations: [

AppComponent,

NavMenuComponent,

HomeComponent,

CounterComponent,

FetchDataComponent,

LoginComponent

],

imports: [

BrowserModule.withServerTransition({ appId: 'ng-cli-universal' }),

HttpClientModule,

FormsModule,

ReactiveFormsModule,

RouterModule.forRoot([

{ path: '', component: HomeComponent, pathMatch: 'full', canActivate: [AuthorizationCheck] },

{ path: 'counter', component: CounterComponent, canActivate: [AuthorizationCheck] },

{ path: 'fetch-data', component: FetchDataComponent, canActivate: [AuthorizationCheck] },

{ path: 'login', component: LoginComponent }

])

],

providers: [

{ provide: HTTP\_INTERCEPTORS, useClass: httpInterceptor, multi: true },

{ provide: HTTP\_INTERCEPTORS, useClass: ErrorInterceptor, multi: true },

AuthorizationCheck, AuthenticationService],

bootstrap: [AppComponent]

})

export class AppModule { }

Demo

To demonstration, I have mark SampleDataController with Authorize attribute so only authenticated user can able to access the method within this controller. Using “fetch-data” route, I am trying to access the page. As see in following snap, request header contains the authorization token and after successful authentication on server side, it returns weather forecast data.

Summary

JWT (JSON Web Token) is most popular and open standard that allows transmitting data between parties as a JSON object in a secure and compact way. Following are some summary of this article. Dot Net Tricks Angular online course aligned with real-time which is delivered by Google GDE and Microsoft MVP globally.

JWT allow us to do token-based authentication

help of the Angular guard, we can redirection for anonymous users to the login page by client-side code.

Angular interceptors help us to set authorization header to each http request and handle 401 unauthorized access status code

Using AllowAnonymous attribute, we can bypass the authorization at server side. This attribute can either applied to controller or individual action method

Controller or action method which mark with Authorize attribute, they allow only the