Token Based Authentication in Web API

Back to: ASP.NET Web API Tutorials For Begineers and Professionals

Token Based Authentication in Web API

In this article, I am going to discuss how to implement **Token Based Authentication in Web API** to secure the server resources with an example. Please read our previous article where we discussed how to implement **Client-Side HTTP Message Handler** with some examples. As part of this article, we are going to discuss the following pointers.

1. **Why do we need Token Based Authentication in Web API?**
2. **Advantages of using Token Based Authentication in ASP.NET Web API**
3. **How does the Token-Based Authentication work?**
4. **Implementing Token-Based Authentication in Web API.**
5. **Testing the Token Authentication using Postman.**

Why do we need Token Based Authentication in Web API?

The **ASP.NET Web API** is an ideal framework, provided by Microsoft that, to build **Web API's**, i.e. **HTTP** based services on top of the .NET Framework. Once we develop the services using Web API then these services are going to be consumed by a broad range of clients, such as

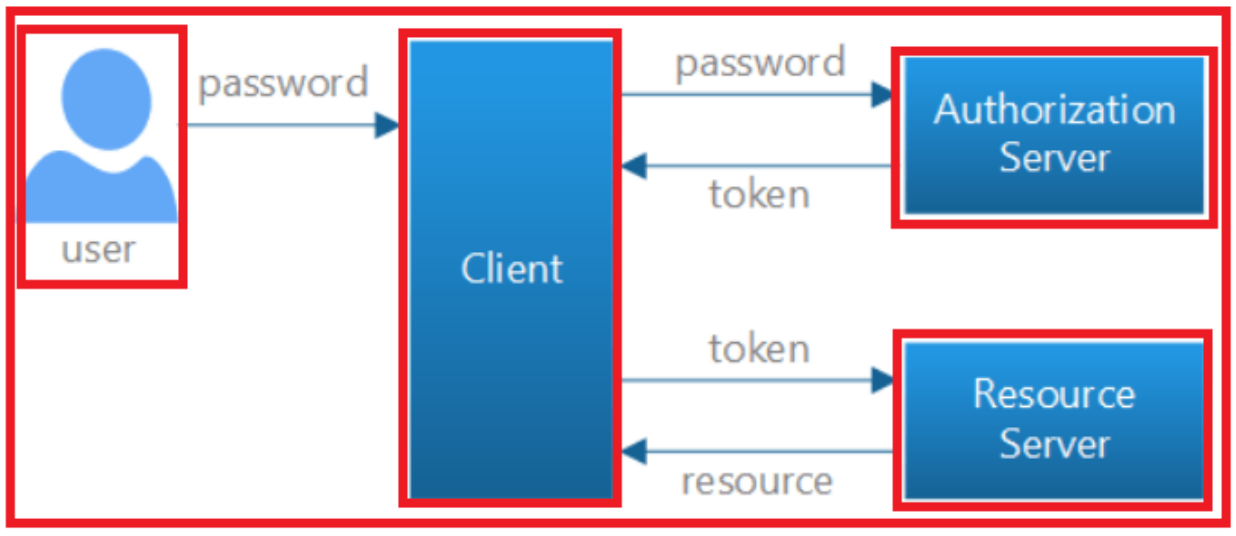
1. **Browsers**
2. **Mobile applications**
3. **Desktop applications**
4. **IOTs, etc.**

Nowadays, the use of WEB API is increasing in a rapid manner. So as a developer you should know how to develop Web APIs. Only developing Web APIs is not enough if there is no security. So, it also very important for us as a developer to implement security for all types of clients (such as Browsers, Mobile Devices, Desktop applications, and IoTs) who are going to use our Web API services.

The most preferred approach nowadays to secure the Web API resources is by authenticating the users in Web API server by using the **signed token** (which contains enough information to identify a particular user) which needs to be sent to the server by the client with each and every request. This is called the **Token-Based Authentication** approach.

How does the Token-Based Authentication work?

In order to understand how the token based authentication works, please have a look at the following diagram.



The Token-Based Authentication works as Follows:

1. The user enters his credentials (i.e. the username and password) into the client (here client means the browser or mobile devices, etc).
2. The client then sends these credentials (i.e. username and password) to the Authorization Server.
3. Then the Authorization Server authenticates the client credentials (i.e. username and password) and generates and returns an access token. This Access Token contains enough information to identify a user and also contains the token expiry time.
4. The client application then includes the **Access Token in the Authorization header** of the HTTP request to access the restricted resources from the Resource Server until the token is expired.

Note: If this not clear at the moment then don't worry, we will explain the above mentioned points one by one in detail with example.

Let's discuss the step by step procedure to implement Token-Based Authentication in Web API and then we will also how to use the token based authentication to access restricted resources using Postman and Fiddler.

Step1: Creating the required database

We are going to use the following **UserMaster** table in this demo.

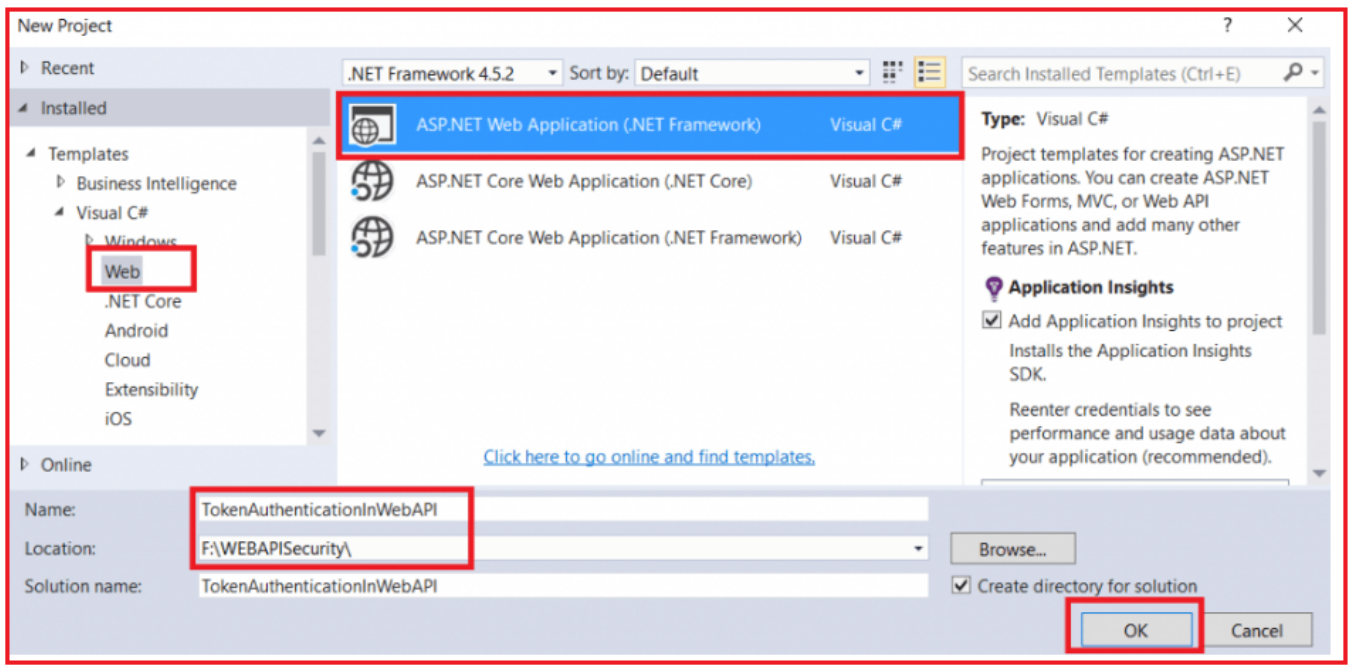
UserID	UserName	UserPassword	UserRoles	UserEmailID
101	Anurag	123456	Admin	Anurag@g.com
102	Priyanka	abcdef	User	Priyanka@g.com
103	Sambit	123pqr	SuperAdmin	Sambit@g.com
104	Pranaya	abc123	Admin, User	Pranaya@g.com

Please use below SQL Script to create and populate the UserMaster table with the required sample data.

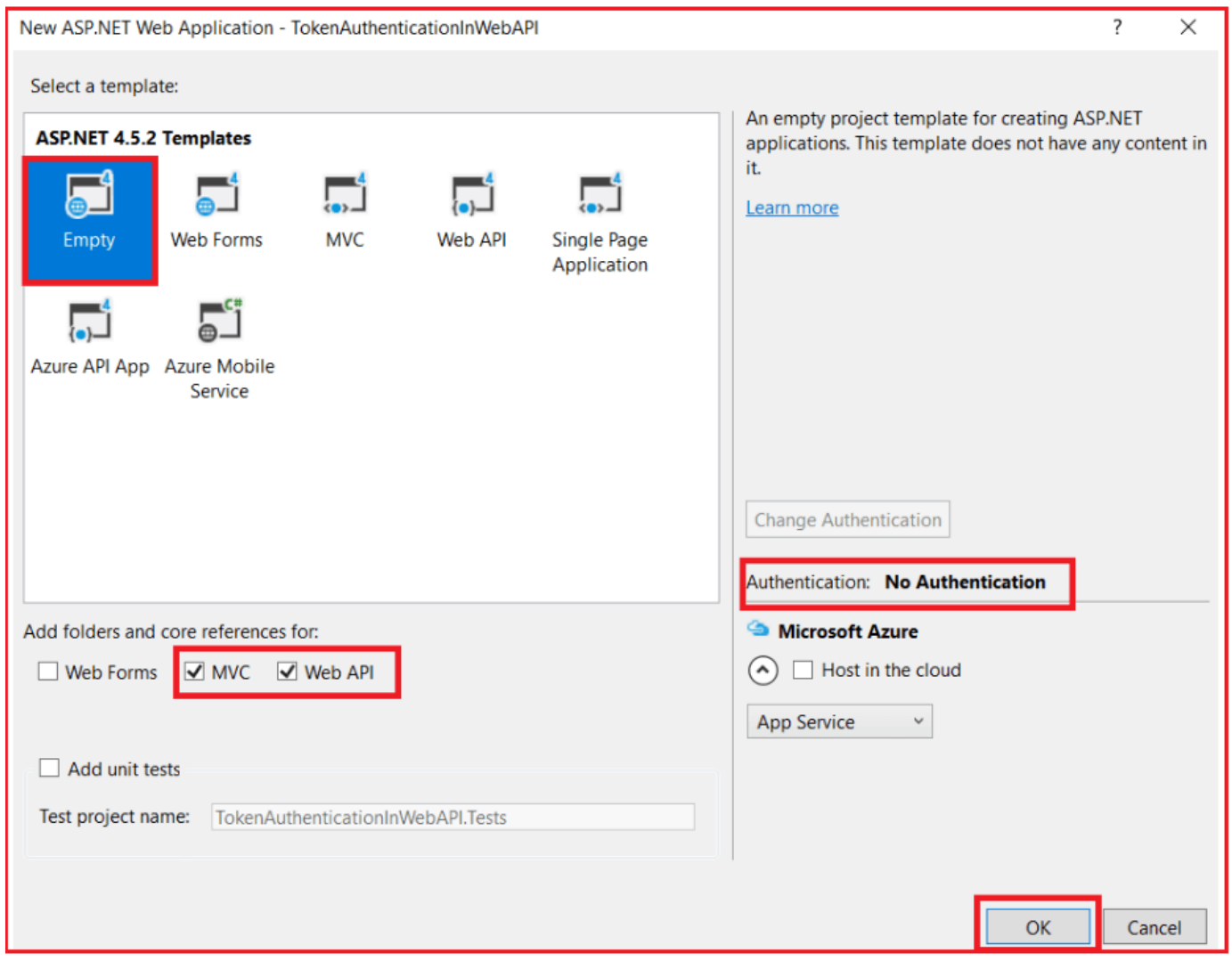
```
CREATE DATABASE SECURITY_DB
GO
USE [SECURITY_DB]
CREATE TABLE UserMaster
(
  UserID INT PRIMARY KEY,
  UserName VARCHAR(50),
  UserPassword VARCHAR(50),
  UserRoles VARCHAR(500),
  UserEmailID VARCHAR(100),
)
GO
INSERT INTO UserMaster VALUES(101, 'Anurag', '123456', 'Admin', 'Anurag@g.com')
INSERT INTO UserMaster VALUES(102, 'Priyanka', 'abcdef', 'User', 'Priyanka@g.com')
INSERT INTO UserMaster VALUES(103, 'Sambit', '123pqr', 'SuperAdmin', 'Sambit@g.com')
INSERT INTO UserMaster VALUES(104, 'Pranaya', 'abc123', 'Admin, User', 'Pranaya@g.com')
GO
```

Step2: Creating an empty Web API Project with the name TokenAuthenticationWEBAPI

Go to the **File menu > create > project** > here select "**asp.net web application**" under web. Provide the application name as **TokenAuthenticationWEBAPI** and select the project location where you want to create the project. Then click on the **OK** button as shown in the below image.



Once you click on the **OK** button, then a new window will open with Name **New ASP.NET Web Application** for selecting the **Project Templates** and from this window, you need to select the **Empty** project template as we are going to do everything from scratch and then checked the **MVC** and **Web API** checkbox from **Add folder and core references for** and then click on the **OK** button as shown in the below image.



Step3: Add the required references from NuGet packages into your application.

In order to Implement the **Token-Based Authentication in ASP.NET Web API**, we need to install the followings references from **NuGet packages.** Later part of this article, we will discuss the use of each the below packages.

1. **Microsoft.Owin.Host.SystemWeb**
2. **Microsoft.Owin.Security.OAuth**
3. **Microsoft.Owin.Cors**
4. **Newtonsoft.json**

For adding the above references from NuGet, Go to **Solution Explorer** > **Right Click on the References** > **Click on Manage NuGet Packages** > Search for the **Microsoft.Owin.Host.SystemWeb**, **Microsoft.Owin.Security.OAuth, Microsoft.Owin.Cors and Newtonsoft.json** and install.

Note: When you install the above packages the dependency references are also automatically installed into your application.

Step4: Creating the ADO.NET Entity Data Model

Here we are going to use the **DB First Approach of Entity Framework** to create the Entity Data Model against the **SECURITY_DB** database which we have already created and then select the **UserMaster** table from the **SECURITY_DB** database.

Step5: Create a Repository class

Now, you need to create a class with the name **UserMasterRepository** which will validate the user and also returns the user information. As you can see in the below code, the **ValidateUser** method takes the username and password as input parameter and then validate this. If the username and password valid then it will return UserMaster object else it will return null. Later we will discuss when and where we will use this method.

```
namespace TokenAuthenticationInWebAPI.Models
{
    public class UserMasterRepository : IDisposable
    {
        // SECURITY_DBEntities it is your context class
        SECURITY_DBEntities context = new SECURITY_DBEntities();
        //This method is used to check and validate the user credentials
        public UserMaster ValidateUser(string username, string password)
        {
            return context.UserMasters.FirstOrDefault(user =>
            user.UserName.Equals(username, StringComparison.OrdinalIgnoreCase)
            && user.UserPassword == password);
        }
        public void Dispose()
        {
            context.Dispose();
        }
    }
}
```

Step6: Add a class for validating the user credentials asking for tokens.

Now we need to add a class with the name **MyAuthorizationServerProvider** into our application. Within that class, we need to write the logic for **validating the user credentials** and **generating the access token**.

We need to inherit the **MyAuthorizationServerProvider** class from **OAuthAuthorizationServerProvider** class and then need to override the **ValidateClientAuthentication** and **GrantResourceOwnerCredentials** method. So, before proceeding and overriding these two methods, let us first understand what exactly these methods are going to perform.

ValidateClientAuthentication Method:

The **ValidateClientAuthentication** method is used for validating the client application. For the sake of simplicity, we will discuss what is a client and how to validate a client in more details in the next article.

GrantResourceOwnerCredentials Method:

The **GrantResourceOwnerCredentials** method is used to validate the client credentials (i.e. username and password). If it found the credentials are valid, then only it generates the access token. The client then using this access token can access the authorized resources from the Resource Server.

As we already discussed, the **signed access token** contains enough information to identify a user. Now the question is how. Let discuss this in details.

First, we need to create an instance of the **ClaimsIdentity** class and to the constructor of **ClaimsIdentity** class, we need to pass the **authentication type**. As we are going to use the **Token-Based Authentication**, so the Authentication Type is "**bearer token**".

Once we create the **ClaimsIdentity** instance, then need to add the claims such as **Role, Name**, and **Email**, etc to the ClaimsIdentity instance. These are the user information which is going to be included in the **signed access token**. You can add any number of claims and once you add more claims. the token size will increase.

MyAuthorizationServerProvider

Create a class file with the name **MyAuthorizationServerProvider.cs** and then copy and paste the following code in it.

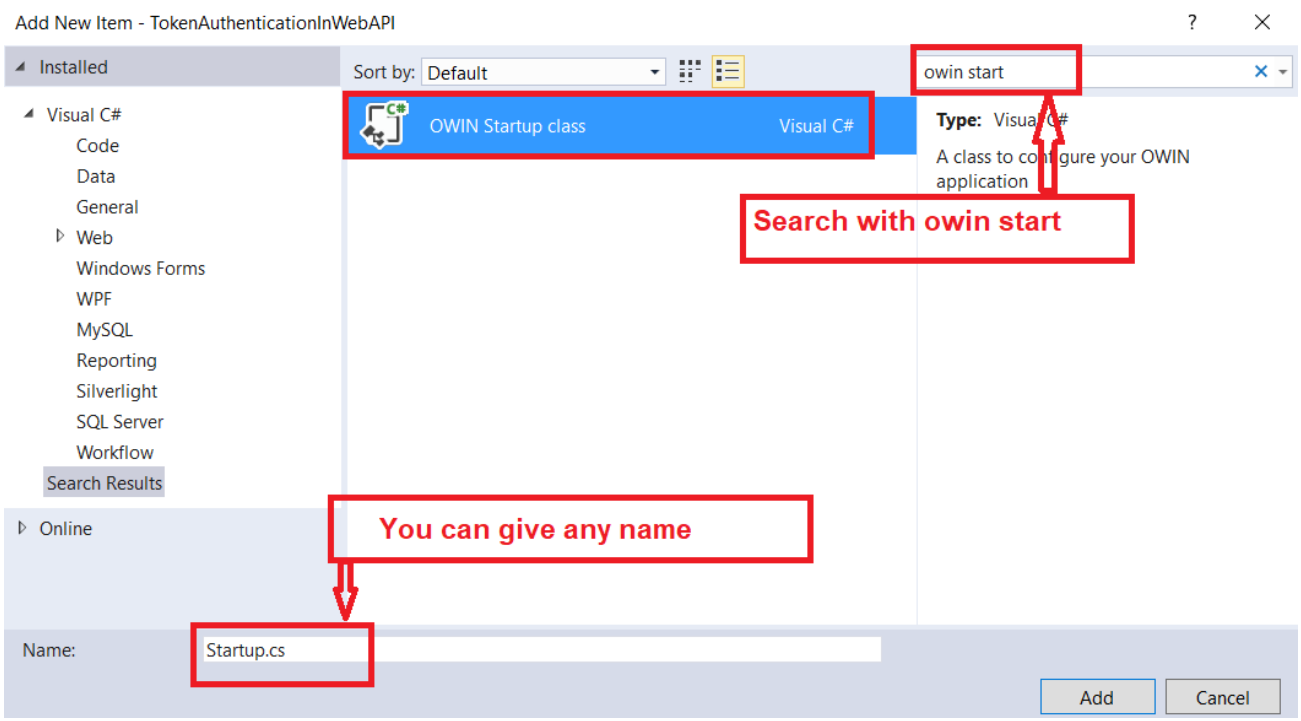
```
using Microsoft.Owin.Security.OAuth;
using System.Security.Claims;
using System.Threading.Tasks;
namespace TokenAuthenticationInWebAPI.Models
{
```

```
    public class MyAuthorizationServerProvider : OAuthAuthorizationServerProvider
    {
        public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext
context)
        {
            context.Validated();
        }
        public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext
context)
        {
            using (UserMasterRepository _repo = new UserMasterRepository())
            {
                var user = _repo.ValidateUser(context.UserName, context.Password);
                if (user == null)
                {
                    context.SetError("invalid_grant", "Provided username and password is incorrect");
                    return;
                }
                var identity = new ClaimsIdentity(context.Options.AuthenticationType);
                identity.AddClaim(new Claim(ClaimTypes.Role, user.UserRoles));
                identity.AddClaim(new Claim(ClaimTypes.Name, user.UserName));
                identity.AddClaim(new Claim("Email", user.UserEmailID));
                context.Validated(identity);
            }
        }
    }
}
```

Step7: Add the OWINStartup class.

Now we need to add the **OWINStartup** class where we will configure the **OAuth Authorization Server**. This is going to be our authorization server.

To do so, go to the **Solution Explorer** > **Right Click on Project Name** form the Solution Explorer > **Add > New Item** > Select **OWIN Startup class** > Enter the class name as **Startup.cs** > and then click on the **Add** button as shown in the below image.



Once you created the Owin Startup class, copy and paste the below code in it.

```
using System;
using Microsoft.Owin;
using Owin;
using TokenAuthenticationInWebAPI.Models;
using Microsoft.Owin.Security.OAuth;
using System.Web.Http;
[assembly: OwinStartup(typeof(TokenAuthenticationInWebAPI.App_Start.Startup))]
namespace TokenAuthenticationInWebAPI.App_Start
{
    // In this class we will Configure the OAuth Authorization Server.
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Enable CORS (cross origin resource sharing) for making request using browser from different
domains
            app.UseCors(Microsoft.Owin.Cors.CorsOptions.AllowAll);

            OAuthAuthorizationServerOptions options = new OAuthAuthorizationServerOptions
            {
                AllowInsecureHttp = true,
                //The Path For generating the Toekn
```

```
                TokenEndpointPath = new PathString("/token"),
                //Setting the Token Expired Time (24 hours)
                AccessTokenExpireTimeSpan = TimeSpan.FromDays(1),
                //MyAuthorizationServerProvider class will validate the user credentials
                Provider = new MyAuthorizationServerProvider()
            };
            //Token Generations
            app.UseOAuthAuthorizationServer(options);
            app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());

            HttpConfiguration config = new HttpConfiguration();
            WebApiConfig.Register(config);
        }
    }
}
```

Understanding the Owin Startup class code:

Here we created a new instance of the **OAuthAuthorizationServerOptions** class and then set its options as follows:

1. Here, we set the path for generating the tokens as "**http://localhost:portnumber/token**". Later we will see how to issue an HTTP Post request to generate the access token.
2. We have specified the expiry time for the access token as **24 hours**. So if the user tried to use the same access token after 24 hours from the issue time, then this request will be rejected and **HTTP status code 401** will be returned.
3. We also specified the implementation on how to **validate the client credentials for users asking for the access tokens** in the custom class named **MyAuthorizationServerProvider**.

Finally, we passed the **options** to the extension method **UseOAuthAuthorizationServer** which will add the authentication middleware to the pipeline.

Step8: Add a Web API Controller.

Now we need to create Web API resources. To do so, add an empty Web API Controller, where we will add some action methods so that we can check the **Token-Based Authentication** is working fine or not.

Go to **Solution Explorer** > **Right click on the Controllers** folder > **Add > Controller > Select WEB API 2 Controller – Empty** > Click on the **Add** button. > Enter the controller name as **TestController.cs** > finally click on the **Add** button which will create the TestController.

Once you created the **TestController**, then copy and paste the following code.

```
using System.Linq;
using System.Security.Claims;
using System.Web.Http;
namespace TokenAuthenticationInWebAPI.Controllers
{
    public class TestController : ApiController
    {
        //This resource is For all types of role
        [Authorize(Roles = "SuperAdmin, Admin, User")]
        [HttpGet]
        [Route("api/test/resource1")]
        public IHttpActionResult GetResource1()
        {
            var identity = (ClaimsIdentity)User.Identity;
            return Ok("Hello: " + identity.Name);
        }
        //This resource is only For Admin and SuperAdmin role
        [Authorize(Roles = "SuperAdmin, Admin")]
        [HttpGet]
        [Route("api/test/resource2")]
        public IHttpActionResult GetResource2()
        {
            var identity = (ClaimsIdentity)User.Identity;
            var Email = identity.Claims
                      .FirstOrDefault(c => c.Type == "Email").Value;
            var UserName = identity.Name;

            return Ok("Hello " + UserName + ", Your Email ID is :" + Email);
        }
        //This resource is only For SuperAdmin role
        [Authorize(Roles = "SuperAdmin")]
        [HttpGet]
        [Route("api/test/resource3")]
        public IHttpActionResult GetResource3()
        {
            var identity = (ClaimsIdentity)User.Identity;
            var roles = identity.Claims
                        .Where(c => c.Type == ClaimTypes.Role)
                        .Select(c => c.Value);
            return Ok("Hello " + identity.Name + "Your Role(s) are: " + string.Join(",", roles.ToList()));
```

```
        }
    }
}
```

Here, in the above controller, we have created three resources as follows,

1. **/api/test/resource1** – This resource can be accessed by all three types of roles such as Admin, SuperAdmin, and User
2. **/api/test/resource2** – This resource can be accessed by the users who are having the roles Admin and SuperAdmin
3. **/api/test/resource3** – This resource can be accessed only by the users who are having the role SuperAdmin

To test this we are going to use a client tool called **Postman**. First, you need to run your Web API application. If you are new to Postman then please read the following where we discussed how to use Postman to test Web API rest services.

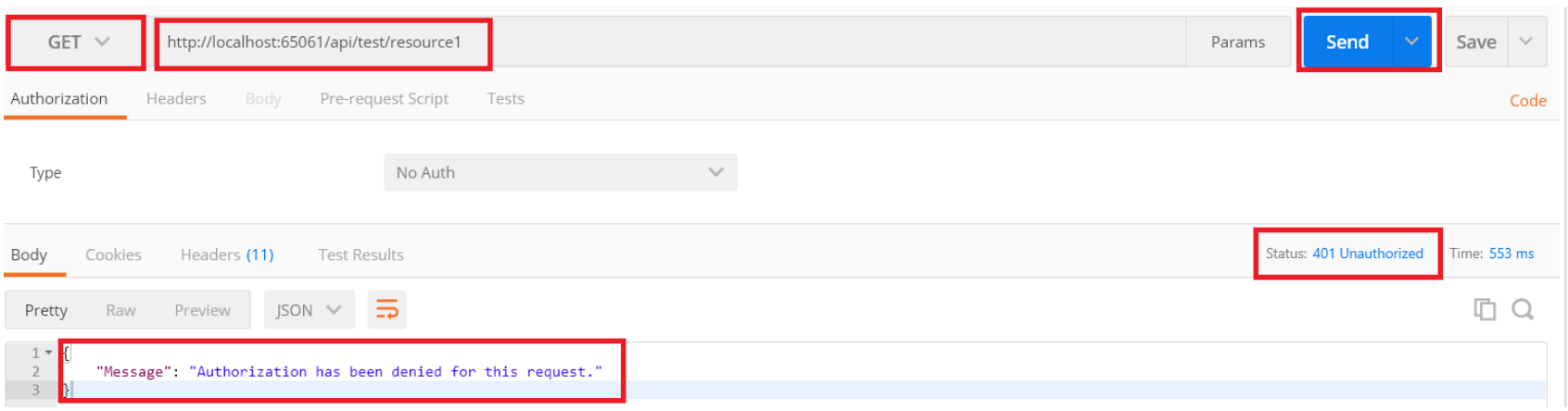
How to use Postman to test Rest Services?

Step9: Testing the Token Authentication

Test1: Without Access Token, try to make a request for following URI

http://localhost:xxxxx/api/test/resource1

You need to change the port number. You have to provide the port number where your Web API application is running.



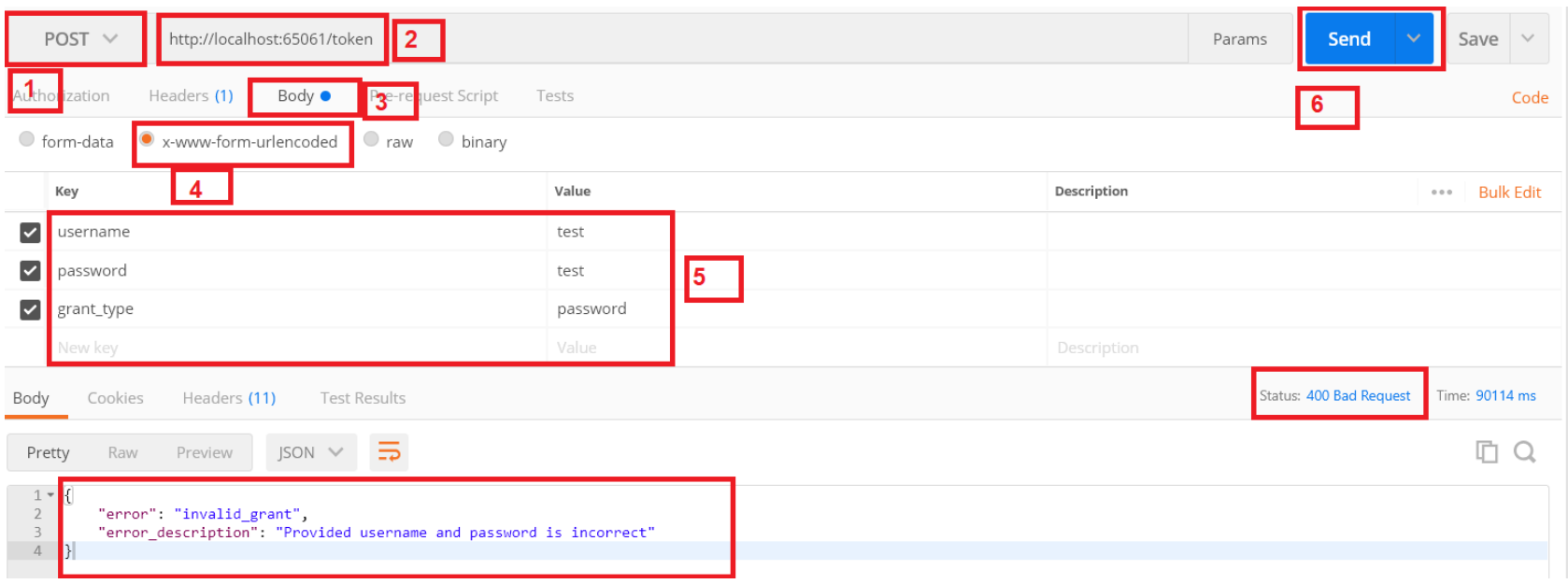
As expected you got 401 unauthorized responses

Test2: Try to create the Access token with invalid credentials

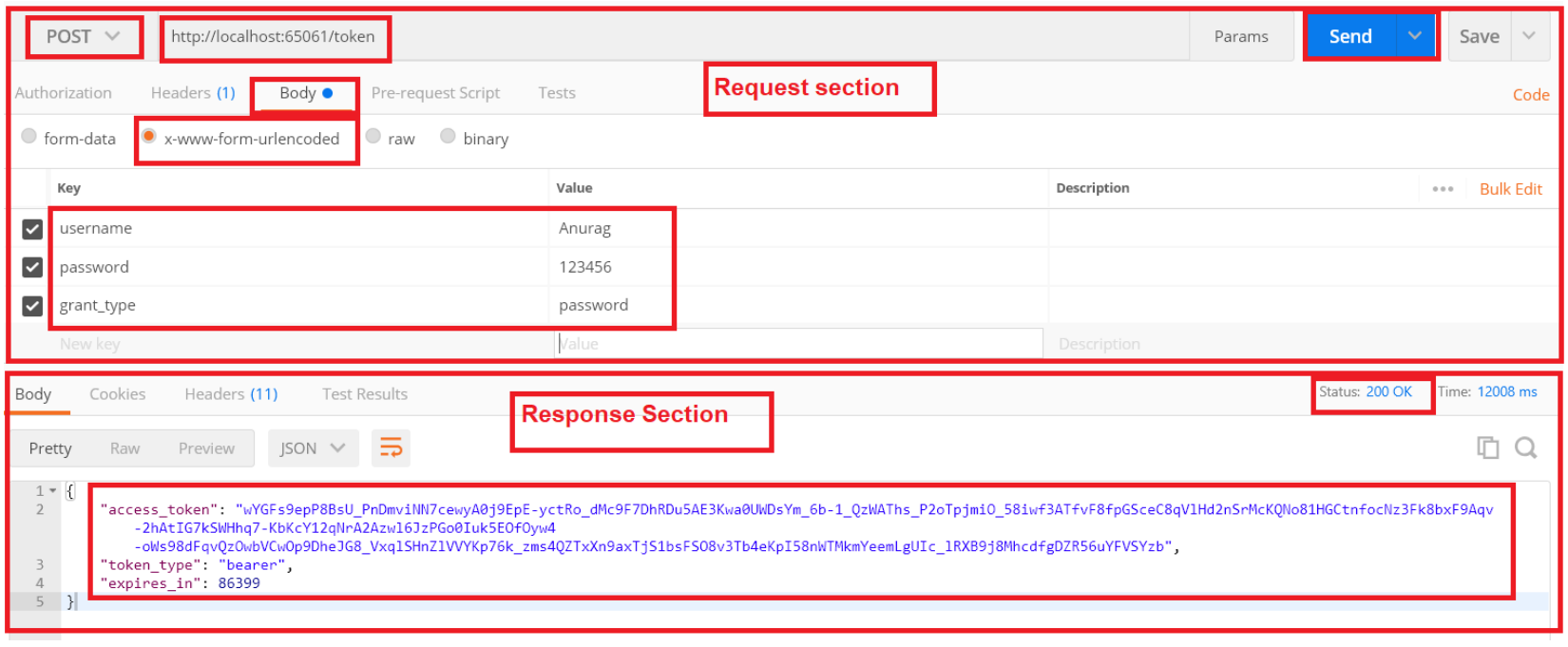
As we don't have any user with the name test, so let's try to create the Access Token for the test user. Select the method type as POST (1), enter the URL as **http://localhost:PortNumber/token** (2) and then click on body tab (3) and then select **x-www-form-urlencoded** (4) and then enter 3 parameters (5)

1. **username (value : test)**
2. **password (value: test)**
3. **grant_type (value: password)**

And then click on the Send button (6).



Once you click on the send button, you will get status as 400 Bad Request as expected and it also tells that in the error description that the provided username and password are incorrect. Let's generate the access token with valid credentials for the user Anurag whose password us 123456 as shown in the below image.



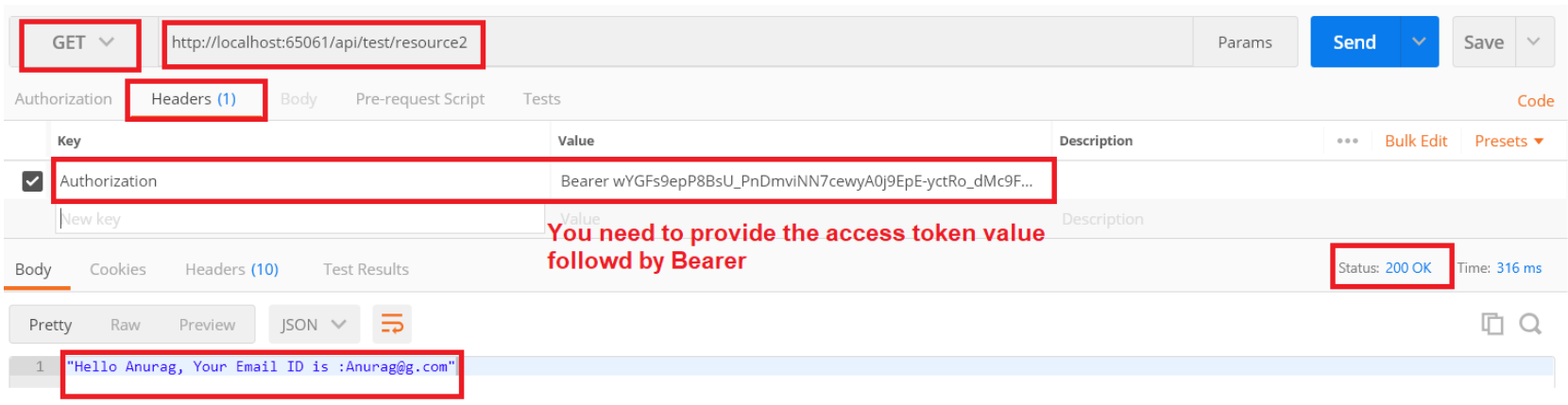
As you can see when you click on the send button, you will get status code 200 Ok along with you will get the access token which contains enough information to identify the user Anurag. You can also see in the Response section that the token type is Bearer and the token expire time in seconds.

Test3: Access restricted resources with the access token.

/api/test/resource2

First copy the access token that we just generated in the previous example that we are going to use the token as shown below.

Authorization: Bearer Access_Token(value)



You can see that, when you click on the Send button, you will get 200 Ok as expected because the resource **/api/test/resource2** has been accessed by the Roles Admin and SuperAdmin and here the user Anurag has the Role Admin so, we get the above response.

But the above user cannot access the resource **/api/test/resource3** because the resource3 can only be accessed by the user whose role is SuperAdmin. Let's prove this.

As you can see, the response is 401 unauthorized. But you generate the token for the user whose Role is SuperAdmin, then you can access this resource.

Let's have a look at the MyAuthorizationServiceProvider class

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

    public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)...
}
```

The first method i.e. ValidateClientAuthentication method is responsible for validating the Client, in this example, we assume that we have only one client so we'll always return that it is validated successfully. But in real-time you may have multiple clients and you need to validate the clients. So In the next article, we will discuss **how to use the ValidateClientAuthentication method to validate the client**.

Advantages of using Token Based Authentication in ASP.NET Web API:

Scalability of Servers:

The token which is sent to the server by the client is self-contained means it holds enough data to identify the user needed for authentication. As a result, you can add easily more servers to your web farm, there is no dependent on shared session stores.

Loosely Coupling:

The client application is not tied or coupled with any specific authentication mechanism. The token is generated, validated and perform the authentication are done by the server only.

Mobile-Friendly:

The Cookies and browsers like each other, but handling the cookies on native platforms like Android, iOS, Windows Phone is not an easy task. The token-based approach simplifies this a lot.

In the next article, I am going to discuss **Client Validation Using Basic Authentication in Web API** with an example. In this article, I try to explain how to implement **Token Based Authentication in Web API** with an example. I hope you enjoy this article.