

**WebAssembly**

WebAssembly (abbreviated *Wasm*) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

A Route is a URL pattern and Routing is a pattern matching process that monitors the requests and determines what to do with each request.

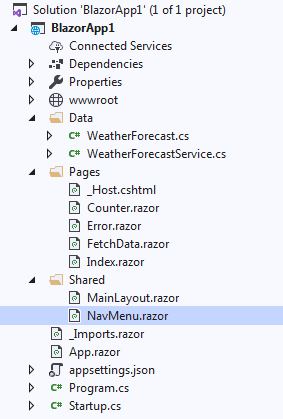
**@page directive**

Using @page directive, you can define the routing in Blazor component. The @page directives are internally converted into RouteAttribute when template is compiled.

In Blazor, Component can have multiple routes. If we require that component can render from multiple route values, we need to define all routes with multiple @page directives.

1. @page "/multiple"
2. @page "/multiple1"

|  |
| --- |
| 1. \_Host.cshtml (Master page of blazor app)  2. App.razor  3. MainLayout.razor  4. NavMenu.razor |



Flow of Application :-

1. Startup. cs

app.UseRouting();

|  |
| --- |
| app.UseEndpoints(endpoints =>  {  endpoints.MapBlazorHub();  endpoints.MapFallbackToPage("/\_Host");  }); |

* By convention, the *host* page is usually named *\_Host.cshtml*.
* The route specified in the host file is called a *fallback route* because it operates with a low priority in route matching.
* The fallback route is considered when other routes don't match

**Route templates**

* The Router component enables routing to each component with a specified route.
* The Router component appears in the *App.razor* file:

|  |
| --- |
| <**RouterAppAssembly**="@typeof(Program).Assembly">  <**FoundContext**="routeData">  <**RouteViewRouteData**="@routeData"**DefaultLayout**="@typeof(MainLayout)"/>  </**Found**>  <**NotFound**>  <**LayoutViewLayout**="@typeof(MainLayout)">  <p>Sorry, there's nothing at this address.</p>  </**LayoutView**>  </**NotFound**>  </**Router**> |

<NavLinkclass="nav-link"href=""Match="NavLinkMatch.All">

There are two NavLinkMatch options that you can assign to the Match attribute of the <NavLink> element:

* NavLinkMatch.All – The NavLink is active when it matches the entire current URL.
* NavLinkMatch.Prefix (default) – The NavLink is active when it matches any prefix of the current URL.

# Core Blazor layouts

Technically, a layout is just another component. A layout is defined in a Razor template or in C# code

|  |
| --- |
| @inheritsLayoutComponentBase  <divclass="sidebar">  <**NavMenu**/>  </div>  <divclass="main">  <divclass="top-row px-4">  <ahref="https://docs.microsoft.com/aspnet/"target="\_blank">About</a>  </div>  <divclass="content px-4">  @Body  </div>  </div> |

To turn a *component* into a *layout*, the component:

* Inherits from LayoutComponentBase, which defines a Body property for the rendered content inside the layout.
* Uses the Razor syntax @Body to specify the location in the layout markup where the content is rendered.

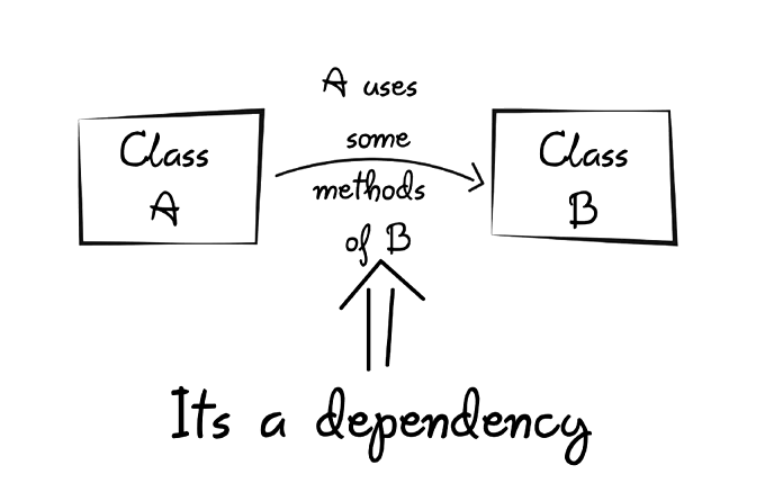
# Passing parameters in route

|  |  |
| --- | --- |
| |  | | --- | | * You can pass param using {} curly braces * Give multiple routing to prevent mandatory parameter passing * [Parameter] attribute is mandatory if using property as parameter in routing |   @page"/authors/authorDetails/{authorid}"  @page"/authors/authorDetails"  @page"/author"  <h3>AuthorDetails</h3>  <p>@AuthorID/ J.K rowling / harry potter</p>  @code{  [Parameter]  publicstringAuthorID { get; set; }  } |

**Passing data type parameters**

|  |
| --- |
| @page"/authors/authorDetails/{authorid}"  @page"/authors/authorDetails/{AuthorIDInt:int}"  @page"/authors/authorDetails"  @page"/author"  <h3>AuthorDetails</h3>  <p>@AuthorID/ J.K rowling string / harry potter</p>  <p>@AuthorIDInt/ book2 int/ book topic</p>  @code{  [Parameter]  publicstringAuthorID { get; set; }  [Parameter]  publicintAuthorIDInt { get; set; }  } |

**Dependency Injection**

****

**Steps :**

1. Add services.AddSingleton<Interface name, class implementing this interface>(); [startup.cs]
2. @injectIAuthorServiceIauthorService; in razor component

|  |  |
| --- | --- |
| @injectIAuthorServiceIauthorService;   |  | | --- | | Works like **class** - declare props,methods in it  Works like **constructor**:- starting point |   @code{  [Parameter]  publicintAuthorId { get; set; }  public Author AuthorDetails { get; set; }  protectedoverrideasync Task OnInitializedAsync()  {  AuthorDetails = awaitIauthorService.GetAuthorById(AuthorId);  }  } |

**FORMS AND VALIDATIONS**

<**EditFormModel**="@author"**OnValidSubmit**="@SaveAuthor">

<**InputText**class="form-control col-3"**@bind-Value**="author.FirstName"placeholder="FirstName"/>

USE <**DataAnnotationsValidator**/> to apply validations

|  |
| --- |
| <**EditFormModel**="@author"**OnValidSubmit**="@SaveAuthor">  <**DataAnnotationsValidator**/>  <**ValidationSummary**/>  <divclass="col-12 row">  <labelclass="col-2 font-weight-bold">First name</label>  <**InputText**class="form-control col-3"**@bind-Value**="author.FirstName"placeholder="FirstName"/>  <**ValidationMessageFor**="@(() =>author.FirstName)"/>  <br/>  <divclass="col-12 row">  <spanclass="col-2"></span>  <inputtype="submit"class="form-control col-1 btnbtn-primary"value="Save"/>  <span>&nbsp;</span>  <inputtype="submit"class="form-control col-1 btnbtn-primary"value="Clear"/>  </div>  </**EditForm**> |

**To show validation error messages: -** <**ValidationSummary**/>

validation against particular field only :-

<**ValidationMessageFor**="@(() =>author.FirstName)"/>

**Add JavaScript in Razor component**

@injectIJSRuntimeJSRuntime;

//To call js function from js file reference to which u can include in \_Host.cshtml

awaitJSRuntime.InvokeVoidAsync("saveMessage", firstname, lastname);

//best method to assign default values

protectedoverrideasync Task OnAfterRenderAsync(boolfirstRender)

{

if (firstRender&& Cities == null)

{

Cities = awaitJSRuntime.InvokeAsync<string[]>("getCities");

}

StateHasChanged();

}

**Components**

Two types of components:-

* 1. Pages component : - Having Routing at top with @Directive
  2. Shareable component :- No directives at top , so these can be nested with others

Example of child reusable shareable component

Component name: -**Servervalidations.razor**

|  |
| --- |
| @if (IsVisible)  {  <divid="divValidationMessage"class="col-10 alert alert-danger row">  <divclass="col-11"style="text-align:left">  <span>@ChildContent has been saved successfully</span>  </div>  </div>  }  @code{  [Parameter]  publicboolIsVisible { get; set; }  [Parameter]  publicRenderFragmentChildContent { get; set; }  } |

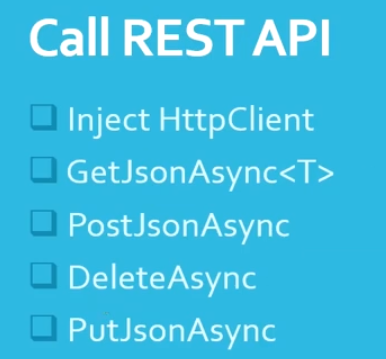
Parent component :-**Test.razor**

|  |
| --- |
| <**ServerValidationsIsVisible**="IsVisible">  <span>Authors : </span>@RecordName  </**ServerValidations**>  @code{  publicboolIsVisible { get; set; }  publicstringRecordName { get; set; }  privateasync Task SaveAuthor()  {  IauthorService.SaveAuthor(author);  IsVisible = true;  varfirstname = author.FirstName;  varlastname = author.LastName;  RecordName = firstname + " " + lastname;  author = new Author();  }  } |

**Razor components - Lifecycle methods**

****

****

****

**Steps :-**

1. Add service services.AddSingleton<HttpClient>(); in startup.cs

2. @inject HttpClient http; in razor component

3. authorList = await http.GetJsonAsync<List<Author>>("http://localhost:50973/api/authors");

Get your response from web api

Note: make sure properties name are same as json properties name

Add **MatBlazor** nuget to include **GetJsonAsync** in HttpClient

//Get

|  |
| --- |
| authorList = await Http.GetJsonAsync<List<Author>>("http://localhost:50973/api/authors");  authorList = authorList.OrderByDescending(auth => auth.AuthorId).ToList();  StateHasChanged(); |

**//POST**

|  |
| --- |
| if (author.AuthorId == 0)  await Http.PostJsonAsync("https://localhost:50973/api/authors", author);  else  await Http.PutJsonAsync("https://localhost:50973/api/authors/" + author.AuthorId,author);  await LoadAuthors(); |

**//PUT**

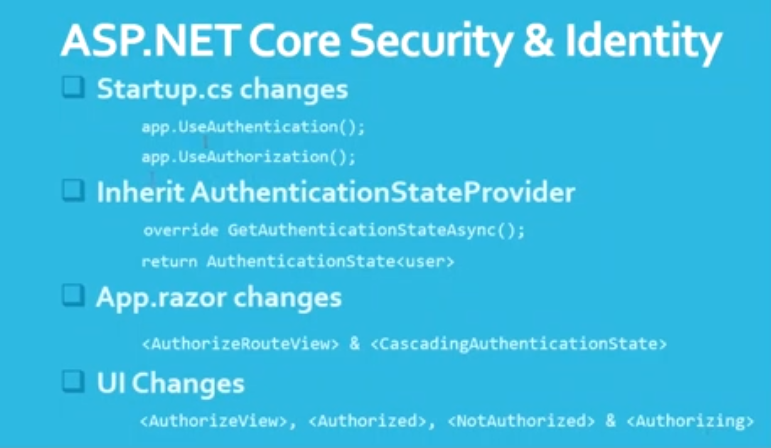
|  |
| --- |
| private void EditAuthor(Author argAuthor)  {  author = argAuthor;  }  Save()  {  if (author.AuthorId == 0)  await Http.PostJsonAsync("https://localhost:50973/api/authors", author);  else  await Http.PutJsonAsync("https://localhost:50973/api/authors/" + author.AuthorId, author);  await LoadAuthors();  } |

**//DELETE**

await Http.DeleteAsync("http://localhost:50973/api/authors/" + authorId);

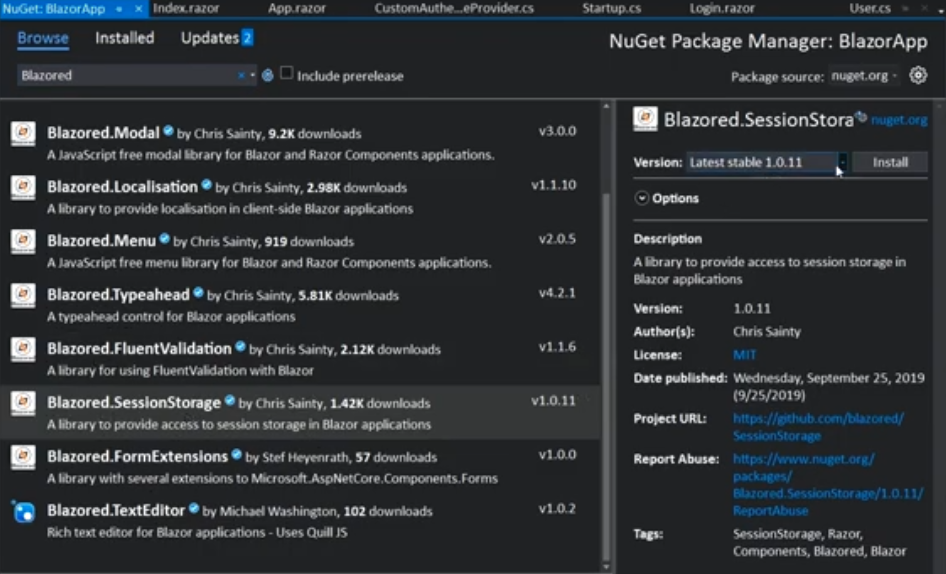
# 

# Authentication | Custom AuthenticationStateProvider

****

**To maintain Sessions/local storage in application**

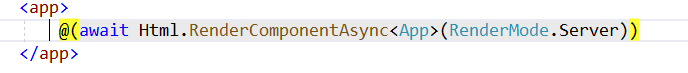
1.Import Blazored.SessionStorage nuget package

****

**2.Add service in ConfigureServices constructor**

****

**3. Add rendermode in \_Host.cshtml**

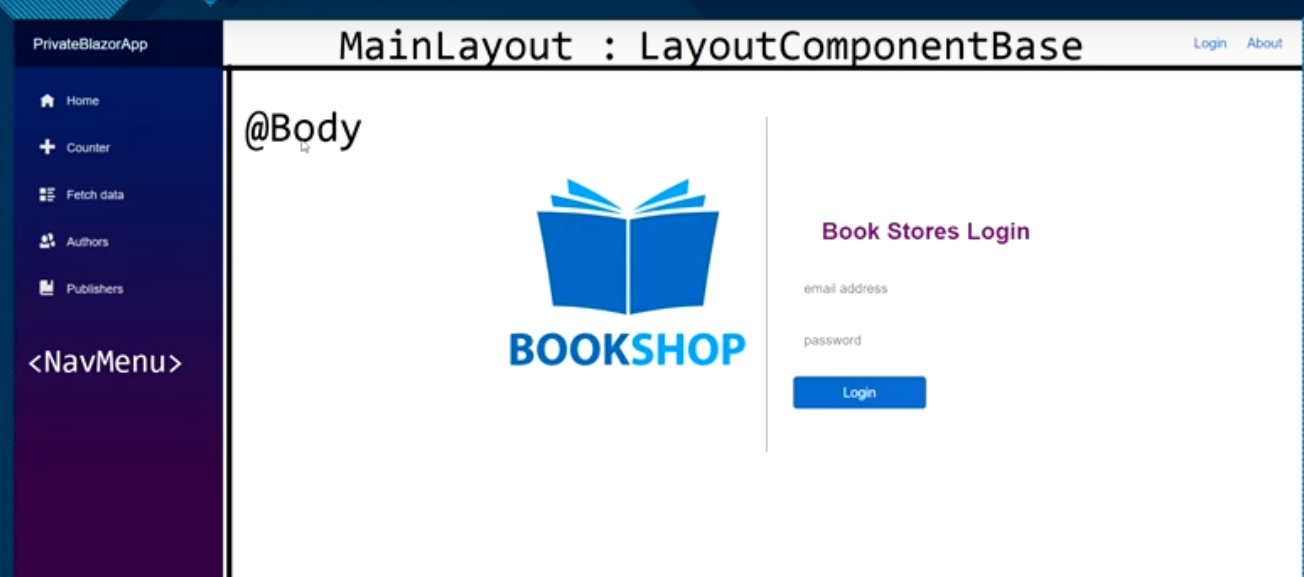
****

this session storage makes js calls and you need to have rendermode.server to allow js calls when using session storage

**4. Inject session storage service in Login page**

****

**LAYOUTS**

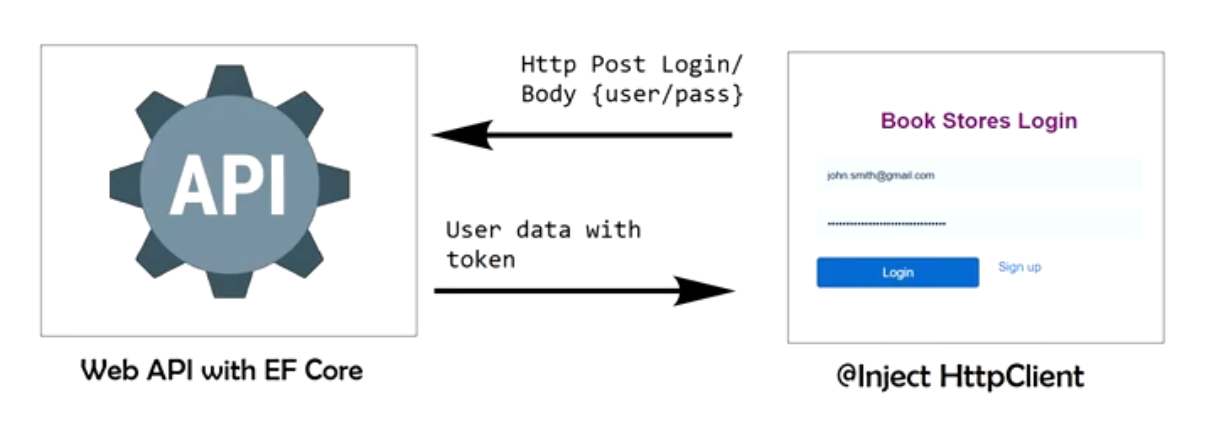
****

1. TO MAKE ANY PAGE AS LAYOUT PAGE,INHERIT LayoutComponentBase

//E.g: - LoginLayout.razor

|  |
| --- |
| @inherits LayoutComponentBase  <div class="col-12">  <br /><br />  @Body  </div> |

lazor HttpClient



Steps:-

1. Add Httpclient service

|  |
| --- |
| services.AddSingleton<HttpClient>(); |

2. Inject Httpclient in login page

|  |
| --- |
| @inject HttpClient Http |

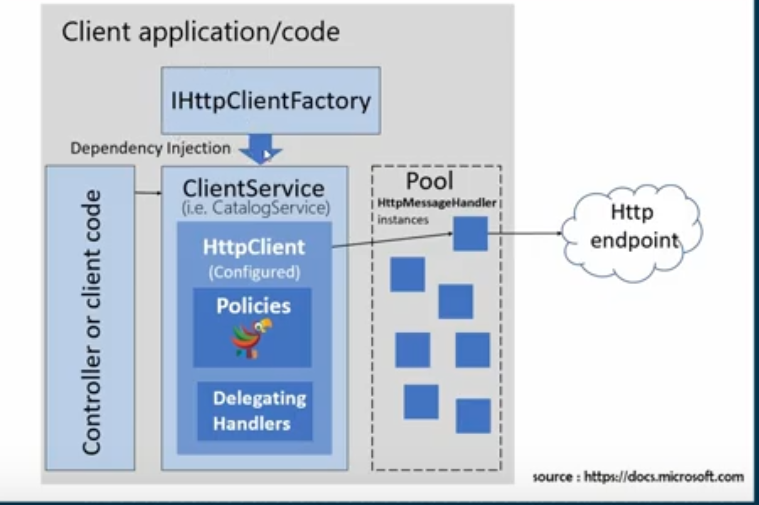
3. Serialize/deserialize with httpclient

|  |
| --- |
| private async Task<bool> ValidateUser()  {  string serialzedUser = JsonConvert.SerializeObject(user);  HttpRequestMessage httpRequestMessage = new HttpRequestMessage();  httpRequestMessage.Method = new HttpMethod("POST");  httpRequestMessage.RequestUri = new Uri("");  httpRequestMessage.Content = new StringContent(serialzedUser);  httpRequestMessage.Content.Headers.ContentType =  new System.Net.Http.Headers.MediaTypeHeaderValue("application/json");  var response = await Http.SendAsync(httpRequestMessage);  var responseStatusCode = response.StatusCode;  var responseBody = await response.Content.ReadAsStringAsync();  if (responseStatusCode.ToString() == "OK")  {  var returnedUser = JsonConvert.DeserializeObject<User>(responseBody);  await sessionStorage.SetItemAsync("emailAddress", user.EmailAddress);  await sessionStorage.SetItemAsync("token", user.Token);  ((CustomAuthenticationStateProvider)AuthenticationStateProvider)  .MarkUserAsAuthenticated(user.EmailAddress);  NavigationManager.NavigateTo("/index");  }  else  {  LoginMesssage = "Invalid username or password";  }  return await Task.FromResult(true);  } |

**IHttpClientFactory**

**Cons of using HttpClient :-**

1. Socket exhaustion
2. DNS name issues

****

**Steps :-**

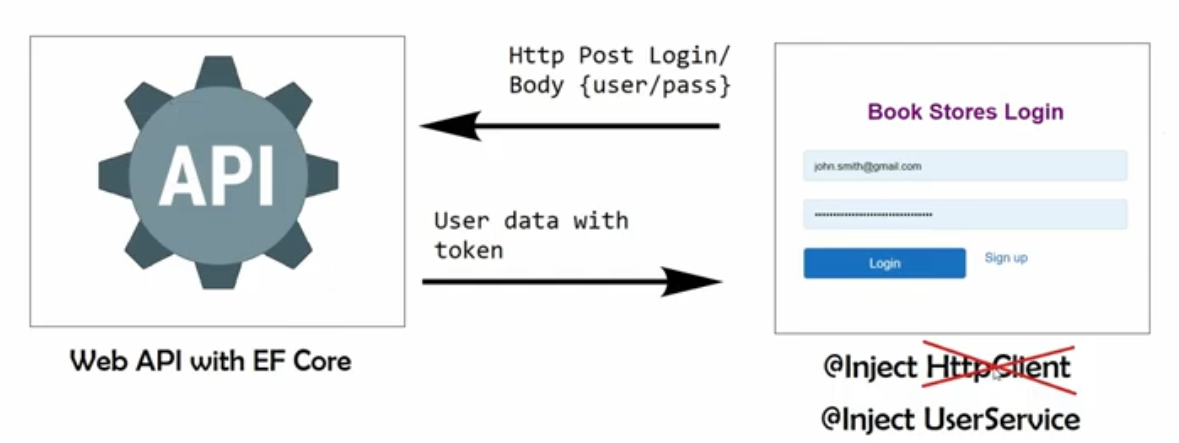
**1.[startup.cs]**

//To use httpClientfactory

services.AddHttpClient();

2.[Login.razor]

@inject IHttpClientFactory httpClientFactory

****

**<LOGIN.RAZOR>**

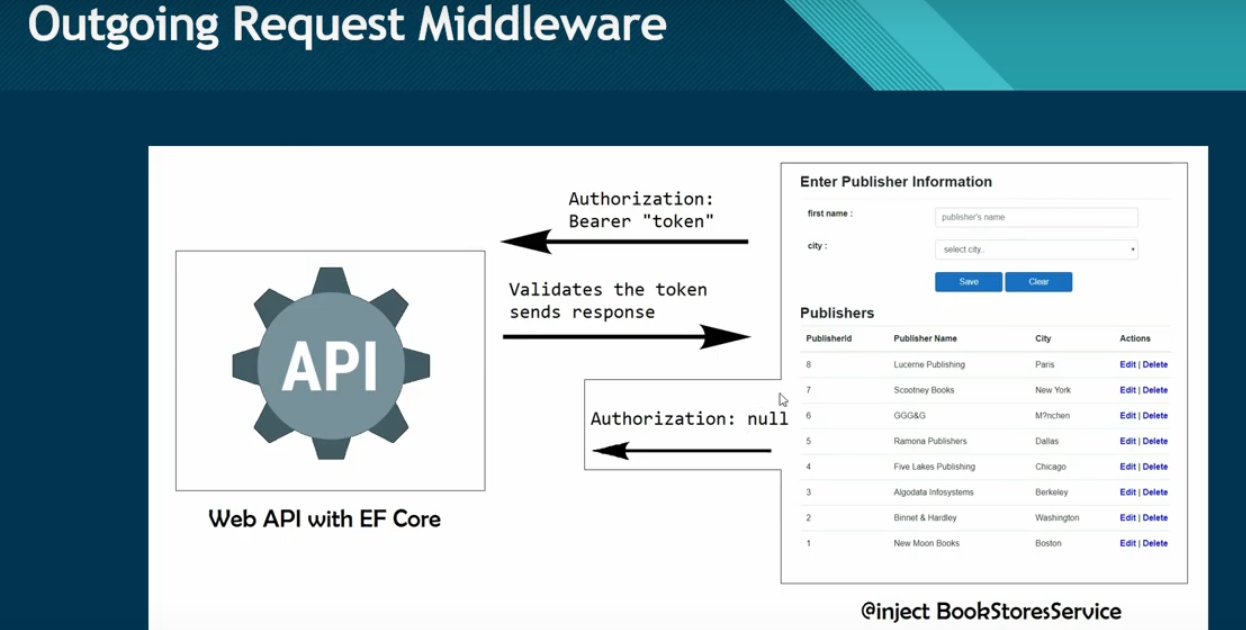
|  |
| --- |
| @inject IHttpClientFactory httpClientFactory  @inject IUserService userService  // SUBMIT LOGIN - CALL AN API [HTTPCLIENTfactory] - LOGIN.razor  private async Task<bool> ValidateUser()  {  var returnedUser = await userService.LoginAsync(user);  if (returnedUser.EmailAddress != null)  {  await sessionStorage.SetItemAsync("emailAddress", returnedUser.EmailAddress);  await sessionStorage.SetItemAsync("token", returnedUser.Token);  ((CustomAuthenticationStateProvider)AuthenticationStateProvider).MarkUserAsAuthenticated(user.EmailAddress);  NavigationManager.NavigateTo("/index");  }  else  {  LoginMesssage = "Invalid username or password";  }  return await Task.FromResult(true);  } |

**install certificate cmd :-**

**dotnet dev-certs https --trust**

**<USERSERVICE.CS**>

|  |
| --- |
| public class UserService : IUserService  {  public HttpClient \_httpClient { get; }  public async Task<User> LoginAsync(User user)  {  string serializedUser = JsonConvert.SerializeObject(user);  var requestMessage = new HttpRequestMessage(HttpMethod.Post, "Users/Login");  requestMessage.Content = new StringContent(serializedUser);  requestMessage.Content.Headers.ContentType  = new System.Net.Http.Headers.MediaTypeHeaderValue("application/json");  var response = await \_httpClient.SendAsync(requestMessage);  var responseStatusCode = response.StatusCode;  var responseBody = await response.Content.ReadAsStringAsync();  var returnedUser = JsonConvert.DeserializeObject<User>(responseBody);  return await Task.FromResult(returnedUser);  }    } |

****

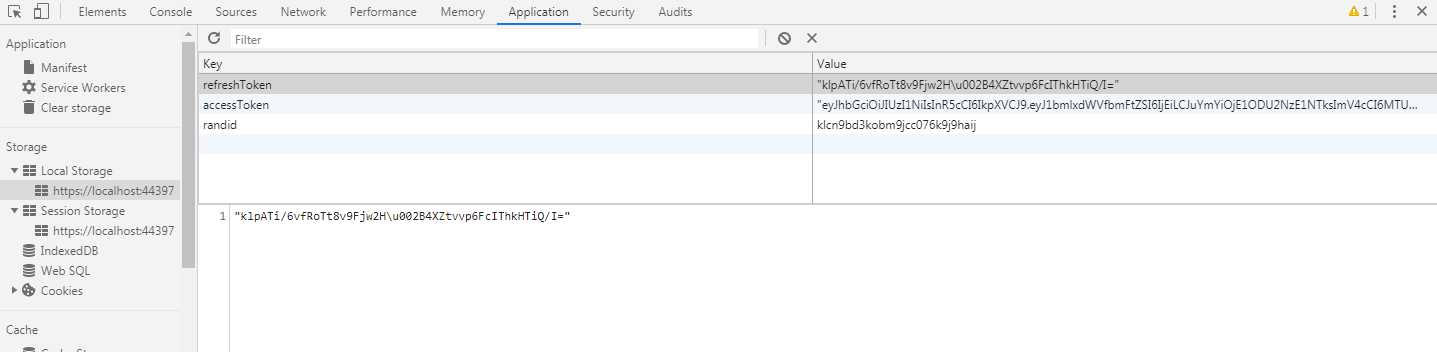
**Using local storage to store tokens :-**

1. Add Blazored.localstorage nuget pkg.

2. Add service in <Startup.cs>

|  |
| --- |
| //ADD LOCAL STORAGE SERVICE USING BLAZORED.LOCALSTORAGE NUGET PACKAGE  services.AddBlazoredLocalStorage(); |

Tokens added local storage can be checked as:-



# Role-based Authorization

Add role attribute in .razor page

//publisher.razor

@attribute [Authorize(Roles = "publishers")]

|  |
| --- |
| <**AuthorizeView** **Roles**="publishers">  <**Authorized**>  <th>Action</th>  </**Authorized**>  </**AuthorizeView**>  <**AuthorizeView** **Roles**="publishers">  <**Authorized**>  <th>  <a style="text-underline-position:below; cursor:pointer; color:blue" **@onclick**="(() => EditPublisher(publisher))">Edit</a> |  <a style="text-underline-position:below; cursor:pointer; color:blue" **@onclick**="(() => DeletePublisher(publisher.PubId))">Delete</a>  </th>  </**Authorized**>  </**AuthorizeView**> |

|  |
| --- |
| //index.razor  <**AuthorizeView** **Roles**="publishers">  <**Authorized**>  <p><b>you are a publisher</b></p>  </**Authorized**>  <**NotAuthorized**>  <p> <b>You are not a publisher!! </b></p>  </**NotAuthorized**>  </**AuthorizeView**> |