Contents

Blazor WebAssembly E-Commerce Website

Full Stack Foundations Building a Walking Skeleton

Create Blazor WebAssembly Project

- Search for Blazor WebAssembly project
- Give it a name
- Choose no authentication
    - We will do our own with json web tokens
- Choose .Net Core Hosted
    - Will give a nice folder structure
        - Will start with a Client project
        - Will start with a Server Project
            - Is the start file

        - Will Start with a Shared Project
            - Will hold the models

Trust the Dev Certificate

- In command line type dev-certs https –help
- Then type dotnet dev-certs –trust
- This is to trust the connection and will be needed for stripe checkout

Creating a component Server Side

- Components will be placed in the shared folder
- Right click and create a razor file.
    - To create a code behind file create a file that is named the same but ends in .cs
        - Ex: ProductList.razor.cs will be code behind for ProductList.razor
        - The code behind file should be a partial class and inherit from ComponentBase.
- To write c# code in the html use @ then followed by the code
    - Ex @foreach(var item in "prop/field in codebehind")
- To add css to just this component create a file with the same name but end it with .css
    - Ex ProductList.razor.css
- To use the component Go to a page and open angle brackets and write in the name
    - <ProductList />
    - You might need to add the directory to the _Imports.razor file if you put this file in a folder inside shared.

Injecting, Using parameters and implement an interface in a component

The methods differ between a code behind and a razor file on how to do these things.
For the razor file:

- Injecting:
    - At the top add @inject followed by the interface to inject
        - @inject IProductService
- Implement an interface to the code block in the razor file
    - At the top write @implments followed by the interface
        - @implements IDisposable

For the code behind file:

- Create a property and above it add [Inject}
    - [Inject]
    - public IProductSerivice? ProductSerivice { get; set; }
- Implementing an interface is the same as with a normal class
    - `public partial class Index : ComponentBase`

Using parameters is the same for the code block and codebehind

Create a property and add [Parameter above it.

- [Parameter]
- public string? CategoryUrl { get; set; } = null;

Component statemanagement

The Component have methods to execute when certain things happen in the component.

Ie OnParameterSet and OnParamterSetAsync you can decide what happens when a parameter is set for a component.

With OnInitialized and OnInitalizedAsync you can decide what happens when a component is first initialized.

Creating an API Controller and Adding Swagger

- Right click on controllers folder and select create controller
- Change from MVC Controller to API Controller
- Make sure You name it “Name”Controller and it inherits from ControllerBase
- At the top you can see the route api/[controller]
    - [controller] is what ever is before Controller in the class name
    - It can be changed to what ever.
- Make sure at the top under route it says [ApiController]
- When creating a method you need to specify which call it should answer to.
    - [HttpGet]
        - For all get request
    - [HttpPost]
        - For all post request
    - [HttpPut]
        - For put requests
    - [HttpDelete]
        - For Delete requests
- This annotations need to go above the method.
- To pass in a parameter with the url open a parantheses after the request and pass in a string.
    - Ex [HttpGet(“id”)
    - Now you can get the id from the url
    - Ex GetProductById(int id) this id will be the same as in the url
- To add swagger download nuget SwashBuckle.AspNetCore
- In program.cs add builder.Services.AddSwaggerGen();
- In the middleware after the app is build use app.UseSwaggerUI();
- After Enviroment set up use app.UseSwagger()
- Access swagger documentation at /swagger/index.html
- A http request need to return ActionResult<Class> to show as a schema in swagger

Set up Database Connection and installing EntityFramwork and enable CodeFirst and SQL Server

- The connection string should be in appsetting.json file
- Create a new section called ConnectionsStrings
- Open the body and add a property with the value of ýour connection string
- To work with ef core you need to install 3 packages in server project
    - EfCore
    - EfCore Design
    - EfCore SqlServer
- These packages will make sure evything works

- Another good package to install is EFCore Tools
    - It will make the commands in the packamanger console easier to work with
- Then create a data folder that will hold a class named DataContex, or what ever
- Make sure that class inherits from DbContext from ef core
- In it's constructor pass in DbCOntextOptions with the type of the context you just created and pass it upto the base class.
- The database tables will be representing by a dbSet so create on dbset for each model class that should be in the db.
- To seed data override the OnModelCreating method.
- Use modelBuilder.Enity<ModelClass>.HasData() to pass in the data you want to seed.
- In program.cs add a service to add the DbContext.
    - builder.Services.AddDbContext<DataContext>(opt =>
    - {
    - opt.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
    - });
- This will take the connection string from the appSettings file.

Best Practices

We will use thin controllers and not fat ones.

- Create a serviceResponse class that will be used By services
    - It should be a generic and take in the data, a success prop and a message incase of an error.
        - ```
          public class ServiceResponse<T> where T : class
          {
          public T? Data { get; set; }
          public bool Success { get; set; } = true;
          public string Message { get; set; } = string.Empty;
              }
          ```
- In server create a service folder that will contain all services
- Add a service class and add the methods you want.
    - Extract them to an interface
- Go to Program.cs
- Use builder.services.AddScoped to dependency inject the service class.
- In client do the same →Service Folder→Service class with methods → Exctract interface →register DI
- Registering DI in program.cs
    - `builder.Services.AddScoped<IProductSerivice, ProductSerivice>();`
- Example of service class in server side
    - ```
      public class ProductService : IProductService
      {
      private readonly DataContext _context;

      public ProductService(DataContext context)
      {
      _context = context;
      }
      public async Task<ServiceResponse<List<Product>>> GetProductsAsync(){
      ServiceResponse<List<Product>> response = new()
      {
      Data = await _context.Products.ToListAsync()
      ```

```
};
return response;
}
}
```

- Example of service in client

```
private readonly HttpClient _httpClient;

public List<Product> Products { get; set; } = new List<Product>();

public ProductSerivice(HttpClient httpClient)
{
_httpClient = httpClient;
}
public async Task GetProducts()
{
var result = await
_httpClient.GetFromJsonAsync<ServiceResponse<List<Product>>>("api/produ
ct");

if (result != null && result.Data != null && result.Success)
{
Products = result.Data;
}
}
```

Remember to add global using for some classes so you don't need to add them all the time.

It is done in program.cs files and they exists in both server and client.

Using local storeage

Install the nuget Blazored.LocalStorage.

Then in the places you want to access localStorage inject ILocalSotorageService.

Use the GetItem or GetItemAsync method and specify the return type and pass in the name of the storage.

```
var cart = await _localStorageService.GetItemAsync<List<CartItem>>("cartItems");
```

This will return a List of CartItems from the cartItems place in LocalStorage.

To insert into LocalStorage use SetItem or SetItemAsync, pass in the storage name and the object to store.

```
await _localStorageService.SetItemAsync("cartItems", cart);
```

Calling apis in the client side.

Whhen you want to call an api inject HttpClient. To get the items straight to objects use GetFromJsonAsync and specify the object.

```
var result = await
_httpClient.GetFromJsonAsync<ServiceResponse<Product>>($"api/product/{id}");
```

Result will contain a ServiceResponse of type Product in this call.

To post something you use Post methods. To convert an object straight to json use PostAsJsonAsync.

```
var response = await _httpClient.PostAsJsonAsync("api/cart/products", cartItems);
```

This will convert cartItems to a json string and send it to the uri.

Authentication with JSON Web Tokens

Creating a webToken

When logging in with a user a webtoken should be created.

To create one you need a method that takes in the user the token should represent.

Then create a list of claims that container ClaimTypes and pass in the user id and email and role as corresponding types.

```
List<Claim> claims = new List<Claim>
            {
                new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
                new Claim(ClaimTypes.Name, user.Email),
                new Claim(ClaimTypes.Role, user.Role)
            };
```

Create a key with SymmetricsSecurityKey class and send in a secret key, it is usually stored in appsettings and you need to have access to IConfiguration getSection to access it.

```
var key = new SymmetricSecurityKey(Encoding.UTF8
                .GetBytes(_configuration.GetSection("AppSettings:Token").Value));
```

Use the key when creating credentials with SigningCredentials that takes in a key and a security algorithm siugnature Use HmachSha512.

```
var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);
```

Now you can create a token with HwtSecurityToken class and pass in the claims as claims and set an expire date and pass in the credentials.

```
var token = new JwtSecurityToken(
                    claims: claims,
                    expires: DateTime.Now.AddDays(1),
                    signingCredentials: creds);
```

To get the token as a string use JwtSecurityTokenHandler and its writeToken method.

```
var jwt = new JwtSecurityTokenHandler().WriteToken(token);
```

Now when a user logs in return this token as a string so the client side can use it.

Enabling JWT auth server side for the controller

In program.cs you need to add Authentication service.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
```

```
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey =
                new SymmetricSecurityKey(System.Text.Encoding.UTF8

.GetBytes(builder.Configuration.GetSection("AppSettings:Token").Value)),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

And in the middle ware you need to add both authorization and authentication middleware.

```
app.UseAuthentication();
app.UseAuthorization();
```

Using the JWT Token Client side.

When logging in put the token you get from the response in to localstorage so it can be used else where.

Createing CustomAuthStateProvider

The AuthStateProvider will be the class that handles authorization and will be used by components to either authorize or deny access to a page.

The class needs to inherit from AuthenticationStateProvider which comes from a nuget called Microsoft.AspNetCOre.Components.Authorization.

Implement the base class method `GetAuthenticationStateAsync`.

In this method take the authToken from the localstorage and create a new Claims identy variable and set the httpCLients deafault Authorization header to null.

Check if auth token is empty if not try to create a new claimsIdentity by passing in the parsed value of the authToken. Then set the default request authorization header to a new Bearer token.

Then notify that the Auth state has been changed with the NotifyAuthenticationStateChanged method that takes in the state.

```
public override async Task<AuthenticationState> GetAuthenticationStateAsync()
        {
            // get token from local storage
            string authToken = await
_localStorageService.GetItemAsStringAsync("authToken");

            var identity = new ClaimsIdentity();
            // user is not authorizes as a default
            _httpClient.DefaultRequestHeaders.Authorization = null;

            // pars the claims and set new claims
            if (!string.IsNullOrWhiteSpace(authToken))
            {
                try
                {
                    identity = new ClaimsIdentity(ParsClaimsFromJwt(authToken),
"jwt");
                    // remove qoutation marks and set a new valid auth header.
                    _httpClient.DefaultRequestHeaders.Authorization = new
System.Net.Http.Headers.AuthenticationHeaderValue("Bearer",
authToken.Replace("\"", ""));
                }
```

```
                catch (Exception e)
                {
                    await _localStorageService.RemoveItemAsync("authToken");
                    identity = new ClaimsIdentity();
                }
            }
            var user = new ClaimsPrincipal(identity);
            var state = new AuthenticationState(user);

            NotifyAuthenticationStateChanged(Task.FromResult(state));

            return state;
        }
```

Parsing the jwt token

To parse the jwt token to a claim value you need two methods. One to parse it to base64 with out padding.

```
private byte[] ParseBase64WithOutPadding(string base64)
        {
            switch(base64.Length % 4)
            {
                case 2: base64 += "=="; break;
                case 3: base64 += "="; break;
            }
            return Convert.FromBase64String(base64);
        }
```

And one methods that firs splits the token by . and pass in the first index into the parseBase64 method. Then deserialize that result into a dictionary of strings and objects. Then create new claim with a select method on the dictionary.

```
  private IEnumerable<Claim> ParsClaimsFromJwt(string authToken)
        {
            var payload = authToken.Split(".")[1];
            var jsonBytes = ParseBase64WithOutPadding(payload);
            var keyValuePairs = JsonSerializer.Deserialize<Dictionary<string,
object>>(jsonBytes);

            var claims = keyValuePairs.Select(kv => new Claim(kv.Key,
kv.Value.ToString()));

            return claims;
        }
```

Enabling Authorization Client

IN program.cs add these three services.

```
builder.Services.AddOptions();
builder.Services.AddAuthorizationCore();
builder.Services.AddScoped<AuthenticationStateProvider,
CustomAuthStateProvider>();
```

Now in app.razor Instead of RouteView use AuthorizeRouteView.

```
<AuthorizeRouteView RouteData="@routeData"
DefaultLayout="@typeof(ShopMainLayout)" >
                <NotAuthorized>
```

```
                    <h3>Woops! You are not allowed to see this page</h3>
                    <h5>Please <a href="login">login</a> or <a
href="register">register</a> for a new account</h5>
                </NotAuthorized>
            </AuthorizeRouteView>
```

No you can use the NotAuthorized component that will be displayed when a user is not logged in.

Now in components that should display different data depending on if the user is Authorized or not you need to First use AuthorizeView as a wrapper of the content that should be displayed.

Then You can use Authorized or NotAuthorized components to wrap content depending on what you want.

```
    <AuthorizeView>
            <Authorized>
                 <a href="profile" class="dropdown-item ms-0">Profile</a>
                 <hr />
                <button class="dropdown-item" @onclick="Logout">Logout</button>
            </Authorized>
            <NotAuthorized>
                <a
href="login?returnUrl=@Navigation.ToBaseRelativePath(Navigation.Uri)"
class="dropdown-item ms-0">Login</a>
                <a href="register" class="dropdown-item ms-0">Register</a>
            </NotAuthorized>
        </AuthorizeView>
```

Here profile and logout will only be displayed if the user is logged in otherwise only register and login will be displayed.

To make a page private use the @attribute [authorize]

```
@page "/profile"
@attribute [Authorize]
```

Remember to put `@using Microsoft.AspNetCore.Components.Authorization;` `@using Microsoft.AspNetCore.Authorization; in import.razor to access the components and attributes.`

Adding Stripe Payment

Create an account at https://stripe.com/. Navigate to Dashboard.

Create an Project account. Make sure you are in test mode. Download stripe cli. It can only be run I cmd. Use stripe login to login to your account. When done use –forward-to “project local address”. This will make stripe send a post reques to the desired api endpoint and you can fulfill your order.

Now in the code project download nuget `Stripe.net.`

In a service method create a CheckOut sessions. A Session needs to have items so loop throught the products and add them to a list of SessionLineItemsOptions that is part of the Stripe nuget.

```
var lineItems = new List<SessionLineItemOptions>();
            products.ForEach(p => lineItems.Add(new()
            {
                PriceData = new()
                {
```

```
                    UnitAmountDecimal = p.Price * 100,
                    Currency = "usd",
                    ProductData = new()
                    {
                        Name = p.Title,
                        Images = new List<string> { p.ImageUrl }
                    }
                },
                Quantity = p.Quantity
            }));
```

Next create a SessionCreateOptions that should contain the user Email and Address options and the lineItems, payment method and two urls one for success and one for fail.

```
var options = new SessionCreateOptions
            {
                CustomerEmail = _authService.GetUserEmail(),
                ShippingAddressCollection =
                    new SessionShippingAddressCollectionOptions
                    {
                        AllowedCountries = new List<string> { "US","SE" }
                    },
                PaymentMethodTypes = new List<string>
                {
                    "card"
                },
                LineItems = lineItems,
                Mode = "payment",
                SuccessUrl = "https://localhost:7168/order-success",
                CancelUrl = "https://localhost:7168/cart"
            };
```

Now create a sessionsService which is part of the stripe nuget. Then Create a session which will be the result of the sessionSerive Create method where you pass in the options.

```
SessionService service = new();
            Session session = service.Create(options);
            return session;
```

Now we send back the session and you can navigate to the url provided by the session.

FullFillStripe payment

Make sure you run stripe cli and listen to an api endpoint to make the events fire.

Then create a method that should run when the event is fired.

```
var json = await new StreamReader(request.Body).ReadToEndAsync();
            try
            {
                var stripeEvent = EventUtility.ConstructEvent(
                    json,
                    request.Headers["Stripe-Signature"],
                    _configuration.GetSection("AppSettings:StripeWebHook").Value
                    );
                if(stripeEvent.Type == Events.CheckoutSessionCompleted)
                {
                    var session = stripeEvent.Data.Object as Session;
                    var user = await
_authService.GetUserByEmail(session.CustomerEmail);
                    await _orderService.PlaceOrder(user.Id);
                }
                return new ServiceResponse<bool> { Data = true };
```

```
            }
            catch (Exception e)
            {

                return new ServiceResponse<bool> { Data = false, Success = false,
Message = e.Message };
            }
        }
```