Creating Project and Setup

* Create New Project => Asp.NET Core MVC template => Authentication Type: Individual Accounts / Configure for HTTPS / Enable Razor runtime compilation
* Manage Nuget packages – install Microsoft.EntityFramework.Core, Microsoft.EntityFramework.Core.Tools, Microsoft.EntityFramework.Core.SqlServer
* Initial formatting of Program.cs and Startup.cs and aspsettings.json DefaultConnection
* Settings for usings to be in the deepest scope Tools > Options > Search:Code Editing > C# > Code Style > ‘using’ preference > Inside namespace
* Creating models in Data => Models folders => Car.cs etc
* Separating into projects – Add > New Project > Class Library C# and reference it

Creating Entities in Data => Models folder:

* [Required] attribute for strings / int is required by default unless with a ? (int? when not required)
* [MaxLength] – these 2 attributes are used for setting up the database
* Class DataConstants with public const int CarBrandMaxLength etc. When the constants become too many, the DataConstants class can be separated into nested classes public class Car{} Public class Category{} Public class Dealer{} etc.

Creating Migration:

* Down the VS window in the Package Manager Console
* Use command Add-Migration InitialCreate (or any other name as one word)
* Create extension for migrating the database every time the app is started in case any changes ( new folder in project “Infrastructure” => ApplicationBuilderExtention.cs static class with static method PrepareDatabase() and static SeedData()

**Controllers:**

* HomeController Setup and clear the \_Loyout.cshtml by removing privacy and adjusting title and DateTime.UtcNow.Year at the bottom
* CarsController – model binding (public IActionResult Add() and [HttpPost] public IActionResult Add(CarInputModel input) and model validation (if(!ModelState.IsValid){return this.View(input)}
* NB! If the CarInputModel is called ‘’model” and in the CarInputModel there is a property named “Model” there will be confusion – avoid naming the input with same name as any of its property!
* Model Binding – the names of the properties in the input model should match the names in the form (the .cshtml file) in order for the model binding to be successful. It looks through several data sources – form values, route values (values in the routing), query string (query string params in the URL), etc.
* CRUD in CarsController: actions All(), Add(), Delete(), Edit()
* Search – the search term goes as a parameter of the IActionResult All(string brand, string searchTerm)
* Get the data from the database var carsQuery = this.data.Cars.AsQueryable()
* Filter the data according to the search term, etc, always say carsQuery = carsQuery….
* Sorting - create an enum model and add it to the ViewModel / QueryModel. In the view itself add a dropdown with option enum values and enum names. In the controller pass the enum as param of the action and then sort the carsQuery with a switch case
* Paging – in the QueryModel /View Model add property const int CarsPerPage and another property int CurrentPage and another one for int TotalCars. In the query.Cars we add =

carsQuery.

Skip((query.CurrentPage -1) \* AllCarsQueryModel.CarsPerPage)

.Take(AllCarsQueryModel.CarsPerPage)…

In the viewModel we add buttons for prev and next page. TotalCars property is needed in calculations for next page. Also in order for the paging and the sorting to work together we need to send the other params in the asp-routing together with asp-route-currentPage

**Models:**

**InputModels** and **ViewModels** in Models folder creating sub-folder for each entity’s models:

* Models => Car => CarInputModel with business requirements attributes that the **ModelState** will be using to validate the input
* ([MaxLength] attribute vs [StringLength] explained in the template).
* Adding an error manually to the ModelState (ex. If selected car category does not exist): this.ModelState.AddModelError(nameof(car.CategoryId), “Category does not exist!”
* Custom Model Validation by inheriting : ValidationAttribute (ASP.Net Core Jun21, Working with Data @1:52, softuni.bg)
* When we want to visualize a drop-down menu with options we create a viewModel with the name and Id of these options and add the model as IEnumerable<viewModel> to the inputModel (ex. CarCategoryViewModel to CarInputModel as

public IEnumerable<CarCategoryViewModel> Categories {get; set;}

**Views:**

* \_Layout.cshtml – add link for adding car instead of the Home link
* Favicon - <https://favicon.io/emoji-favicons/> download and copy in the wwwroot folder
* Welcome page – bootstrap jumbotron

Each entity’s view is created in its own folder(named as the controller) and should be named as that action in the controller

* Views => Cars => Add -> View -> Razor View Empty => Add.cshtml
* Adding bootstrap – create an empty view, check the version of bootstrap in wwwroot => lib => bootstrap => dist => css => bootstrap.css, google bootstrap 4 form, get the first link <https://getbootstrap.com/docs/4.0/components/forms/>, copy the part for the form and paste it in the Add.cshtml
* Start adjusting it by
* first mentioning the @model CarInputModel (as per the model in the controller action)
* and add @using WebApplicationTemplate.Models.Cars in \_ViewImports.cshtml (CTRL + S to save it) so that the view files can have access to the model
* give a title of the page = @{ViewData[“Title”] = “Add car”;}
* after that logic can be added, ex. Display first 3 cars from the model list

@{

var firstCar = Model.Any()? Model[0] : null;

var secondCar = Model.Count > 1? Model[1] : null;

var thirdCar = Model.Count > 2? Model[2] : null;

}

* Add.cshtml -> <form method=”post”>
* Adjust the form fields as per the model properties – asp-for=”Brand” etc.
* Name of input fields the way they appear in the actual form can be adjusted from 2 places:
* In the .cshtml file <label **asp-for**="Description">Car description</label>
* Or in the input model CarInputModel.cs by using attribute [Display(Name = “Car Description”)]
* HTML adjustment:
* <textarea asp-for=”Description” rows=”4”….>
* <div class=”col-sm-12(on small screen let field take up 12 cols)

offset-lg-2(on large screen skip 2 cols and then start)

col-lg-8(on large screen let my field take up 8 cols)

offset-xl-3(on x-large screen skip 3 cols)

col-xl-6(on x-large screen take up 6 cols)”

* Set up margins in site.css by creating class

.heading-margin{margin-top: 20px; margin-bottom: 30px;}

Or by using class=”mt-5/mb-6”(margin-top/margin-bottom)

* Create submit button <input class="btn btn-primary" type="submit" value="Save" />
* Create dropdown menu:

<div class="form-group">

<label **asp-for**="CategoryId">Category</label>

<select **asp-for**="CategoryId" class="form-control">

@foreach (var category in Model.Categories)

{

<option **value**="@category.Id">@category.Name</option>

}

</select>

</div>

* Multiple Select – the property in the model should be IEnumerable<> and the tag in the view should be <select **asp-for**="CategoryId" class="form-control" multiple>
* Add Validation: <span **asp-validation-for**="Brand" class="small text-danger"></span>

Or do a summary on top: <div **asp-validation-summary**="All" class="small text-danger"></div>

* Create link <a asp-controller=”Cars” asp-action=”Details” asp-route-id=”@car.Id”>
* Uploading files (ASP.Net Core Jun21, Working with Data @2:00, softuni.bg)

<form method="post" enctype="multipart/form-data" **asp-controller**="Files" **asp-action**="Upload">

<input type="file" name="file" (or name=”files” multiple)/>

And in the action in the controller we must add IFormFile or IEnumerable>IFormFile as an attribute of the action

* “label asp-for” when the Label on top of the field is clicked, the cursor moves directly inside the field
* Adjusting greeting on \_LoginPartial @{if (this.SignInManager.IsSignedIn(this.User))

{

string name = UserManager.GetUserAsync(User).Result.FirstName;

<li> …etc

* Save the current work in VS with CTRL+S. Refresh the page with CTRL+F5 and in F12-Network-Disable Cache V

**Services**

* Every service should be registered in Startup.cs > ConfigureServices > services.AddTransient<IExampleService, ExampleService>()

**Identity User**

* Keep the SOLID principles and try to keep the IdentityUser class as it is and build the business logic around it by adding other classes that have one-on-one relationship to this IdentityUser class
* Getting the user ID
* can be done by injecting the UserManager in the controller

UserManager<ApplicationUser> userManager

var user = await this.userManager.GetUserAsync(this.User);

var userId = this.userManager.GetUserId(this.User);

* or with claims by using System.Security.Claims

var userId = this.User.FindFirst(ClaimTypes.NameIdentifier).Value;

or = this.User.Claims.First(c => c.Type == ClaimTypes.NameIdentifier).Value;

which can be shortened by creating an extension class of the ClaimsPrinciple in the infrastructure folder

public static string GetId(this ClaimsPrincipal user)

=> user.FindFirst(ClaimTypes.NameIdentifier).Value;

, that allows us to use this.User.GetId() in the actions in the controller;

* ..
* Authorise multiple roles:

[Authorize(Roles = GlobalConstants.PatientRoleName + "," + GlobalConstants.DoctorRoleName)]

* [Authorize] over controller and [AllowAnonymous] over certain actions or

simply [Authorize] directly over actions, where needed

* Roles: var userIsAdmin = this.User.IsInRole(“Administrator”);
* Scaffolding ASP.NET Core Identity:
* Right-click on the project >>Add >> Add New Scaffolded Item >> Identity
* Add Identity window >> select which pages to scaffold (login, register, logout…)
* Select Layout page from Views > Shared > \_Shared.cshtml and ApplicationDbContext
* Find the scaffolded item can be edited from Areas >> Identity >> Pages >> Accounts >> Manage

**Web API**

Web API (Application Programming Interface) can be seen as an url that does not visualize a web page, but rather returns data in the form of json/xml

How to create Web API in ASP.NET

* In Controllers folder create API folder
* Create controller CarsApiController and inherit ControllerBase
* Specify it is an apiController by using the attribute [ApiController]
* this gives us automatic Http 400 responses (for model state errors, so no need to use if(ModelsState.IsValid), it is checked automatically)
* easier binding, i.e. no need to specify the location of the parameter’s value such as [FromBody], [FromQuery], [FromForm], [FromRoute] etc. If it is a complex type parameter, Asp.NET will take it from the body/from json (i.e. Car car), if it is a primitive type it will look in the query (i.e. string name)
* Specify the route by attribute [Route(“api/[controller]”)] or “api/cars” – it is a must
* If additional routing is specified for each action it will be added to the routing of the controller, i.e. [Route(“details”)] => api/cars/details
* Also separate routing could be placed for each action rather than one for the whole controller
* Naming of models
* StatistincsResponseModel, instead of StatisticsViewModel,, in Folder Models > create Api folder > Statistics > create models inside
* AllCarsApiRequestModel instead of ViewModel, Model > Api > Cars > model
* CORS – Cross-origin Resource Sharing – by default the browser security prevents a web page from making requests to a diff domain, the so called SOP (Same Origin Policy). In order to enable the CORS we need to specifically allow the server to send certain access-control headers to the client in the request. This is done globally by middleware:
* In Startup.cs > public void ConfigureServices (…) > services.AddCors();
* Startup.cs > public void Configure(…) > app.UseCors(builder => builder.WithOrigins(“http://example.com”)) or builder.AllowAnyOrigin()

Or by Attribute on the action or on the controller:

* [HttpGet][EnableCors(AllowSpecificOrigin)]
* [HttpGet][DisableCors]

**REST**

* Web api, where CRUD operations are always towards the same url