**Which ASP.NET Core**

**How To Choose & How To Use**

**ASP.NET Core Razor Pages Project**

* Program.cs is where the app starts, just like a console app.
  + If you look in the project properties you can see the output type as console app.
* Startup.cs is where a lot of the configuration happens.
* Appsettings.json where you store all of your settings, like the connection string.
* Appsettings.Development.json can use different values and override appsettings.json, good to use in your development environment.
* Secrets.json “User Secrets” not in the project, its hidden on your disk.
* Wwwroot folder holds all static resources, like js or css files.
* All of the ASP.NET Core projects come with bootstrap 4 default, not the full bootstrap.
* Razor Pages is a server-side framework, the pages get rendered on the server and then sent to the client as just html and css.
* You mix c# and html, but the user never gets access to the c#. Only the final rendered version.
* Razor page has code behind “page model” associated with it.
* MVVM structure – has a two communication/ binding. MVC is a one way binding.
* One page model for one page

**ASP.NET Core MVC Project**

* Controllers, tells the view what info its going to get and what view to launch. Essentially a class with a bunch of methods. The methods are called actions. Information sending to the view (one way binding) nothing can be pulled from the controller. Any data you have to get on a page you have to send through the return View().
  + MVC is run by controllers, the controller is the heart of the MVC app.
* Models
* Views
  + When you want to create a new view, you create an action in a controller, then right click and select Add View. And it will scaffolded out a view based on your naming convention.
* MVC is a one way binding, you send info but it cannot be pulled – you can push info but you cannot be pull info. The controller can send info to the view but a view cant pull info from the controller.
* One controller, multiple views. Each view is supported by one or more methods.

**ASP.NET Core API Project**

* No wwwroot, no Models folder, no Views folder, but we do have a controllers folder.
* No View, because there are no views at all.
* No Models, because they don’t include one, but you can add them yourself.
* An API is not designed to talk directly to humans, its designed to talk to other machines.
* An API is most useful when its trying to have multiple user interfaces.
* An API can talk to another API.
* They don’t have a GUI, but they still have an interface.
* There isn’t a page, so you have your controller and it just sends back data. Same concept as controllers in MVC, the big difference is the return type. You’re sending back data instead of a view.
* By default its API/ something – for routing

**Blazor Server Project**

* Data folder – just demo stuff
* Pages folder – sample pages
* Shared folder – MainLayout.razor, how the page gets designed inside of the \_Host.cshtml.
* Inside \_Host.cshtml render an app, the app is App.razor.
* App.razor says if you found the route go there and use default layout of MainLayout.razor
* MainLayout.razor injects body of page.
* “.razor” is use in the Blazor projects, because its built with razor components
* Blazor pages are just razor components, that are marked as a page.
* Blazor use SignalR in the background.
* SignalR is a communication system the uses the most efficient communication it can for your web app, it depends on your browser, machine, server, and server machine. And find the most efficient communication method possible.
* Blazor both client and server has a concept as code section which is the supporting code for that particular page. Like code behind, but not behind. You can take that code and create a separate class file and put the code and join it to the page, making it unit testable.
* You can use dependency injection just like all the other ASP.NET apps
* MVVM structure – binding from code and page
* To add a new page right click on Pages folder and select Add, then select Razor Component.
  + The blazor page is a razor component.
  + You can then reuse that component
* Page is same a blazor client, difference in how you talk to data
* You can add a DLL and you can talk directly to it, you can talk to another project, you can use the same client library or the same data access library for razor pages, MVC, API, and blazor server

**Blazor Client Project**

* No Startup.cs for configuration, configuration moved into Program.cs.
* No Appsettings.json, because we don’t really need one.
* There is no direct access to a database.
* When you have more than one project in a solution you have to choose one is the startup project, unless you what to start more than one. To change right click project and select Set as Startup Project, or in the drop down menu.
* Page is same a blazor server, difference in how you talk to data
* You have to talk to your data via an API
* Blazor client is meant to run entirely on the client

**Project Similarities**

* They’re all essentially the same project type, just console apps.
* All founded off the Program.cs
* HostBuilder sets up a lot of our settings, initial configuration for appsettings.json, logging, enabling dependency injection
* All have the concept of wwwroot folder, except for API doesn’t share files it just disseminates information using the browser
* All use the concept of the razor syntax, except for API

**Project Differences**

* Minor tweaks on the same concepts, the approach or outcome.
* The pages highlight the biggest differences.

Mixing and matching is just a matter of saying yes I want this piece, and adding a little bit configuration in startup.cs that it needs, and go for it.

Just make sure you project type is compatible