**Which ASP.NET Core**

**How To Choose & How To Use**

**ASP.NET Core Razor Pages Project**

Simple web application with some pages

* 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
* Simplest, its one page with code that supports it, and the code that supports it is a two way binding which makes it easy to connect the frontend and the backend

**ASP.NET Core MVC Project**

Hybrid experience with a javascript framework

* 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.
* Use to the way to go, you had web forms then you had MVC.
  + A more structured way of doing web forms, more robust, testable, and is stateless
  + Use to be the best option for web development in c#
* MVC is now good, for being a hybrid. The ability to display things like razor pages (regular views), and also the benefits of being an API (controllers that return just data and not views)
  + You can view that has angular inside of it, or one of the single page applications frameworks inside of one page, and those javascript frameworks can be supported by an MVC controller. The other actions in the controller can send data and get data from those angular applications. The js frameworks have to have an API to support them, and MVC can be that API.
  + MVC can support the front applications both with a view and with the data communication back and forth through the controller

**ASP.NET Core API Project**

For talking to other applications

* 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
* Designed to talk to other projects
* Talking to other systems allows for integrations, a ne interface, mobile app, ect.
* An API allows you to put all your business logic, all your data access behind the API. The API just has endpoints that you talk to. They don’t how it give you the info (what type of DB it is), or the process it went through to verify. Simplifies the giving of information. You could build more complex user interfaces quicker.
* You can have a mobile app, web app, and desktop app all use the same logic and the same data access, because theyre all pointing to the API.
* Its goal is to have other projects talk to it.

**Blazor Server Project**

Rich client experience

* 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
* It gives you all the those feature that you like from angular or react with the interactivity on the client, yet your still running in c sharp, and you’re still on the server
  + The difference is you have the SignalR connection going back and forth (very scalable), some work on the client and some work on the server
* It doesn’t give you offline access, or the ability to be a progressive web app

**Blazor Client Project**

For offline access on the clients side

* 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
* Designed to be an app that runs entirely on the clients machine, you send down the entire app, all the source code everything gets loaded onto the clients machine. (Like angular and react). The client has full access to read all the js, and can modify the local js (cached version).
  + Because its in the client it really cant have data access, anything that is proprietary any business logic, any data access logic, you wouldn’t want to put on a clients browser machine.
* Once the app is downloaded the work gets done on the users machine
* It designed to work offline

**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

A web app you think of a server that sends down data.