MVC Concepts

.NET CORE

*The **Model-View-Controller (MVC)** architectural pattern separates an application into three main groups of components: **Models**, **Views**, and **Controllers**. This pattern helps to achieve separation of concerns.*

HTTPS://DOCS.MICROSOFT.COM/EN-US/ASPNET/CORE/MVC/OVERVIEW?VIEW=ASPNETCORE-3.1

MVC - Separation of Concerns

https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#separation-of-concerns

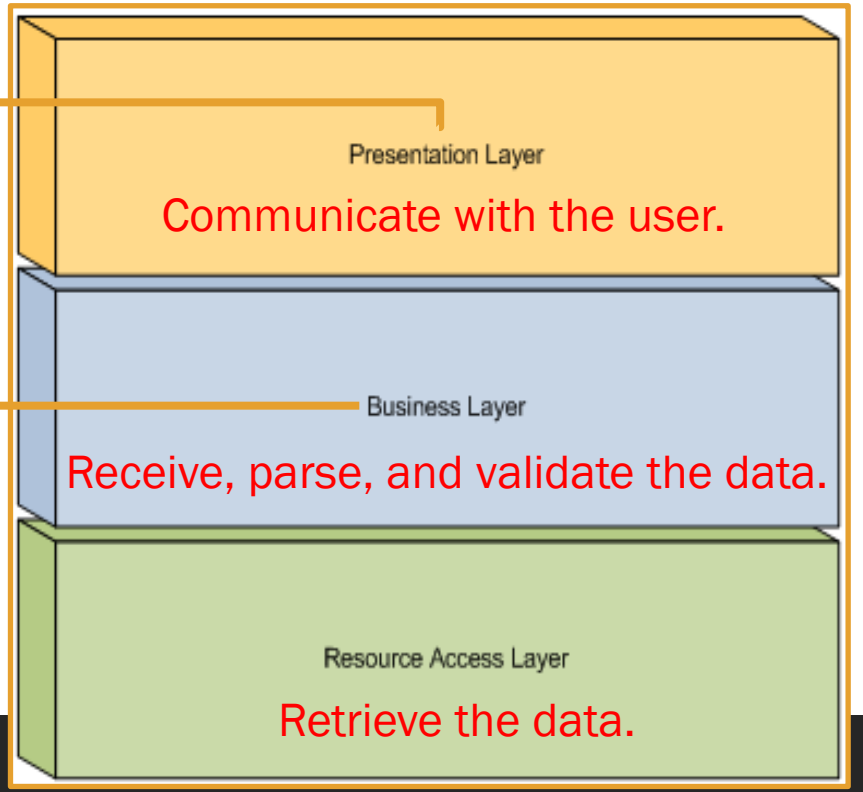
Consider an application that includes ***logic*** for identifying noteworthy items to display to the user and formats those items in a way to make them more noticeable.

There are two separate behaviors responsible for:

1) choosing which items to format
2) formatting the items.

Applications should be built to separate ***core business behavior*** from ***infrastructure*** from ***user interface*** logic. ***Separation of concerns*** is a key consideration behind the use of ***layers*** in application architectures.

This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something that has a single job.



MVC – Control/Data Flow

https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1#what-is-the-mvc-pattern

The ***Model-View-Controller (MVC)*** architectural pattern separates an application into ***Models***, ***Views***, and ***Controllers***.

1. User requests are routed to a ***Controller*** which
2. works with the ***Model*** to perform user actions and/or retrieve results of queries.
3. The ***Controller*** then chooses the appropriate ***View*** and
4. provides it with the ***Model*** data it requires to display results to the user.

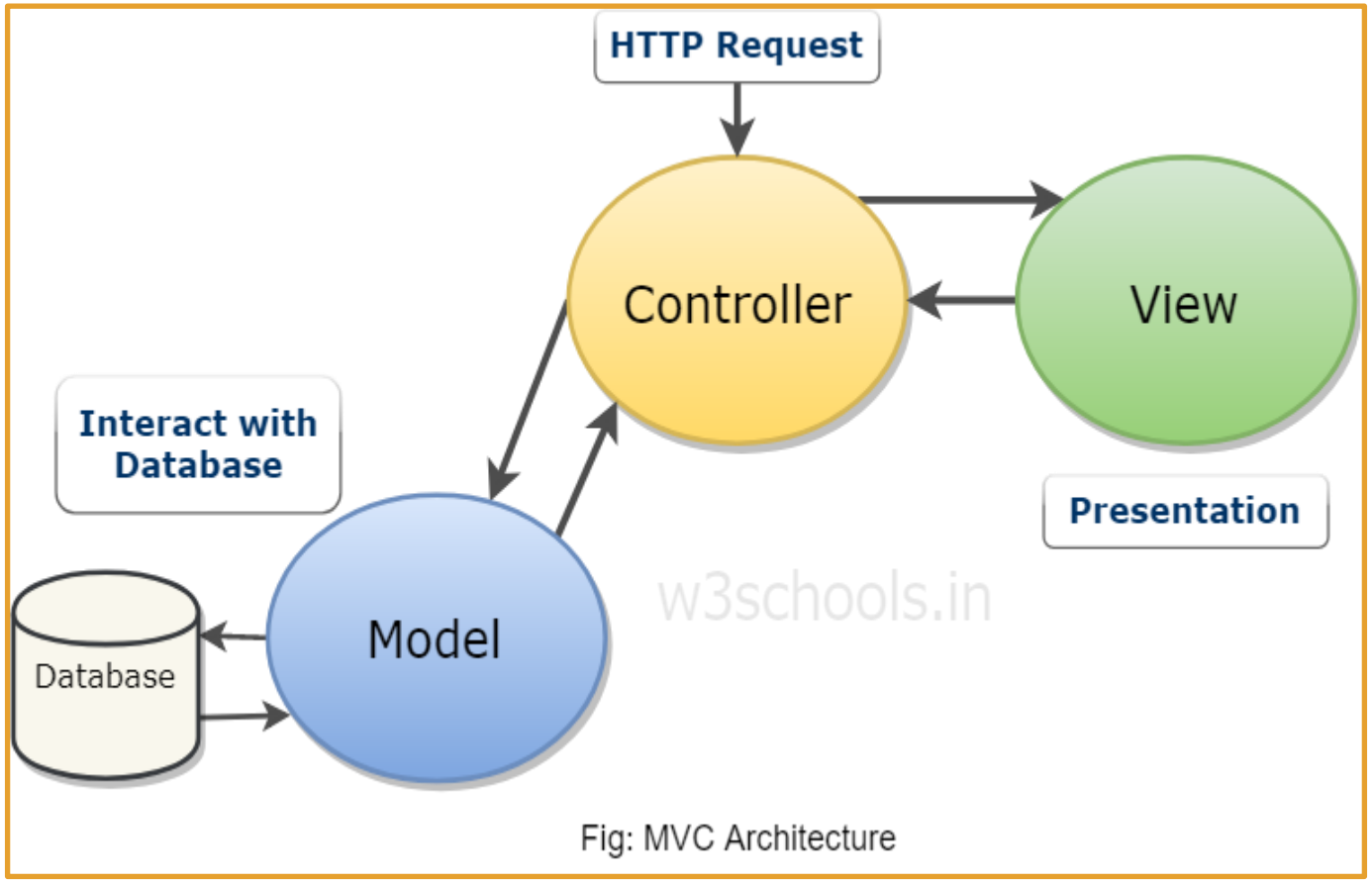


Fig: MVC Architecture

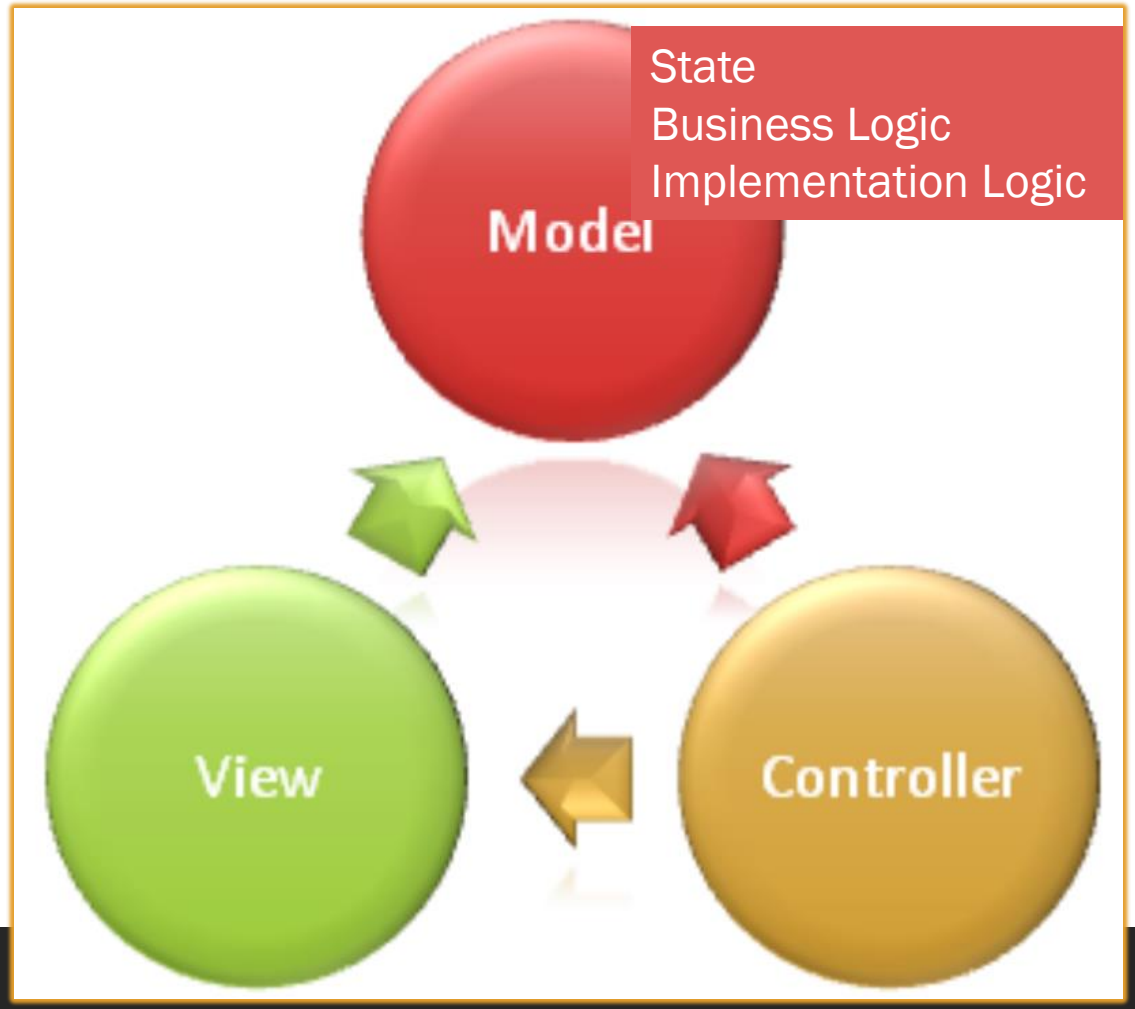
MVC – Model

https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1#model-responsibilities

The ***Model*** part of an ***MVC*** application represents the state of the application and any ***business logic*** or operations that should be performed by it.

Business logic is encapsulated in the ***Model***, along with any ***implementation logic*** (DbContext) for persisting the state of the application (the DataBase).

Strongly-typed views typically use ***ViewModel*** types designed to contain the data to display on that ***view***. The ***Controller*** creates and populates these ***ViewModel*** instances from the ***Model layer***.



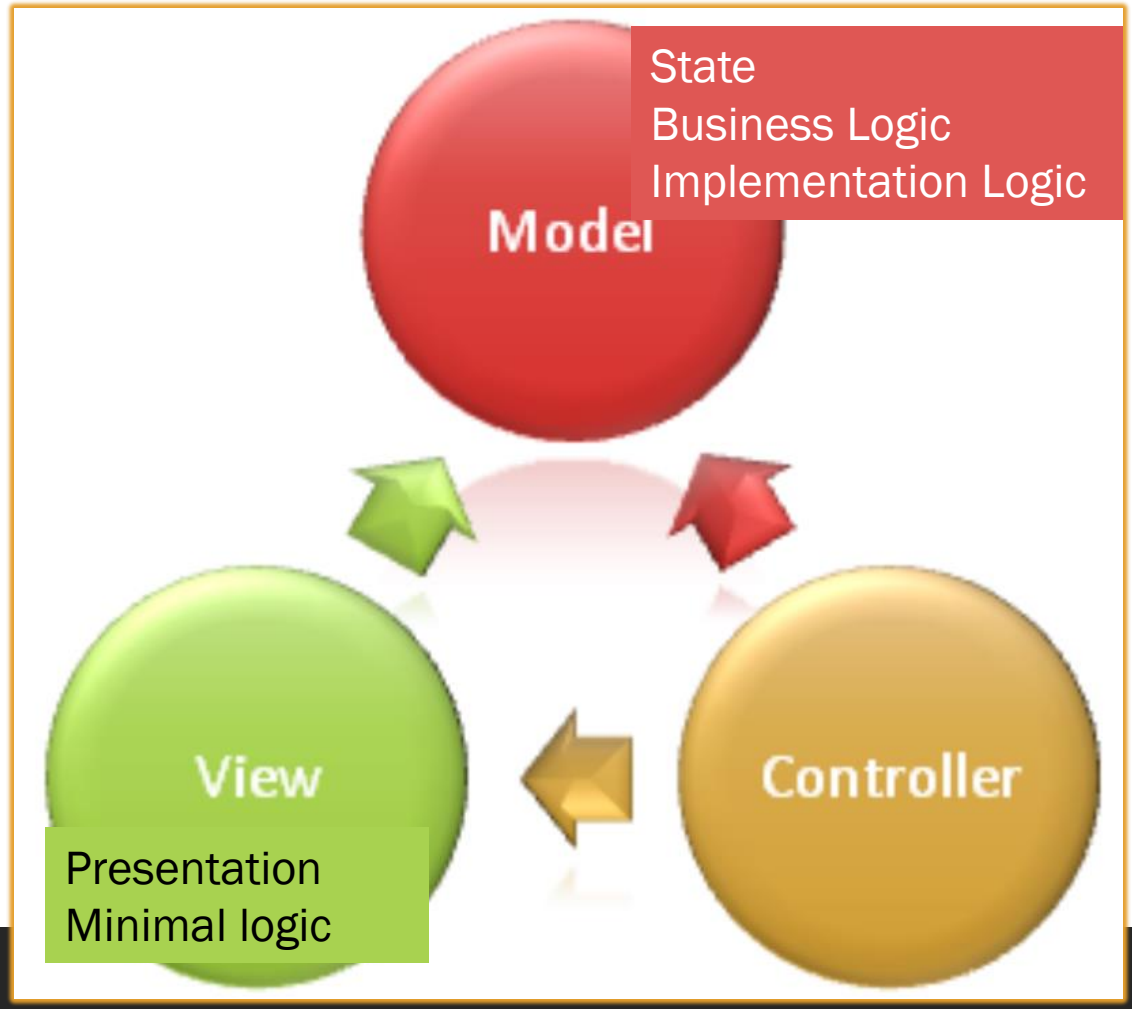
MVC – View

https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1#model-responsibilities

Views are responsible for presenting content through the user interface.

Views use the ***Razor view engine*** to embed .NET code in HTML markup.

Logic in ***Views*** should relate to presenting content only. If logic is necessary in order to display data from a complex model, use a ***View Component*** or ***ViewModel*** simplify the view.



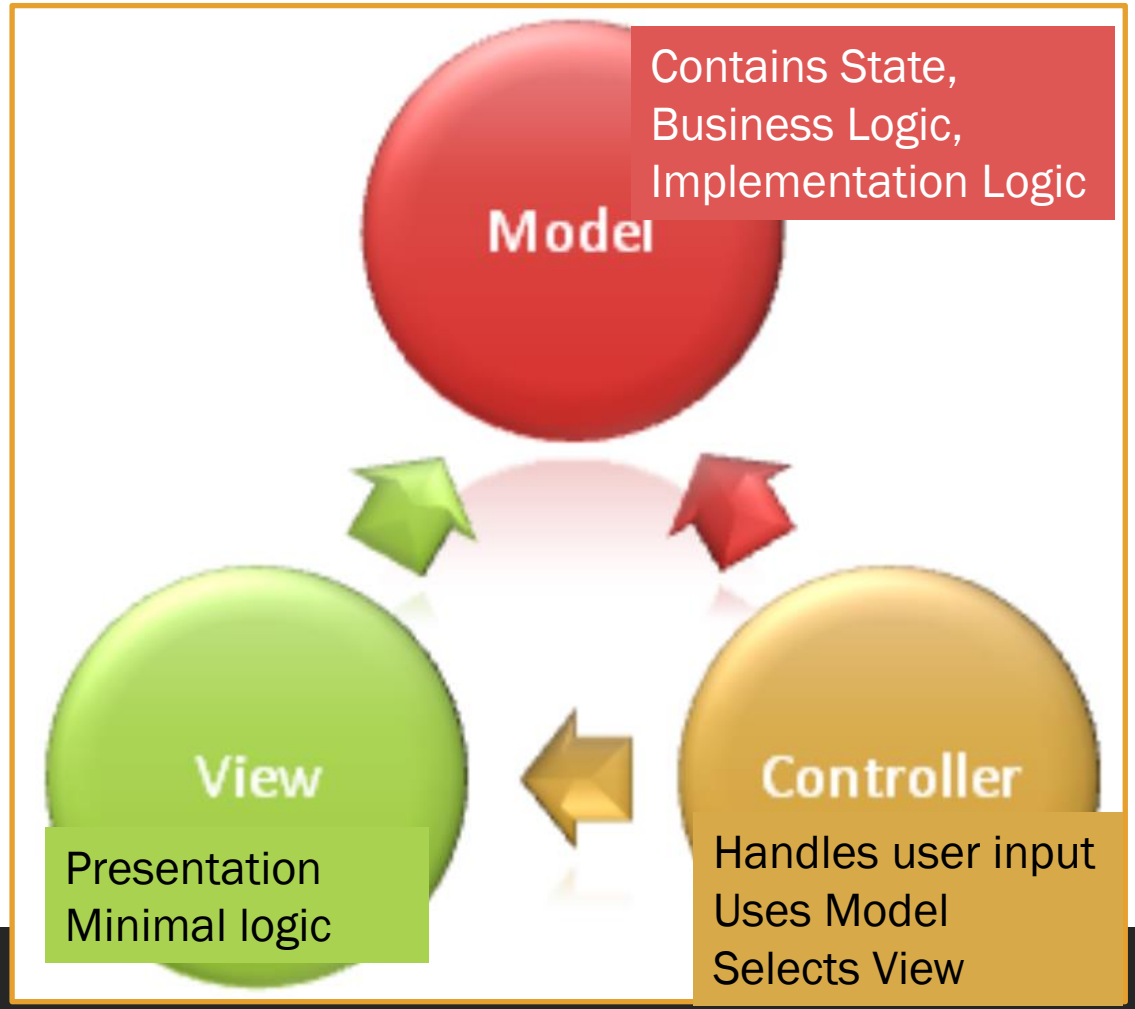
MVC – Controller

https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1#model-responsibilities

Controllers are the components that

- handle user input,
- work with the ***model***, and
- select a ***view*** to render.

The ***Controller*** handles and responds to user input. In the MVC pattern, the ***Controller*** is the initial entry point, and is responsible for selecting which ***Model*** types to work with and which ***View*** to render (hence its name - it controls how the app responds to a given request).



Why Use ASP.NET Core MVC?

Use of the MVC approach helps you create applications that separate the different aspects of your application (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the application.

This architecture is good but what about testing? What about dynamic web pages? What about model validation and Web API's? Wouldn't it be nice to have a framework with all those technologies built in?

This is where ASP.NET Core MVC comes into the picture.

ASP.NET Core MVC - Overview

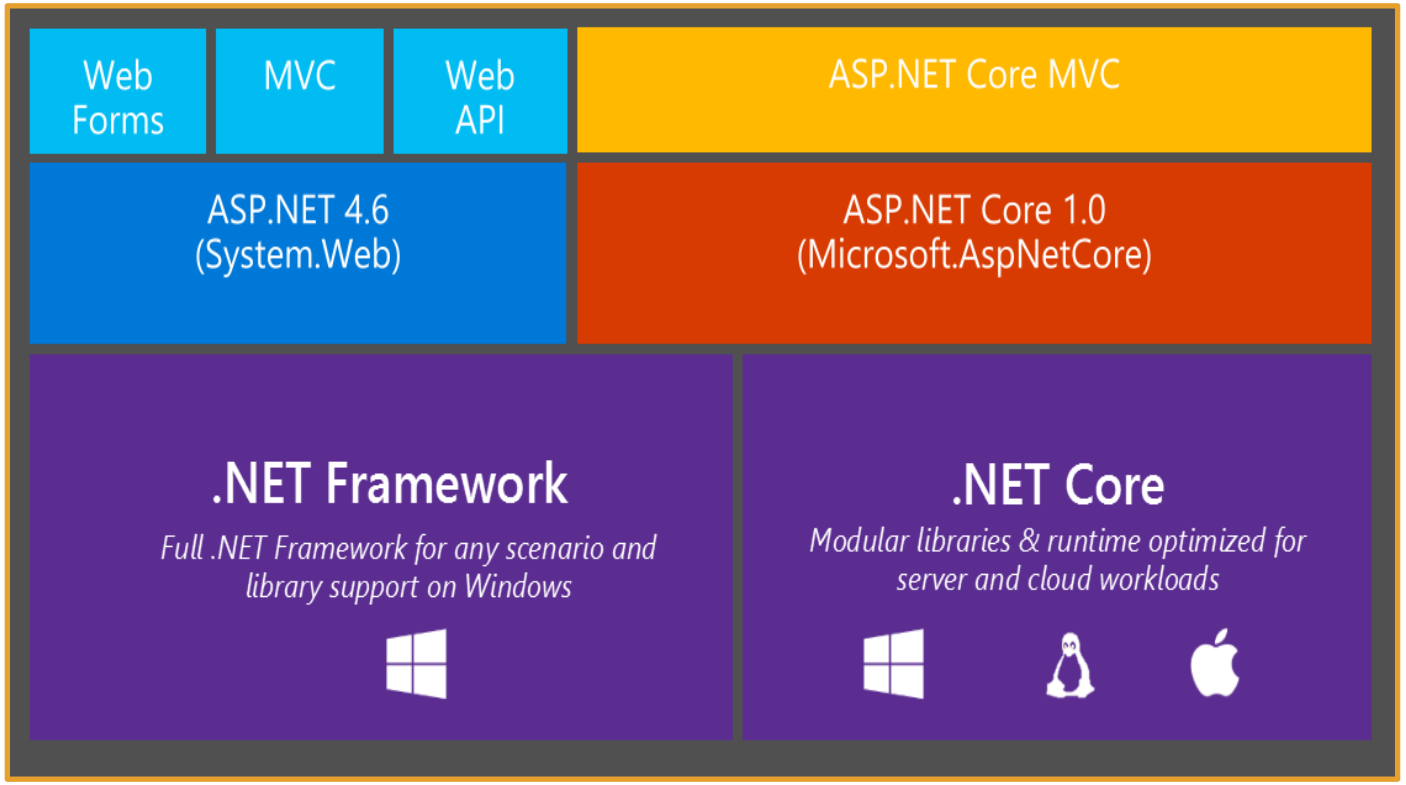
https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1&source=docs#what-is-aspnet-core-mvc

The ***ASP.NET Core MVC*** framework is a lightweight, open source, highly testable presentation framework optimized for use with ***ASP.NET Core*** but using the MVC architectural pattern.

ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns.

ASP.NET Core MVC:

- full control over markup,
- supports TDD-friendly development
- uses the latest web standards.



ASP.NET Core MVC Tutorial

Complete the **ASP.NET Core MVC** tutorial here

Nicks p1 notes on whiteboard.

user clicks on "details for restaurant #1"

routing + model binding
-> RestaurantController.Details(1)

controller is instantiated with an instance of the repository
the repository is instantiated with an instance of the dbcontext

} configured in Startup.ConfigureServices (+ the ctors)

in Details action method... the repo is called to get the given restaurant by ID.

(the repository exposes methods which talk only in terms of the business logic models. it treats the EF entity classes as an implementation detail hidden from the controllers or anyone else using the repository.)

the repo tells the dbcontext to get the restaurant (entity) by ID.

EF sends SQL to the Azure SQL DB with joins, etc.

the repo makes a restaurant domain model based on the entity it got back.

this code to map between those two forms I moved to a separate Mapper class