# Introduction to MVC

* Stands for Model-View-Controller
* It is a design pattern we used to separate our application into three main groups
* This will help us achieve separation of concern
  + Makes it easier to scale our application
  + Easier to code, debug, and test
  + Make your application easier to digest

## Model

* The model process data / user input
* It represents the current state of application

## View

* It is what the end-user sees and interacts with
* Essentially what the webpage you see when visiting a website
* So it is composed of your HTML, CSS, JS, or other assets

## Controller

* The controller handles the client request
* It will call upon appropriate business logic to process what the client wants from the database/server
* The controller will then choose the appropriate view that the user should interact with
  + It will also fill in any model data needed to display that view

# ASP.NET MVC

* It is the web application framework developed by Microsoft that implements that MVC design pattern
* Since it uses the MVC design pattern, your web application is separated into a model, view and a controller

# Communication between the server and the client

* Basically, how writing a URL address in your browser and magically brings you a nice-looking webpage.
* One thing to note: we will mainly focus on HTTP/HTTPS
  + Hyper Text Transfer Protocol Secure

## Request response life cycle

1. Client (your browser) will send a request (the url you sent)
2. The server will receive that request and will do some process
3. The server will send an appropriate response (html, css, js, etc.) and a status code
4. The client will receive the response and the browser will process that response

# DNS

* It stands for Domain Name System
* It is essentially a directory of names and ip address
* It translates our pretty name of a website (google.com) into a numerical ip address (0.0.0.0 – 255.255.255.255) for locating the right server that is connected to that website
* Main reason is so people can easily find your website since it is easier to memorize a name verses a combination of numbers.

# HTTP Request

* It has 3 requests
  + Request Line
    - Method – describes what action will be perform
    - Target – describes where to send that request
  + Header
    - Meta data
    - Ex: content type – what data type the body has
  + Body
    - Data you want the server to process

# HTTP Response

* It has 3 parts
  + First line
    - Gives a **status code**
  + Middle subsequent lines
    - Meta data
    - Like HTTP Request Header
  + Final block of lines
    - The data block (it is the data that the http response gives to you)

# HTTP Verb/Methods

* Describes what action the client wants the server to perform on a given resources
* Common Verbs
  + GET – Used to retrieve data from the server
  + POST – Used to submit data to the server
  + PUT – Replaces/updated the existing data on the server
  + DELETE – Deletes existing data on the server
  + HEAD – it is like a get method, but I will only give you the header

## Safe methods

* A method is safe if it doesn’t change/alter your data in your server
* Ex: GET or HEAD

## Idempotent methods

* If you call on the same method repeatedly, it should give the same result/effect
* So, image you created a DELETE http method that will delete the last entry in your server
  + Try calling the method 4 times and it will delete different row each time
  + This will make the DELETE http method not idempotent only because it gives different result each time
  + To fix this, make sure your delete http method uses the customer id to find its row and try deleting the same row every time.

# HTTP status code

* Gives the result of an HTTP request
* They are group of 5 different levels:
  + 1XX – Informational
  + 2XX – Successful
  + 3XX – Redirection
  + 4XX – Client error – Your frontend needs work
  + 5XX – Server error – Your backend needs work

# Controller

## Controller Actions

* An action is essentially a method inside a controller that will be called depending on what url http request the user is asking to be displayed on their web browser.

## Request Parameter

* Controller actions has the capability of adding parameters into the methods to essentially pass data from the url to the controller action.
* Common ways of application are getting data from your database and displaying to the view

# Model

## Model Binding

* It is a way to bind data coming from your HTTP request direct into a model
* It is essentially the fancy way of converting the JSON file into objects that C# understands

## Data Annotation

* Great way to include validation for your models
* Essentially instead of adding implementation details in your properties to enforce certain data to only be stored, data annotation can be used to enforce it.

## ViewModel

* This is how we pass model information to our views
* This is done by using our controllers and having the controller’s action pass some model to the view

# View

* It is basically html with c# coding mixed into it
* Allows us to have dynamically changing views

## Partial-View

* It is an incomplete view that is mostly used with other views
* Useful if you want a view that needs to be displayed in numerous other views

## Strongly-View

* It is when you pass in the model itself in the controller’s action
* It enforces strongly typed checking in the view itself

## Weakly-View

* It is weakly-typed checking since datatype are resolved at runtime
* Try to avoid since intellisense won’t help you see certain datatypes cannot be stored in certain properties of a model
  + i.e. storing a string into an int property
* More error-prone

# Introduction to ORM

* Stands Object-Relational Mapper
* They are used to essentially translate data from our C# objects into table that SQL understands
  + So in essence, ORM translates table from database to objects or vice versa
* The ORM is Entity Framework

# Entity Framework Core

* One of the popular ORM for .Net
* It allows us to work with a database by using C# objects and almost completely remove the need for most of the data-access code you usually must write.

## Two approaches to EF

* Database-first approach
  + Create a database first and EF will scaffold the rest into C#
* Code-first approach
  + We create the models, and we use EF to scaffold the database architecture in SQL

## CLI Steps to do code-first approach

1. Move DBcontext to RRDL folder
2. Remove entities folder
   1. Make note of the properties in those entities and ensure your model properties are similar
3. Fix the compiler exceptions you’ll get for removing the entities folder
   1. Most of them are just removing the mapping from model to entity or vice versa we did
   2. Tip: start on DBContext first then your repository cloud
4. Add EF design package to RRWebUI and EF SQL Server Package
   1. This is our new startup project
5. Add project references in RRWebUI
   1. You need to reference your BL, DL, and Model
6. Configure appsettings.json to have connection strings that point to your database
7. Configure services to tell MVC that it will depend on the DBContext from the DL
8. Cd to DL and run this scaffold cli command
   1. dotnet ef migrations add [name of migration] -c [name of you dbcontext class] --startup-project [location of your MVC project]
   2. Remove all the tables from your database
   3. dotnet-ef database update --startup-project [location of your MVC]

## Some artifacts you’ll be working with

* DBContext
  + A class in EF that represent a session with the database and can be used to get, manipulate, add, etc. entities to your database
* Connection String
  + A string that will be used to connect our database
  + Basically, a long string that has information about our database
* Migration
  + They are a snapshot of your database architecture/schema depending on your models
  + Acts as a version control of your database
* Entities
  + They are the C# classes equivalent of your database tables
  + So a student table in your database will have a student class entity in your C# application

## Two ways to load data from the database

* Eager Loading
  + It will process the query of one entity and will load any related entities as part of the query
  + So, if Restaurant has many reviews, it will also load all the review data that is related to that restaurant as part of the query
  + You must use the **Include ()** method to make it eager loading
* Lazy Loading
  + Opposite of eager loading and will delay loading the related data until you access it
  + By default, EF core will lazy load your queries