ASP.NET MVC

public class FirstController : Controller { public string Index() { return "Hello World"; } public IActionResult Hello() { return View(); }

Adding Dynamic Data to the View: public IActionResult Hello() { ViewBag.Message = "Hello World"

<h2>@ViewBag.Message</h2>

Adding Model and transferring its data to the View: Person person = new Person(); person.Name = "John"; return View(person);}

@model Person

Name: @Model.Name Name: @Model.Age Name: @Model.Location

Passing data with ViewData:

public ActionResult Index() { ViewData[“EmployeeName”] = “John Nguyen”; ViewData[“Company”] = “Web Development Company”;

Employee Name: @ViewData[“EmployeeName”]
 Company Name: @ViewData[“Company”]

Passing data with tempdata:

TempData[“Employee”] = employee; return RedirectToAction(“PermanentEmployee”); } //Controller Action 2(PermanentEmployee) public ActionResult PermanentEmployee() { Employee employee = TempData[“Employee”] as Employee; return View(employee); }

ASP.NET Core Action Methods are public methods defined inside the Controllers. These methods are mapped to incoming requests made from the client through routing rules

Session:public IActionResult SessionExample() { HttpContext.Session.SetString("CurrentDateTime", DateTime.Now.ToString());

<a asp-controller=”Home” asp-action=”Index”>Link: Generates a link with a URL that’s generated by routing based on the specified controller and action. : Generates a <img> tag with a cache-busting query string to ensure the latest version of the image is fetched. <form asp-controller=”Account” asp-action=”Login” method=”post”>...</form>: Generates a form element with the appropriate action attribute for the specified controller and action. <input asp-for=”Username” />: Generates an input field with attributes based on the model property specified by asp-for.

A **Domain Model** represents the object that represents the data in the database. The Domain Model usually has one to one relationship with the tables in the database. The Domain Model is related to the data access layer of our application. They are retrieved from the database or persisted to the database by the data access layer

The **View Mode**l refers to the objects which hold the data that needs to be shown to the user. The View Model is related to the presentation layer of our application.

The **Edit Model** or Input Model represents the data that needs to be presented to the user for modification/inserting

The **ActionName** attributes define the name of the action. The routing engine will use this name instead of the method name to match the action name routing segment. You will use this attribute when you want to alias the name of the Action method.

[ActionName("Modify")]

public string Edit()

Remember that you can also use the route attribute to change the Action Name.

public class HomeController : Controller {

[Route("Home/Modify")]

public string Edit()

Web API

Problems without Web APIs: Duplicate logic for each Application, Error-Prone Code, Some Front-end frameworks cannot communicate directly with the Database,Hard to Maintain **Advantages of Web API**: Using Web API, we can avoid code duplication, Extend Application Functionality, Abstraction, Security **Client-Server decoupling:** client and server programs must be independent **Uniform Interface**: There should be a uniform and standard way of interacting with a given server for all client types **Statelessness:** each request must contain all the information needed to process it. **Layered System architecture**: REST APIs must be designed so neither the client nor the server can tell whether they communicate with the final application or an intermediary **Methods of REST API:** GET: The HTTP GET method is used to read (or retrieve) a representation of a resource. In the safe path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST), POST: The POST verb is most often utilized to create new resources. In particular, it’s used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status, PUT: it updates a resource by replacing its content entirely. As a RESTful API HTTP method, PUT is the most common way to update resource information. However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. PUT is not safe operation but it’s idempotent. PATCH: It is used to modify capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource. This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch. PATCH is neither safe nor idempotent. DELETE: It is used to delete a resource identified by a URI. On successful deletion, return HTTP status 200 (OK) along with a response body. **Idempotence:** HTTP method that can be called many times without different outcomes **Register the database context:** services such as the DB context must be registered with the dependency injection (DI) container: builder.Services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList")); **Add a database context:** The database context is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class, namespace TodoApi.Models; public class TodoContext : DbContext **[Api Controller]:**This attribute indicates that the controller responds to web API requests **Swagger:** Swagger is used to generate useful documentation and help pages for web APIs. app.UseSwagger();app.UseSwaggerUI(); **GetTodoItem:** [HttpGet("{id}")] public async Task<ActionResult<TodoItem>> GetTodoItem(long id) { var todoItem = await _context.TodoItems.FindAsync(id); if (todoItem == null) { return NotFound(); } return todoItem; } **PutTodoItem:** [HttpPut("{id}")] public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem) { if (id != todoItem.Id) { return BadRequest(); } _context.Entry(todoItem).State =

You can let Routing Engine know that a particular method is **not an Action** method by decorating it with NonAction attribute as shown below public class HomeController : Controller { [NonAction] public string Edit()
The **Action verbs selector** is used when you want to control the Action method based on HTTP request method. This is done using the set of attributes provided by MVC, such as the HttpGet and HttpPost attributes, which collectively called as HTTP Attributes
In the Tutorial on Attribute routing, we showed you how to configure the route in the Route Attribute. Instead, you can use the HTTP Action verbs to do the same.[HttpGet("")] [HttpGet("Home")] [HttpGet("Home/Index")]
public string Index() { return "Hello from Index method of Home Controller"; }

objects defined by the conceptual model
DB Context: the database session and acting as a bridge between the application and the database
A LINQ query that gets invoice data through the Customer object: var invoicesList = from customer in mmaBooks.Customers from invoice in customer.Invoices orderby customer.Name, invoice.InvoiceDate select new { customer.Name, invoice.InvoiceID, invoice.InvoiceTotal };
Eager loading means that the related data is loaded from the database as part of the initial query. **Explicit loading** means that the related data is explicitly loaded from the database at a later time. **Lazy loading** means that the related data is transparently loaded from the database when the navigation property is accessed
When to use what Use Eager Loading when the relations are not too much. Thus, Eager Loading is a good practice to reduce further queries on the Server. Use Eager Loading when you are sure that you will be using related entities with the main entity everywhere. Use Lazy Loading when you are sure that you are not using related entities instantly.
A query expression that uses a navigation property to load related objects(Lazy loading):var customerInvoices = (from customer in mmaBooks.Customers where customer.CustomerID == customerID select new { customer.Name, customer.Invoices }).Single();
A query expression that uses the Include method to load related objects (Eager Loading) var selectedCustomer = (from customer in mmaBooks.Customers.Include("Invoices") where customer.CustomerID == customerID select customer).Single();
Code that explicitly loads the objects on the many side of a relationship:var selectedCustomer = (from customer in mmaBooks.Customers where customer.CustomerID == customerID select customer).Single(); if (! mmaBooks.Entry(selectedCustomer).Collection("Invoices").IsLoaded) mmaBooks.Entry(selectedCustomer).Collection("Invoices").Load();
Code that explicitly loads the object on the one side of a relationship:var selectedInvoice = (from invoice in mmaBooks.Invoices where invoice.InvoiceID == invoiceID select invoice).Single(); if (! mmaBooks.Entry(selectedInvoice).Reference("Customer").IsLoaded) mmaBooks.Entry(selectedInvoice).Reference("Customer").Load();
Code that retrieves a customer row:var selectedCustomer = (from customer in mmaBooks.Customers where customer.CustomerID == CustomerID select customer).Single(); **Code that modifies the data in the customer row selected:** Customer.Name = txtName.Text; selectedCustomer.City = txtCity.Text; **A statement that saves the changes to the database**:mmaBooks.SaveChanges();
A statement that marks the Invoice object for deletion:mmaBooks.Invoices.Remove(selectedInvoice);

EntityState.Modified; try { await _context.SaveChangesAsync(); } catch (DbUpdateConcurrencyException) { if (!TodoItemExists(id)) { return NotFound(); } else { throw; } } return NoContent(); }**DeleteTodoItem:** [HttpDelete("{id}")] public async Task<IActionResult> DeleteTodoItem(long id) { var todoItem = await _context.TodoItems.FindAsync(id); if (todoItem == null) { return NotFound(); } _context.TodoItems.Remove(todoItem); await _context.SaveChangesAsync(); return NoContent(); }
DTO: The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model

LINQ

public class Author { public int AuthorId { get; set; } public string Name { get; set; } public virtual ICollection<Book> Books { get; set; } }
Entity Framework is an ORM, Entity Framework can generate the database schema from your code, or map your existing database to your code
When you execute a query against a **conceptual model**, Object Services works with the EntityClient data provider and the Entity Data Model to translate the query into one that can be executed by the database. When the results are returned from the database, Object Services translates them back to the objects defined by the conceptual model
Migrations Used for creating and applying database schema changes.
LINQ (Language Integrated Query) Used for querying data in a type-safe manner
DbContext: Represents the database session and provides a way to query and interact with the database. **DbSet:**Represents a collection of entities in the database. **Entity Classes:**POCO classes that map to database tables. **Configuration:**Fluent API or Data Annotations to configure entity properties and relationships.
The data model consists of entity classes that correspond to your database tables, and a context class that inherits from DbContext and manages the connection and transactions
Three ways to query a conceptual model: LINQ to Entities Entity SQL, Query builder methods
To use Entity with LINQ, we need to add an ADO.NET Entity Data Model in our application.
//Query Syntax
IEnumerable<int> numQuery1 = from num in numbers where num % 2 == 0 orderby num select num;
//Method Syntax
IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n)
EF Core is an open-source object-relational mapping (ORM) framework developed by Microsoft. It is a lightweight and cross-platform version of Entity Framework (EF).
The Northwind.**Context.tt file** is a T4 code-generation file that is responsible for creating the class that contains the Context that we will be working with in code
The **Northwind.Context.cs** file is the class that was generated by the T4 template, and acts as the context class for this model.
Northwind.Designer.cs and **Northwind.edmx.diagram** are files used in creating and changing the diagram we saw earlier.
Northwind.tt is another T4 code generation template that is responsible for generating the classes for all objects modeled by the database.
See the 1 and * markers? Those signify the **relationship** between the two tables: for every 1 region, there can be many territories. This is called a 1-to-many relationship.
This relationship looks similar, but now we have 0..1 to *. This is called a **zero-or-one-to-many relationship**
Now there's an * on each side of the relationship. This is called a **many-to-many relationship.**
Entity Data Model that defines a conceptual model, a storage model, and mappings between the two models
The Entity Framework provides a layer between the database used by an application and the objects used by an application. This layer provides an interface that allows the data in the database to be presented to the application as objects
When you execute a query against a conceptual model, Object Services works with the EntityClient data provider and the Entity Data Model to translate the query into one that can be executed by the database. When the results are returned from the database, Object Services translates them back to the