# Short Answer:

Answer the following questions with complete sentences in your own words. You are encouraged to conduct your own research online or through other methods before answering the questions. If you research online, please consult multiple sources before you write down your answers. You are expected to be able to explain your answers in detail (Provide examples to each question).

1. What is MVC?

MVC

MVC is an architecture that separates business logic, presentation and data. In

MVC,

● M stands for Model

● V stands for View

● C stands for controller.

MVC is a systematic way to use the application where the flow starts from the view

layer, where the request is raised and processed in controller layer and sent to model

layer to insert data and get back the success or failure message.

● Model

○ The Model in an MVC application represents the state of the application and any

business logic or operations that should be performed by it

● View

○ Views are responsible for presenting content through the user interface.

● Controller

○ Controllers are the components that handle user interaction, work with the model,

and ultimately select a view to render

Separation of Concern

● Loose coupling between model, view, and controller

● Easy to scale

MVC stands for Model-View-Controller, it is an architectural pattern used to structure the code in a way that separates the concerns of the application.

The Model represents the data and business logic of the application. It contains the classes and methods that are used to interact with the data source (e.g. database), perform business logic, and provide data to the Controller.

The View represents the user interface of the application. It contains the code that is responsible for displaying the data to the user.

The Controller is responsible for handling user input and interacting with the Model to retrieve and update the application's data. It receives the input from the user, performs any necessary logic, and updates the view to reflect the changes.

The key idea behind MVC is to separate the concerns of the application, so that changes to one component do not affect the other components. This makes it easy to maintain, test, and extend the application.

In summary, The Model contains the data and business logic, the View handles the presentation of the data, and the Controller handles user input and updates the Model and the View accordingly. They communicate with each other by following the predefined interfaces and contracts.

2. Why should you use MVC?

The Model-View-Controller (MVC) pattern is a widely-used software architectural pattern that is used to separate an application's concerns into three distinct components: the Model, the View, and the Controller.

Here are some reasons why you should use MVC:

1. **Separation of concerns:** MVC provides a clear separation of concerns, which means that the different components of an application are responsible for different tasks. This makes the application easier to understand, develop, test, and maintain.
2. **Modularity:** MVC promotes the use of small, independent, and reusable components, that can be reused across multiple views or controllers. This allows you to change the implementation of a single component without affecting the rest of the application.
3. **Testability:** MVC makes it easier to write automated unit tests for your application. Because the different components of the application are loosely coupled, it's easy to test them in isolation.
4. **Scalability:** MVC architecture allows for more scalability, it allows a developers to add new functionality, or to replace existing functionality, with minimal impact on the existing codebase.
5. **Flexibility:** MVC allows for greater flexibility in the way that your application handles different types of requests and responses. It gives the developer the ability to handle different types of requests and responses with different actions.

Overall, MVC is a popular architectural pattern that is widely used in modern web development. It helps to keep your code organized, easy to understand and maintain, and it also promotes the use of best practices and principles such as separation of concerns, testability, scalability, and flexibility.

1. What is ASP.NET Core MVC request life cycle?

Diagram

Description automatically generated

The request life cycle in an ASP.NET Core MVC application refers to the series of steps that occur between the time that a client makes an HTTP request to your application, and the time that the response is sent back to the client. The exact order of steps may vary slightly depending on the version of ASP.NET Core and the specific configuration of your application, but generally, the request life cycle in ASP.NET Core MVC includes the following steps:

1. **Request Received:** The web server receives the incoming HTTP request and parses it to extract the method, URI, headers, and body of the request.
2. **Routing:** The request is passed to the routing system, which is responsible for determining the correct controller and action method to handle the request. Routing is based on the URI of the request and the routing rules that are defined in your application.
3. **Controller Selection:** The specified controller is instantiated, and the action method is invoked based on the routing result.
4. **Model binding and validation:** The framework automatically maps the request data to the action method's parameters, and it also performs validation on the input.
5. **Action execution:** The action method is executed, it will perform any necessary business logic, and it will interact with the model to retrieve or update data.
6. **View selection and rendering:** The framework selects the appropriate view to render the response and use it to generate the HTML markup that will be sent back to the client as the response.
7. **Response sent:** The response is sent back to the client along with the appropriate HTTP status code and headers.
8. **Complete:** Once the response has been sent, the request life cycle is complete and the connection is closed.

In addition to these basic steps, the request life cycle can be affected by middlewares, filters, or custom code that can be added to customize or extend the behavior of the process. Overall understanding request life cycle can be helpful in debugging and troubleshooting issues in your application, as well as for customizing the behavior of your application.

Top of Form

Bottom of Form

1. Explain Controller and Action?

Controller

● A controller is used to define and group a set of actions.

● An action (or action method) is a method on a controller which handles

requests.

● Controllers logically group similar actions together.

● Requests are mapped to actions through routing

Controller

● A controller is an instantiable class, usually public, in which at

least one of the following conditions is true

○ The class name is suffixed with Controller.

○ The class inherits from a class whose name is suffixed with

Controller.

○ The [Controller] attribute is applied to the class.

Action

● Public methods on a controller, except those with the [NonAction]

attribute, are actions. Parameters on actions are bound to request

data and are validated using model binding.

● Actions can return anything, but frequently return an instance of

IActionResult (or Task<IActionResult> for async methods) that

produces a response.

Action

Anything that could be returned

● HTTP Status Code

● Redirect

● View

● Formatted Response

In the context of an ASP.NET Core MVC application, a Controller is a class that is responsible for handling incoming HTTP requests and returning the appropriate HTTP response. Controllers are responsible for handling user input, performing any necessary business logic, and updating the Model and View to reflect the changes.

An Action is a method within a controller that is responsible for handling a specific type of request, such as a GET or a POST request. Each Action is typically associated with a specific URI, and it is responsible for performing the logic that is needed to handle the request, such as retrieving data from the database or performing some other kind of processing, and then returning the appropriate response to the client.

Each controller class can have multiple action methods, each one will handle different type of requests, routes, and HTTP verbs. For example, a controller class called **StudentController** can have the following action methods:

[HttpGet]

[Route("/students")]

public IActionResult GetAllStudents()

{

//logic to get all students

return View(students);

}

[HttpGet]

[Route("/students/{id}")]

public IActionResult GetStudent(int id)

{

//logic to get a specific student

return View(student);

}

The first action method will handle a GET request to the **/students** URI, it will get all the students and return the view that will display them. The second action method will handle a GET request to the **/students/{id}** URI, it will get a specific student based on the provided id and return the view that will display it.

In summary, Controllers are classes that handle incoming HTTP requests, while Actions are methods within controllers that handle specific types of requests, They handle the input and the logic of the application, and returns the appropriate response back to the client.

1. What is Routing? What are two types of routing?

Routing

Routing is the process through which the application matches an incoming URL

path and executes the corresponding action methods. ASP.NET Core MVC uses a

routing middleware to match the URLs of incoming requests and map them to

specific action methods.

Text

Description automatically generated

There are two types of routing for action methods:

● Conventional Routing

● Attribute Routing

Conventional Routing

When we create a new ASP.NET Core MVC application using the default template,

the application configures a default routing.

Text

Description automatically generated

\*You can define multiple conventions or to add routes that are dedicated to a specific action

Attribute Routing

By placing a route on the controller or the action method, we can make use of the

Attribute Routing feature.

● Route()

● Http[Verb]()

Text

Description automatically generated Text

Description automatically generated

In the context of an ASP.NET Core MVC application, routing is the process of mapping incoming URLs to specific controllers and actions. It is responsible for determining which controller and action method should handle a given request based on the URI of the request.

Routing is defined in the application's routing rules, which are typically defined in the **Startup.cs** file. These routing rules specify how the application should respond to incoming requests based on the URI of the request.

There are two types of routing in ASP.NET Core:

1. **Conventional Routing:** This is the default routing mechanism in ASP.NET Core. It uses a set of predefined conventions to map URLs to controllers and actions. In conventional routing, the routing rules are defined in the **Startup.cs** file and are based on the structure of the URI of the request and the names of the controllers and actions.

For example, a routing rule that maps the URI **/api/values** to the **ValuesController** and its **Get** action method, would be defined like this:

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "api/{controller=Values}/{action=Get}");

});

1. **Attribute routing:** This is an alternative routing mechanism in which the routing rules are defined directly on the controllers and actions using attributes. With attribute routing, you can define routes on the controllers and actions themselves, which allows you to more easily specify the URLs that map to specific actions.

For example, you can use the **[Route]** attribute to define a route on a controller like this:

[Route("api/[controller]")]

public class ValuesController : Controller

{

// Actions go here

}

This will tell the framework that all actions in the **ValuesController** should have the prefix **api/Values/**. You can also use **[Route]** attribute on action methods, like this:

[HttpGet]

[Route("{id}")]

public IActionResult Get(int id)

{

// Action logic

}

This will tell the framework that this action should be called when the client hits an endpoint like **api/Values/{id}** with GET method

Attribute routing provides a more flexible and powerful way to define routes in an ASP.NET Core MVC application and gives you greater control over how URLs are mapped to controllers and actions. Conventional routing and attribute routing can be used together in the same application, each one has its own use case and purpose, and you can decide which one to use based on the complexity of the routing rules, and your application's needs.

1. How can we pass data to Controller?

There are several ways to pass data to a controller in an ASP.NET Core MVC application:

1. **Query strings:** Data can be passed to a controller by appending it to the URI of the request as query string parameters. For example, you can pass a value for the "id" parameter in the URI like this: **/api/values?id=5**
2. **Route values:** Data can be passed to a controller by specifying it in the routing rules and binding it to the action method's parameters. For example, you can define a route that includes a parameter like this: **/api/values/{id}**, and bind it to the action method's parameters like this:

[HttpGet]

[Route("/api/values/{id}")]

public IActionResult Get(int id)

{

// Get the value with id

return Ok(value);

}

1. **Body of the request:** Data can also be passed to a controller by including it in the body of an HTTP POST or PUT request. For example, you can pass JSON or XML data in the body of a POST request, this will be deserialized into an object by the framework, and passed to the action method as a parameter.

[HttpPost]

[Route("/api/values")]

public IActionResult Post([FromBody]Value value)

{

// Add the value

return Ok();

}

1. **Form data:** Data can also be passed to a controller by including it in the form data of an HTTP POST request. The framework automatically maps the form data to the action method's parameters, or you can use **IFormCollection** to get all the form data at once

[HttpPost]

[Route("/api/values")]

public IActionResult Create(string name, int age)

{

// Create a new value

return Ok();

}

1. **Files:** Data can also be passed to a controller by including it as a file in an HTTP POST request. The framework will automatically map the files to the action method's parameters, or you can use **IFormFile** to get the file.

[HttpPost]

[Route("/api/values/upload")]

1. How can we pass data from Controller to View?

Pass Data from Controller to View

Pass data to views using several approaches:

● Strongly typed data: viewmodel

● Weakly typed data

○ ViewData ([ViewData] Attribute)

○ ViewBag

Strongly-typed data (viewmodel)

Specify a model type in the view. This model is commonly referred to as a

viewmodel. You pass an instance of the viewmodel type to the view from the action

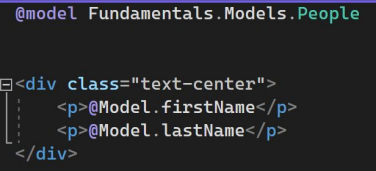
● Using a viewmodel to pass data to a view allows the view to take advantage of

strong type checking. Strong typing (or strongly typed) means that every

variable and constant has an explicitly defined type (for example, string, int, or

DateTime). The validity of types used in a view is checked at compile time.

Strongly-typed data (viewmodel)

● Specify a model using the @model directive. Use the model with @Model:  


Strongly-typed data (viewmodel)

● To provide the model to the view, the controller passes it as a parameter:

Text

Description automatically generated \*There are no restrictions on the model types that you can provide to a view. We recommend using Plain Old CLR Object

(POCO) viewmodels with little or no behavior (methods) defined

Weakly Typed Data

Weak types (or loose types) means that you don't explicitly declare the type of data you're using.

You can use the collection of weakly typed data for passing small amounts of data in and out of

controllers and views

This collection can be referenced through either the ViewData or ViewBag properties on

controllers and views.

● The ViewData property is a dictionary of weakly typed objects. T

● The ViewBag property is a wrapper around ViewData that provides dynamic properties for

the underlying ViewData collection

Weakly Typed Data (ViewData)

ViewData is a ViewDataDictionary object accessed through string keys

A picture containing text

Description automatically generated

Weakly Typed Data (ViewData)

Graphical user interface, text, application, email

Description automatically generated

Weakly Typed Data ([ViewData] attribute)

Graphical user interface, text, application, email

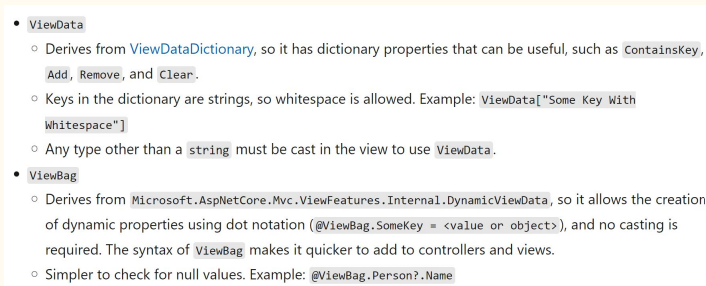
Description automatically generated

Weakly Typed Data (ViewBag)

Graphical user interface, text, application, Teams

Description automatically generated

ViewData vs. ViewBag



There are several ways to pass data from a controller to a view in an ASP.NET Core MVC application:

1. **ViewData and ViewBag:** Both ViewData and ViewBag are objects that are used to pass data from the controller to the view. They are dynamic objects that can be used to store key-value pairs of data, and their values can be accessed in the view using their keys.

For example, in the controller you can use ViewData to pass data to the view like this:

ViewData["message"] = "Hello World!";

And in the view, you can access it like this:

<h1>@ViewData["message"]</h1>

1. **ViewModel:** A view model is a class that contains the data that is needed by the view. You can create a new class that represents the data you want to pass to the view, you can then create an instance of this class and pass it to the view using the **View()** method.

For example, you can create a view model called **StudentViewModel**

public class StudentViewModel

{

public int Id { get; set; }

public string Name { get; set; }

public string Address { get; set; }

}

then you can create an instance of this class and pass it to the view like this:

var model = new StudentViewModel

{

Id = 1,

Name = "John Smith",

Address = "123 Main St"

};

return View(model);

Then in the view, you can access the properties of the view model like this:

<div>

<p>Id: @Model.Id</p>

<p>Name: @Model.Name</p>

<p>Address: @Model.Address</p>

</div>

1. **TempData:** TempData is a dictionary object that can be used to pass data between actions, it works like ViewData, but it's values are deleted after they have been read. It's mainly used to pass data between redirects, in which case you can use the **RedirectToAction()** method and pass data in it.

TempData["message"] = "Data has been saved!";

return RedirectToAction("Index");

then in the index action method you can access the data:

var message = TempData["message"];

In summary, there are several ways to pass data from a controller to a view in an ASP.NET Core MVC application, the most common ones are using ViewData, ViewBag, and ViewModels. Each one has its own use case, ViewData and ViewBag are good for passing a small amount of data, while ViewModels are useful when passing

1. If we deﬁne two endpoints,
2. Route(“/student/1”)
3. Route(“/student/{id}”)

And an incoming request localhost:7038/student/1.

Will there be any problem? Which endpoint will be invoked

If you define two endpoints as you described, with the routes **/student/1** and **/student/{id}**, and an incoming request is made to **localhost:7038/student/1**, there may be a problem, because the routing system will be unable to determine which endpoint should handle the request.

The routing system in ASP.NET Core MVC uses the order of the routes in the routing table to determine which route should handle a request. The routing system starts at the top of the routing table and compares each route to the URI of the incoming request until it finds a match.

In this case, both routes match the URI of the incoming request, **/student/1**. The first one is more specific, since it matches the exact URI of the request, while the second one **/student/{id}** is more generic, it matches the URI but also match any other value that the routing system bind into **{id}** parameter.

Therefore, the endpoint that will be invoked is the first endpoint, the one with the route **/student/1**, because the routing system will match it to the URI of the incoming request before it even reaches the second endpoint with the route **/student/{id}**.

It's important to consider this when designing your routing rules, in order to avoid potential conflicts between different routes, it's good practice to order your routes from most specific to most generic. Also, it's a good idea to test your application with different URLs to ensure that the routing

Top of Form

# Coding Questions:

**Coding 1:**

Continue on from yesterday’s coding question.

1. Inject CourseDAO to the CourseController

Creating all necessary action methods/endpoints inside the controller

\*If you ﬁnd there are more Controller or DAO needed, please feel free to add them

1. Create View pages based on the requirements to display the result or gather user actions In the end, we need to have a functioning application to handle Students and Courses (Bonus) \*Please feel free to add any additional features that you think will be helpful