# 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 are the features provided by ASP.NET Core

Fundamentals for ASP.NET Core ● Dependency Injection ● Configuration ● Middleware

Dependency Injection - Loose Coupling

ASP.NET Core includes dependency injection (DI) that makes configured services

available throughout an app. Services are added to the DI container with

WebApplicationBuilder.Services

The built-in container is represented by IServiceProvider implementation that

supports constructor injection by default

Dependency

You work in an organization where you and your colleagues tend to travel a

lot. Your typical travel planning routine might look like the following:

● Decide the destination, and desired arrival date and time

● Call up the airline agency and convey the necessary information to obtain a

flight booking.

● Call up the cab agency, request for a cab to be able to catch a particular flight

● Pickup the tickets, catch the cab and be on your way

Dependency

Now, what if your company suddenly changed the preferred agencies and their contact

mechanisms?

You would be subject to the following relearning scenarios:

● The new agencies, and their new contact mechanisms (say the new agencies offer internet based

services and the way to do the bookings is over the internet instead of over the phone)

● The typical conversational sequence through which the necessary bookings get done (Data instead of

voice)

Dependency

Now let’s say the protocol is a little bit different.

● There is an administration department in the company

● Whenever you needed to travel an administration department interactive telephony system simply

calls you up (which in turn is hooked up to the agencies)

● Over the phone you simply state the destination, desired arrival date and time by responding to a

programmed set of questions

● The flight reservations are made for you, the cab gets scheduled for the appropriate time, and the

tickets get delivered to you

When class A uses some functionality of class B, then its said that class A has a dependency of

class B. Diagram

Description automatically generated Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application, Word

Description automatically generated

Dependency Injection

What is dependency Inject?

● Separating the usage from the creation of the object

Why dependency injection? (or Decoupling)

● It improves the testability

● It's much easier to swap other pieces of code/modules/objects/ components when

the pieces are not dependent on one another

● Modularity — One module doesn't break other modules in unpredictable ways

IoC Container

● IoC — Inversion of Control

○ It is a process whereby objects define their dependencies (that is, the

other objects they work with) only through constructor arguments,

arguments to a factory method, or properties that are set on the object

instance after it is constructed or returned from a factory method.

○ Dependency injection is a pattern through which to implement IoC,

where the control being inverted is the setting of object's

dependencies.

● IoC container injects those dependencies when it creates the instance

Configuration

We use appsettings.json file to configure application level settings, which can be used across

the application and can be replaced the values at runtime without modifying the code.

● Load config data:

○ Load directly via Configuration object

■ Inject IConfiguration in the constructor in any class and then you can read all appsettings within this

object

○ Bind hierarchical configuration data using the options pattern

■ The options pattern uses classes to provide strongly typed access to groups of related settings from

appsettings.json file.

ASP.NET Core is a modern web development framework that provides a wide range of features and benefits for building web applications. Some of the key features provided by ASP.NET Core include:

1. Cross-platform support:

ASP.NET Core can be used to build applications that run on Windows, Linux, and macOS. This allows developers to build applications that can be deployed to a variety of platforms and environments.

1. High performance: ASP.NET Core is designed to be lightweight and efficient, resulting in fast and scalable applications.
2. Modular architecture: ASP.NET Core is built on a modular architecture that allows developers to only include the features and components that they need in their applications. This reduces the overall size of the application and improves performance.
3. Dependency injection (DI): ASP.NET Core includes built-in support for DI, which makes it easier to manage the dependencies of a class and improve the testability of an application.
4. Cloud-ready configuration: ASP.NET Core includes built-in support for configuration options that can be stored in a variety of sources, such as JSON files, environment variables, and Azure Key Vault. This makes it easier to deploy applications to the cloud and manage their configuration.
5. MVC pattern: ASP.NET Core includes support for the Model-View-Controller (MVC) pattern, which separates the concerns of the application into distinct layers for easier development and maintenance.
6. Razor Pages: ASP.NET Core includes Razor Pages, a new way to build web applications that is based on Razor syntax and provides a simpler and more efficient alternative to MVC for creating interactive web pages.
7. Mobile-ready: ASP.NET Core includes support for building mobile-ready applications that can be accessed from a variety of devices, including smartphones and tablets.
8. What are the advantages of ASP.NET Core over ASP.NET?

ASP.NET Core is a modern web development framework that was built from the ground up to be a lightweight, high-performance, and cross-platform framework for building web applications. There are several advantages that ASP.NET Core has over ASP.NET, including:

1. Cross-platform support: ASP.NET Core can be used to build applications that run on Windows, Linux, and macOS, whereas ASP.NET only runs on Windows. This allows developers to build applications that can be deployed to a wider range of platforms and environments.
2. Modular architecture: ASP.NET Core has a modular architecture that allows developers to only include the features and components that they need in their applications. This reduces the overall size of the application and improves performance. In contrast, ASP.NET includes a large number of features and components by default, which can make applications larger and slower.
3. Dependency injection (DI): ASP.NET Core includes built-in support for DI, which makes it easier to manage the dependencies of a class and improve the testability of an application. In contrast, ASP.NET does not have built-in support for DI and requires the use of third-party libraries to implement DI.
4. Cloud-ready configuration: ASP.NET Core includes built-in support for configuration options that can be stored in a variety of sources, such as JSON files, environment variables, and Azure Key Vault. This makes it easier to deploy applications to the cloud and manage their configuration.
5. Improved performance: ASP.NET Core is designed to be lightweight and efficient, resulting in faster and more scalable applications. In contrast, ASP.NET can be slower and less scalable due to its larger size and more complex architecture.
6. Where could we add services required by the application in ASP.NET Core MVC?

In an ASP.NET Core MVC application, services required by the application can be added in the **ConfigureServices** method of the **Startup** class. The **ConfigureServices** method is called during the application startup process and is used to configure the services that the application will use.

Here is an example of the **ConfigureServices** method in an ASP.NET Core MVC application:

public void ConfigureServices(IServiceCollection services)

{

// Add MVC services to the service collection.

services.AddControllersWithViews();

// Add other services to the service collection.

services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

services.AddScoped<IOrderService, OrderService>();

}

In this example, the **AddControllersWithViews** method adds MVC controllers and razor view engine services to the service collection, and the **AddDbContext** method adds a database context service to the service collection. The **AddScoped** method is used to add a custom **IOrderService** service to the service collection, and to specify that a new instance of the service should be created for each request within the same scope.

You can also use the **AddSingleton**, **AddTransient**, and **AddScoped** methods to specify the lifetime options for the services that you add to the service collection. For example, the **AddSingleton** method creates a single instance of a service that is shared by all clients, the **AddTransient** method creates a new instance of a service for each client, and the **AddScoped** method creates a new instance of a service for each request within the same scope.

1. Where could we conﬁgure the request pipeline by adding middleware and deﬁning how the application will respond to HTTP requests?

ASP.NET Core MVC – version before 6.0

● Wwwroot

○ Stores all the static files, ex. .css, .js, .html

● Areas

○ Identity system

● Startup.cs

○ Entry point. Setup configuration of how the app

will be launched

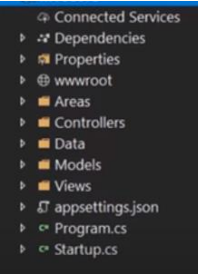
● Program.cs

○ Where this application starts

● Appsettings.json

○ Define all the configs

○ Eg. config for dev, test, prod



Startup.cs

● ConfigureServices(IServiceCollection services)

○ register service for dependency injection

● Configure(IApplicationBuilder app, IWebHostEnviornment env)

○ outline your middleware pipeline order and structure Text

Description automatically generated

.NET 6.0

Putting all services for dependency injection between builder and builder.Build().

Eg. add Authentication, add dbConnection, and etc.

In order to use those services, you need to put app.UseXXX() after builder.Builder()

Text

Description automatically generated

In an ASP.NET Core MVC application, you can configure the request pipeline by adding middleware and defining how the application will respond to HTTP requests in the **Configure** method of the **Startup** class. The **Configure** method is called after the **ConfigureServices** method and is used to configure the HTTP request pipeline of the application.

Here is an example of the **Configure** method in an ASP.NET Core MVC application:

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

});

}

In this example, the **UseDeveloperExceptionPage** and **UseExceptionHandler** middleware components are used to handle exceptions that occur during the execution of the application. The **UseHttpsRedirection** and **UseHsts** middleware components are used to enforce HTTPS communication. The **UseStaticFiles** middleware component is used to serve static files, such as images and CSS files, and the **UseRouting**, **UseAuthorization**, and **UseEndpoints** middleware components are used to enable routing, authorization, and endpoint mapping in the application.

you can add middleware to the request pipeline by calling the **Use** method of the **IApplicationBuilder** parameter in the **Configure** method and passing in an instance of the middleware component. The middleware component is then added to the pipeline and will be executed in the order that it is added.

For example, in the **Configure** method shown above, the **UseHttpsRedirection** middleware component is added to the pipeline first, followed by the **UseStaticFiles** middleware component. This means that the **UseHttpsRedirection** middleware component will be executed before the **UseStaticFiles** middleware component for each request.

You can also use the **Map** method of the **IApplicationBuilder** to specify that certain middleware components should only be executed for requests that match a specific pattern. For example:

app.Map("/api", apiApp =>

{

apiApp.UseMiddleware<ApiMiddleware>();

});

In this example, the **ApiMiddleware** middleware component will only be executed for requests that start with the "/api" path.

1. What is Kestral?

Kestrel is a cross-platform web server for ASP.NET Core applications. It is a lightweight and efficient web server that is built on top of the .NET Core platform, and is used to host ASP.NET Core applications in production.

One of the main benefits of Kestrel is its high performance and scalability. It is designed to be fast and efficient, and can handle a large number of concurrent connections without consuming a lot of resources. Kestrel also has a small footprint, making it well-suited for use in cloud environments where resources are limited.

In addition to its performance and scalability, Kestrel also has a number of other features and benefits:

* Cross-platform support: Kestrel runs on Windows, Linux, and macOS, allowing you to host your ASP.NET Core applications on a wide range of platforms.
* HTTP/2 support: Kestrel supports the HTTP/2 protocol, which can improve the performance of your applications by enabling multiplexing and header compression.
* Secure by default: Kestrel uses the Secure Sockets Layer (SSL) by default to encrypt communication between the server and client.
* Customizable request handling: Kestrel provides a flexible request handling pipeline that allows you to customize the way that requests are processed and responses are generated.
* Easy to use: Kestrel is easy to use and can be integrated with other ASP.NET Core components, such as the MVC framework and Entity Framework, to build powerful and scalable web applications.

1. What is middleware?

Middleware

A middleware is nothing but a component (class) which is executed on every request

in ASP.NET Core application Diagram

Description automatically generated

Middleware

● It has access to both Request and Response

● It may simply pass the Request to next Middleware

● It may process and then pass the Request to next Middleware

● It may handle the Request and short-circuit the pipeline

● It may process the outgoing Response

● Middlewares are executed in the order they are added to the pipeline

A picture containing chart

Description automatically generated Text

Description automatically generated

In the context of web development, middleware refers to software components that are executed on the server as part of the request-response cycle. Middleware components are typically executed in a pipeline, and are responsible for processing requests and generating responses.

There are many different types of middleware that can be used in web applications, including components that perform tasks such as:

* Handling exceptions
* Redirecting requests
* Enforcing security policies
* Serving static files
* Routing requests to the appropriate controller or action
* Modifying the request or response
* Logging request and response data

Middleware components are often added to the request pipeline in the **Configure** method of the **Startup** class in an ASP.NET Core MVC application. You can add middleware to the pipeline by calling the **Use** method of the **IApplicationBuilder** parameter in the **Configure** method and passing in an instance of the middleware component. The middleware component is then added to the pipeline and will be executed in the order that it is added.

1. How to read the conﬁguration from appsettings.json in your application?

To read the configuration from the **appsettings.json** file in an ASP.NET Core MVC application, you can use the **Configuration** property of the **Startup** class. The **Configuration** property is of type **IConfiguration** and represents a set of key/value application configuration properties.

To access the configuration values, you can use the **GetValue** or **GetSection** methods of the **IConfiguration** interface. For example:

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public void ConfigureServices(IServiceCollection services)

{

var connectionString = Configuration.GetValue<string>("ConnectionStrings:DefaultConnection");

services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(connectionString));

}

In this example, the **Configuration** property is injected into the **Startup** constructor by the dependency injection (DI) system, and the **GetValue** method is used to retrieve the value of the "ConnectionStrings:DefaultConnection" key from the **appsettings.json** file.

You can also use the **GetSection** method to retrieve a section of the configuration as an **IConfiguration** object, and then use the **Get** method to access the individual values within the section. For example:

var emailSection = Configuration.GetSection("EmailSettings");

var host = emailSection.GetValue<string>("Host");

var port = emailSection.GetValue<int>("Port");

I hope this helps to give you an idea of how to read the configuration from the **appsettings.json** file in an ASP.NET Core MVC application! Let me know if you have any questions or need further assistance.

1. What’s the JSON ﬁle that sets the launching conﬁg?

In an ASP.NET Core MVC application, the JSON file that sets the launching configuration is typically the **hosting.json** file. The **hosting.json** file is a configuration file that specifies settings for the ASP.NET Core hosting environment, such as the content root path, the URL to use for the application, and the environments in which the application can run.

Here is an example of a **hosting.json** file:

{

"server.urls": "http://localhost:5000",

"environment": "Development"

}

In this example, the **server.urls** setting specifies the URL to use for the application, and the **environment** setting specifies the environment in which the application is running.

The **hosting.json** file is optional, and is only used if you want to override the default hosting configuration. If you do not include a **hosting.json** file in your application, the default hosting configuration will be used.

----------------

The **appsettings.json** file is a configuration file in an ASP.NET Core MVC application that contains key/value pairs that can be used to configure the application. The **appsettings.json** file can contain any application-specific configuration settings that you want to specify, such as connection strings, email settings, or other application-specific settings.

Here is an example of an **appsettings.json** file:

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

"EmailSettings": {

"Host": "smtp.gmail.com",

"Port": 587,

"Username": "user@gmail.com",

"Password": "password"

}

}

In this example, the **appsettings.json** file contains two sections: "ConnectionStrings" and "EmailSettings". The "ConnectionStrings" section contains a connection string for a SQL Server database, and the "EmailSettings" section contains email server settings.

You can read the values from the **appsettings.json** file in your application by using the **Configuration** property of the **Startup** class. The **Configuration** property is of type **IConfiguration** and represents a set of key/value application configuration properties. You can use the **GetValue** or **GetSection** methods of the **IConfiguration** interface to access the configuration values.

1. How can we inject the service dependency into the controller?

To inject a service dependency into a controller in an ASP.NET Core MVC application, you can use the constructor injection pattern.

Here is an example of how you can inject a service dependency into a controller:

public class HomeController : Controller

{

private readonly IEmailSender \_emailSender;

public HomeController(IEmailSender emailSender)

{

\_emailSender = emailSender;

}

public IActionResult Index()

{

return View();

}

public IActionResult Contact()

{

return View();

}

[HttpPost]

public async Task<IActionResult> Contact(ContactViewModel model)

{

if (ModelState.IsValid)

{

await \_emailSender.SendEmailAsync(model.Email, model.Subject, model.Message);

return RedirectToAction(nameof(Index));

}

return View(model);

}

}

In this example, the **HomeController** has a constructor that takes an **IEmailSender** service as a parameter. The **IEmailSender** service is then stored in a private field, **\_emailSender**, which can be used by the controller's actions to send emails.

To use the constructor injection pattern, you will also need to register the service with the dependency injection (DI) system in the **ConfigureServices** method of the **Startup** class. You can do this by calling the **AddScoped** or **AddTransient** method of the **IServiceCollection** parameter, and passing in the service type and an implementation of the service:

public void ConfigureServices(IServiceCollection services)

{

services.AddScoped<IEmailSender, EmailSender>();

services.AddControllersWithViews();

}

In this example, the **IEmailSender** service is registered as a scoped service, which means that a new instance of the service will be created for each request. You can also use the **AddTransient** method to register the service as a transient service, which means that a new instance of the service will be created each time it is requested.

1. Naming the service life?

Service Lifetimes

Services can be registered with one of the following lifetimes:

● Transient

● Scoped

● Singleton

Service Lifetimes - Transient

Transient lifetime services are created each time they're requested from the service container. 

\*Transient is considered the "default" service lifetime in .NET 6. This means that you should

make all dependencies transient unless they truly need to use one of the other service lifetimes

Service Lifetimes - Scoped

A scoped lifetime indicates that services are created once per client request

(connection)



\*When using Entity Framework Core, the AddDbContext extension method

registers DbContext types with a scoped lifetime by default.

Service Lifetimes - Singleton

Singleton lifetime services are created either:

● The first time they're requested.

● By the developer, when providing an implementation instance directly

to the container. This approach is rarely needed

\*In apps that process requests, singleton services are disposed when the

ServiceProvider is disposed on application shutdown

DI Container

Dependency Inject in .NET can be done in three ways:

● Constructor

● Setter (by default not supported, need built-in/3rd-party DI Container)

● Property (by default not supported, need built-in/3rd-party DI Container) Graphical user interface, text, application, Teams

Description automatically generated

# Coding Questions:

Write code in C#/.NET Core MVC to solve following problems. Please write your own answers. You are highly encouraged to present more than one way to answer the questions. Please follow best practice when you write the code so that it would be easily readable, maintainable, and eﬃcient. Clearly state your assumptions if you have any. You may discuss with others on the questions, but please write your own code.

Develop an **ASP.NET Core MVC Application** that uses ADO.NET to communicate with database.

## Tables:

* Student(id, ﬁrstname, lastname, email)
* Student\_Course (id, studentIid, courseId)
* Course(id, name, description, professorId)
* Professor(id, ﬁrstname, lastname, email, oﬃce, title)

【Relationships:

* A student can take any number of courses. A course can be taken by different students.
* Each course is only taught by 1 professor. A professor can teach more than 1 courses.
* No direct relationship between students and professors. 】

## Onboard Models to ASP.NET Core MVC Application based on the table deﬁned

1. **Create a folder named DAO and a CourseDAO.cs within it which includes the following methods:**

Hint: add connectionString to the appsettings.json and load the connectionString via conﬁguration object

* 1. Add
     1. AddStudent - this method takes a student object and saves it to the db
     2. AddCourse - this method takes a course object and saves it to the db
     3. AddProfessor - this method takes a professor object and saves it to the db
     4. AssignStudentToCourse - this method takes a studentId and a courseId and saves the information to the db
     5. AssignProfessorToCourse - this method takes two inputs: a prodessorId and a courseId, and update the information to the db
  2. Update
     1. UpdateStudent – this method takes a student object and updates it to the db
     2. UpdateCourse - this method takes a course object and Updateit to the db
     3. UpdateProfessor - this method takes a professor object and Updateit to the db
  3. Data Retrieval
     1. FindStudentCoursesByStudentEmail - this method takes an email string and returns a List of courses he/she takes
     2. FindProfessorCoursesByName - this method takes a name string (including both ﬁrstname and lastname) and returns a List of courses he/she taught
     3. FindCourseById - this method takes a courseId and returns a Course object along with the professor’s information

## Inject this CourseDAO.cs class to the HomeController and verify your behaviors there