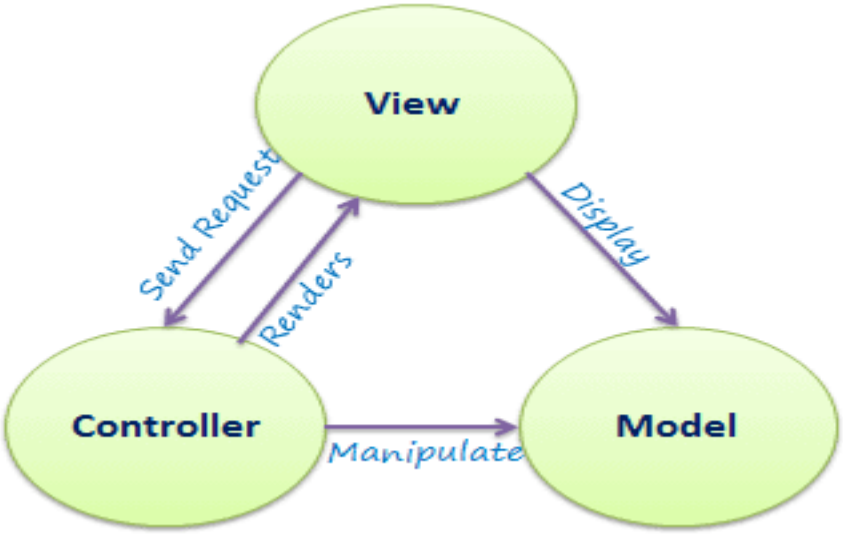
1. Describe the MVC pattern. Create class to showcase constructor, properties, indexers and encapsulation behavior of object oriented language.

Solution:

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application.



Model: Model represents the structure of data, the format and the constraints with which it is stored. It maintains the data of the application. Essentially, it is the database part of the application.

View: View is what is presented to the user. Views utilize the Model and present data in a form in which the user wants. A user can also be allowed to make changes to the data presented to the user. They consist of static and dynamic pages which are rendered or sent to the user when the user requests them.

Controller: Controller controls the requests of the user and then generates appropriate responses which are fed to the viewer. Typically, the user interacts with the View, which in turn generates the appropriate request, this request will be handled by a controller. The controller renders the appropriate view with the model data as a response.

Second Part:

```
namespace CSharpExamples

{

    class Customer

    {
```

```
    private string name;
// Properties
    public int CustomerID { get; set; }
        //This will encapsulate the private variable name. name can only be accessed from this property from other classes.
    public string Name
    {
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
    }
//Indexer
    private string[] purchasedItem = new string[10];
    public string this[int i]
    {
    get
    {
        return purchasedItem[i];
    }
```

```
        set
        {
                purchasedItem[i] = value;
        }
        }
        public int Length
        {
        get { return purchasedItem.Length; }
        }
      // Constructor
        public Customer(int ID, string name)
        {
        Name = name;
        CustomerID = ID;
        }
        // .. Additional methods, events, etc.
    }
// Test class
 class Program
        {
        static void Main(string[] args)
        {
        //Using User defined constructor
        Customer cust = new Customer(1, "Jayanta");
```

```
        //Adding purchased items to demonstrate Indexer
        for (int i = 0; i < 10; i++)
        {
            cust[i] = "Item" + i;
        }
        Console.WriteLine("The customer Name is " + cust.Name + " And ID is " + cust.CustomerID);
        Console.WriteLine(cust.Name + " Purchased following items");
        for (int j = 0; j < cust.Length; j++)
        {
            Console.WriteLine(cust[j]);
        }
        Console.ReadLine();
        }
        }
}
```

2. Explain the architecture and design principles of .NET. Create methods to insert, update, delete and read all data for the table Student having following fields StudentId(int), Name varchar(200), RollNo (int), Class varchar(50) using Entity Framework.

Solution:

.NET is tiered, modular, and hierarchical. Each tier of the .NET Framework is a layer of abstraction. .NET languages are the top tier and the most abstract level. The common language runtime is the bottom tier, the least abstracted, and closest to the native environment. This is important since the common language runtime works closely with the operating environment to manage .NET applications. The .NET Framework is partitioned into modules, each with its own distinct responsibility. Finally, since higher tiers request services only from the lower tiers, .NET is hierarchical. The architectural layout of the .NET Framework is given below.

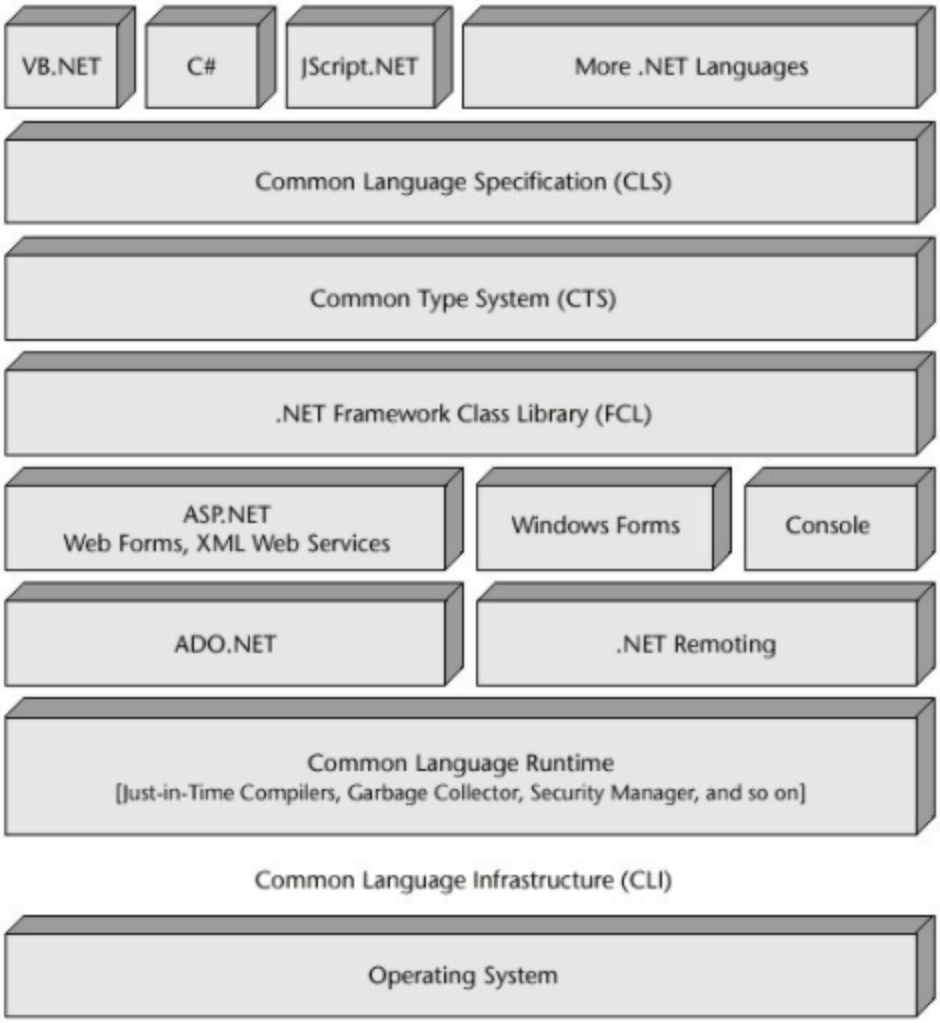


Figure: An overview of the .NET architecture.

- *Common Language Runtime (CLR)* is the heart of the .Net Framework. It resides above the operating system and handles all .Net applications. It handles garbage collection, Code Access Security (CAS) etc.
- *Common Language Infrastructure (CLI)* provides a language-independent platform for app development and its performance. It also includes the function for exception handling, waste collection, security, etc.
- *Common Type System(CTS)* specifies a standard that represent what type of data and value can be defined and managed in computer memory at runtime. For example, in C# we define data type as int, while in VB.NET we define integer as a data type.
- *Common language Specification(CLS)* is a subset of common type system (CTS) that defines a set of rules and regulations which should be followed by every language that comes under the .net framework. For example, in C# and VB.NET language, the C# language terminate each statement with semicolon, whereas in

VB.NET it is not end with semicolon, and when these statements execute in .NET Framework, it provides a common platform to interact and share information with each other.

- ***FCL (Framework Class Library)*** provides the various system functionality in the .NET Framework, that includes classes, interfaces and data types, etc. to create multiple functions and different types of application such as desktop, web, mobile application, etc.

Design principles of .NET:

- Interoperability
- Portability
- In-built security mechanism
- Robust memory management
- Simplified deployment
- Asynchronous Programming
- High Performance

Second Part

```
namespace CSharpExamples
{
    class EFCoreTest
    {
    private StudentContext _context = new StudentContext();
    public List<Student> GetAll()
    {
    return _context.tblStudent.ToList();
    }
    public void InsertStudent(Student emp)
    {
    _context.Add(emp);
    _context.SaveChanges();
```

```
}
public int EditStudent(Student emp)
{
if (emp.StudentId == -1)
{
        return 0;
}
var stud = _context.tblStudent.Find(emp.StudentId);
if (stud == null)
{
        return 0;
}
int updatedCount = 0;
try
{
_context.Update(stud);
updatedCount = _context.SaveChanges();
}
catch (DbUpdateConcurrencyException)
{
        return 0;
}
return updatedCount;
}
```

```
public int deleteStudent(int StudentId)
{
if (StudentId == -1)
{
        return 0;
}
var stud = _context.tblStudent
        .FirstOrDefault(m => m.StudentId == StudentId);
if (stud == null)
{
        return 0;
}
_context.tblStudent.Remove(stud);
int updatedCount =  _context.SaveChanges();
return updatedCount;
}
public void showStudents(List<Student> students)
{
foreach (Student student in students)
{
        Console.WriteLine("Student ID = " + student.StudentId);
        Console.WriteLine("Student Name = " + student.Name);
  Console.WriteLine("Student RollNo = " + student.RollNo);
  Console.WriteLine("Student Class = " + student.Class);
```

```
        }
        }
    }
Class Program
        {
        static void Main(string[] args)
        {
        EFCoreTest test = new EFCoreTest();
            //Read All Students
        List<Employee> employees = test.GetAll();
        Console.WriteLine("Initial Records");
        test.showStudents(employees);
         //Insert Student
        Employee stud = new Employee();
        stud.Name = "Ram";
              stud.RollNo = 31;
            stud.Class = "Sixth Sem";
        test.InsertStudent(stud);
        Console.WriteLine("After Insert");
        test.showStudents(employees);
         //Update Student
        stud.Name = "UpdatedName";
        test.EditStudent(stud);
        Console.WriteLine("After Update");
```

```
        test.showStudents(employees);

        //Delete Student

        test.deleteStudent(stud.EmployeeId);

        Console.WriteLine("After Delete");

        test.showStudents(employees);

        Console.ReadLine();

        }

        }

}
```

3. How does the system manage state in stateless HTTP? Design a page to show client side validation for login page using jquery or angular or react.

Solution:

HTTP is called a stateless protocol because each command request is executed independently, without any knowledge of the requests that were executed before it.

A few techniques can be used to maintain state information across multiple HTTP requests:

1. Cookies: A web server can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

2. Hidden fields of the HTML form: Web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type = "hidden" name = "sessionid" value = "12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the **GET** or the **POST** data. Each time the web browser sends the request back, the **session_id** value can be used to keep the track of different web browsers.

3. URL rewriting: We can append some extra data at the end of each URL. This data identifies the session; the server can associate that session identifier with the data it has stored about that session. For example, with **https://collegenote.com/file.htm;sessionid=12345**, the session identifier is attached as **sessionid = 12345** which can be accessed at the web server to identify the client.

Client Side Validation using jQuery

```
<html>

<head>

<title>Jquery login page validation</title>

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js"></script>

<script type="text/javascript">

    $(document).ready(function(){

        $('#submit').click(function(){

            var username=$('#user').val();

            var password=$('#pass').val();

            if(username=="")

            {

                $('#dis').slideDown().html("<span>Please type Username</span>");

                return false;

            }

            if(password=="")

            {

                $('#dis').slideDown().html('<span id="error">Please type Password</span>');

                return false;

            }
```

```
        });
    });
</script>
</head>
<body>
  <fieldset style="width:250px;">
    <form method="post" action="">
      <label id="dis"></label><br>
      Username: <input type="text" name="user" id="user" /><br />
      Password: <input type="password" name="pass" id="pass" /><br /><br />
      <center><input type="submit" name="submit" id="submit" /></center>
    </form>
  </fieldset>
</body>
</html>
```

4. Write an application showing sql injection vulnerability and prevention using ado.net.

Solution:

Consider the following action method that validates user login.

```
[HttpPost]
public IActionResult SubmitLogin1(String uname, String pwd)
{
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;Initial Catalog=db_Mac1; Integrated Security=True");
```

```
    con.Open();

        SqlCommand cmd = new SqlCommand("select * from tbl_login where uname=' "+uname+" ' and password=' "+pwd+" ' ", con);

    SqlDataReader dr = cmd.ExecuteReader();

    if (dr.Read())

    {

        return Content ("Login Successful");

    }

    else

    {

        return Content("Login Unsuccessful");

    }

}
```

The above action method is vulnerable to SQL injection attack. It is because we've used the form input values name and pwd with no data validation at all including Empty form validations. Nothing bad will happen if we're sure that this value will only come from trusted sources, but this is not always. If the attacker doesn't know what the username is then he/she simply provides a ' ' or 1=1 for the username. So, when the user presses the submit button the resulting query will be formed as

```
  select * from tbl_login where Username = ' ' or 1=1-- ' and Password = ' '
```

The above query will return entire rows from table tbl_login if there is atleast one row in the table thereby displays "Login Successful" message. Anything placed into that TextBox control will be added to your SQL string. This situation invites a hacker to replace that string with something malicious.

Preventing SQL Injection:

Using parameterized query will prevent such injection. Using parameterized queries is a three-step process:

1. Construct the SqlCommand command string with parameters.

2. Declare a SqlParameter object, assigning values as appropriate.

3. Assign the SqlParameter object to the SqlCommand object's Parameters property.

Program:

```
[HttpPost]

public IActionResult SubmitLogin1(String uname, String pwd)

{

    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;Initial Catalog=db_Mac1; Integrated Security=True");

  con.Open();

    SqlCommand cmd = new SqlCommand("select * from tbl_login where uname = @uname and password = @pwd ", con);

  cmd.Parameters.AddWithValue("@uname", uname);

  cmd.Paramterers.AddWithValue("@pwd", pwd);

  SqlDataReader dr = cmd.ExecuteReader();

  if (dr.Read())

  {

    return Content ("Login Successful");

  }

  else

  {

    return Content("Login Unsuccessful");

  }

}
```

5. How do you host and deploy the ASP.NET core application?
Solution:
There are 2 types of hosting models in ASP.NET Core:

1. Out-of-process Hosting Model: In Out-of-process hosting models, we can either use the Kestrel server directly as a user request facing server or we can deploy the app into IIS which will act as a proxy server and sends requests to the internal Kestrel server. In this type of hosting model, we have two options:

- *Using Kestrel:* Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the web server that's included by default in ASP.NET Core project templates. Kestrel itself acts as an edge server which directly server user requests. It means that we can only use the Kestrel server for our application.
- *Using a Proxy Server:* Due to limitations of the Kestrel server, we cannot use this in all the apps. In such cases, we have to use powerful servers like IIS, NGINX or Apache. So, in that case, this server acts as a reserve proxy server which redirects every request to the internal Kestrel server where our app is running. Here, two servers are running. One is IIS and another is Kestrel.

2. In-process Hosting Model: In this type, only one server is used for hosting like IIS, Nginx or Linux. It means that the App is directly hosted inside of IIS. No Kestrel server is being used. IIS HTTP Server (IISHttpServer) is used instead of the Kestrel server to host apps in IIS directly.

Steps to Deploy ASP.NET Core to IIS:

Step 1: Publish to a File Folder. Publish to Folder With Visual Studio.

Step 2: Copy Files to Preferred IIS Location.

Step 3: Create Application in IIS.

Step 4: Load Your App!

6. Describe the process of adding authentication to apps and identify service configurations.
Solution:

Services are added in ConfigureServices.The typical pattern is to call all the Add{Service} methods, and then call all the services.Configure{Service} methods.

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddDbContext<ApplicationDbContext>(options =>
        // options.UseSqlite(
        options.UseSqlServer(
        Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(
```

```
        options=>options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
         services.AddRazorPages();
        services.Configure<IdentityOptions>(options =>
        {

        //Password settings.
        options.Password.RequireDigit = true;
        options.Password.RequireLowercase=true;
        options.Password.RequireNonAlphanumeric=true;
        options.Password.RequireUppercase = true;
        options.Password.RequiredLength = 6;
        options.Password.RequiredUniqueChars = 1;

        //Lockout settings.
        options.Lockout.DefaultLockoutTimeSpan=TimeSpan.FromMinutes(5);
        options.Lockout.MaxFailedAccessAttempts=5;
        options.Lockout.AllowedForNewUsers = true;

        // User settings.
        options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
        789-._@+";
        options.User.RequireUniqueEmail = false;
        });
        services.ConfigureApplicationCookie(options =>
        {

        // Cookie settings
        options.Cookie.HttpOnly=true;
        options.ExpireTimeSpan=TimeSpan.FromMinutes(5);
        options.LoginPath = "/Identity/Account/Login";
        options.AccessDeniedPath="/Identity/Account/AccessDenied";
        options.SlidingExpiration = true;
        });
}
```

The preceding highlighted code configures Identity with default option values. Services are made available to the app through dependency injection.

Identity is enabled by calling UseAuthentication. UseAuthentication adds authentication middleware to the request pipeline.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
        if (env.IsDevelopment()) {
```

```
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        }
else {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();
        app.UseAuthentication();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
        endpoints.MapRazorPages();
        });
}
```

The template-generated app doesn't use authorization. app.UseAuthorization is included to ensure it's added in the correct order should the app add authorization.
UseRouting, UseAuthentication, UseAuthorization, and UseEndpoints must be called in the order shown in the preceding code.

7. Differentiate between abstract class, sealed class and interface. What is the task of an Object Relational Mapper?
Solution:
Abstract Class

A class is said to be an abstract class, if we can't instantiate an object of that class without creating its subclass. Such a class only exists as a parent of derived classes from which objects are instantiated. Abstract Class contains at least one abstract method. Abstract classes are used to provide an Interface for its subclasses. Classes inheriting an Abstract Class must provide definition to the abstract method.

Example:

```
using System;

namespace ConsoleApp

{

  class Program
```

```
  {
    static void Main(string[] args)
    {
       Demo d;     //d is Reference Type variable of Abstract Class Demo
       Demo1 d2 = new Demo1();
       d = d2;
       d.Display();
       Console.ReadKey();
          }
   }
   abstract class Demo     //Abstract Class
   {
       public abstract void Display();     //Abstract Method
   }
   class Demo1 : Demo     //Concrete Class
   {
       public override void Display()
   {
       Console.Write("Derived Class Method Invoked");
   }
  }
}
```

Sealed Class

Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the sealed keyword or modifier. This keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class.

Example:

```
class Demo

{

}

sealed class Demo1:Demo

{

}

class Demo2 : Demo1      //This Statement generates compile time error as it attempts to inherit Sealed Class

{

}
```

Interface

An interface is a named collection of methods declaration without implementations/definition. Interface define what a class must do but not how it does. To declare an interface, we use *interface* keyword. A class that implements interface must implement all the methods declared in the interface.

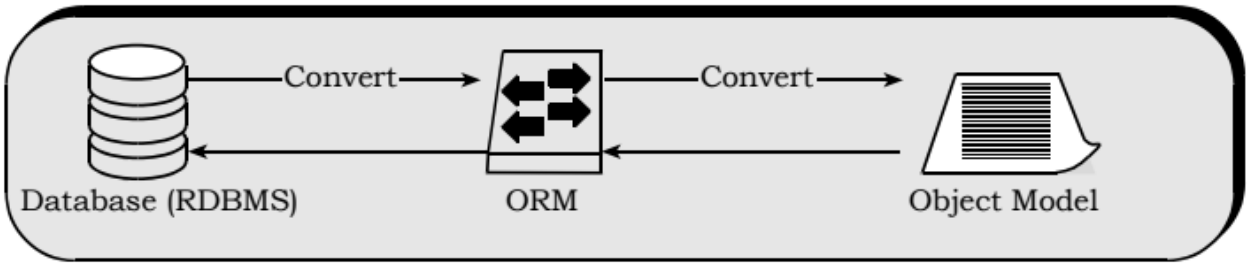
Syntax: Creating Interface

```
interface <interface_name >

{

   //Methods Declaration

}
```

Syntax: Implementing Interface

```
class class_name :< interface_name>
```

An Object Relational Mapper (ORM) is an application or system that support in the conversion of data within a relational database management system (RDBMS) and the object model that is necessary for use within object-oriented programming.

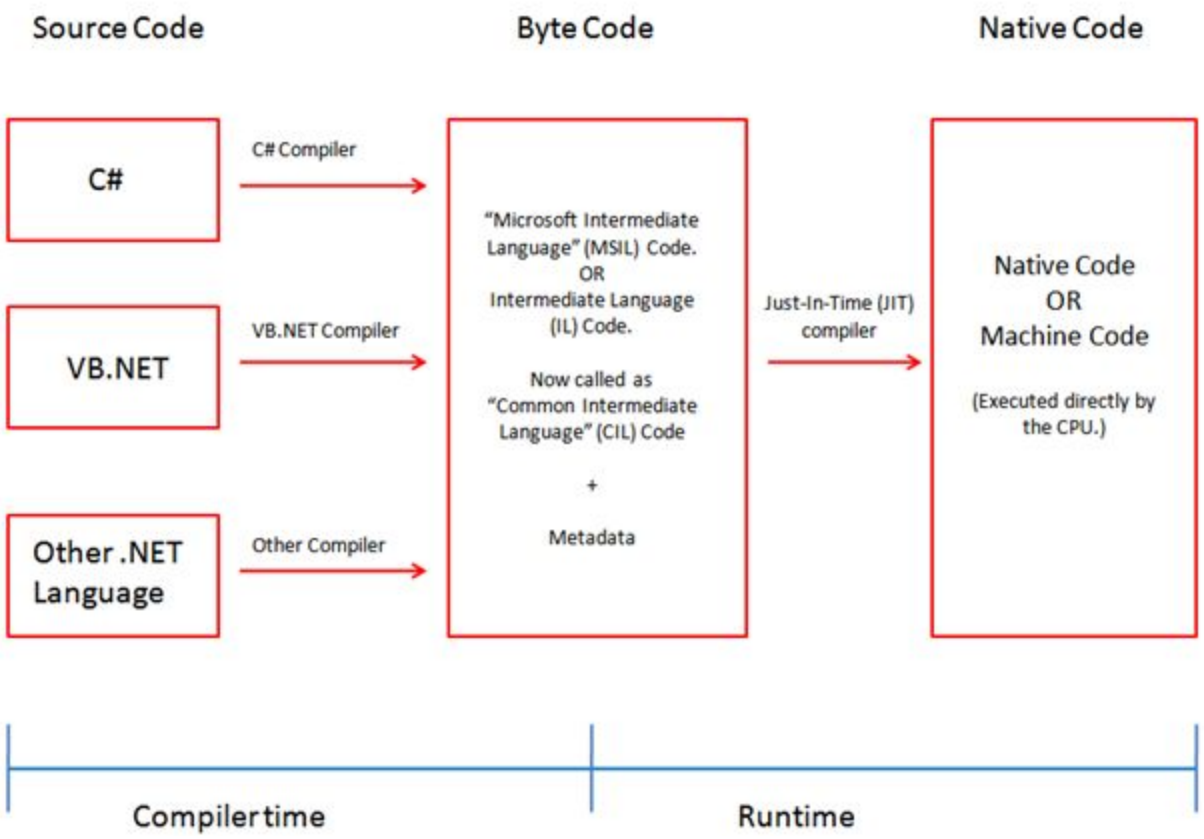


8. Explain the process of compiling and executing .NET applications.
Solution:

C# programs run on the .NET Framework, which includes the common language runtime (CLR) and a unified set of class libraries. The CLR is the commercial implementation by Microsoft of the common language infrastructure (CLI), an international standard that is the basis for creating execution and development environments in which languages and libraries work together seamlessly.

Source code written in any .NET languages (C#, VB.Net, etc.) is compiled into an Microsoft Intermediate Language (MSIL) or simply(IS) that conforms to the CLI specification. The IL code are stored on disk in an executable file called an assembly, typically with an extension of .exe or .dll. CLR performs just in time (JIT) compilation to convert the IL code to native machine instructions. The CLR also provides other services related to automatic garbage collection, exception handling, and resource management.

Code that is executed by the CLR is sometimes referred to as "managed code," in contrast to "unmanaged code" which is compiled into native machine language that targets a specific system.



9. Why do we need generics? What are the significances of MSIL?
Solution:

Generic is common to write a program that processes a collection - e.g. a collection of numbers, a collection of contacts, a collection of Name, etc. With generic programming, we can write code that handles a collection 'in the general' and C# handles the specifics for each collection type, saving you a great deal of work.

Generic collection in C# is defined in System.Collection.Generic namespace. It provides a generic implementation of standard structure like List, Stack, Queues, Dictionaries etc.

E.g.

```
using System;

using System.Collections.Generic;

public class GenericList

{

   public static void GenericListMethod()
```

```
    {
        List<int> genericList = new List<int>();
        // No boxing, no casting:
        genericList.Add(12);
        genericList.Add(13);
        genericList.Add(14);
        genericList.Add(15);
    }
}
```

There are mainly two reasons to use generics as in the following:

1. Performance: Collections that store the objects uses boxing and unboxing on data types. A collection can reduce the performance. By using generics it helps to improve the performance and type safety.

2. Type Safety: there is no strong type information at compile time as to what it is stored in the collection.

Microsoft Intermediate Language (MSIL)

During the compile time , the compiler converts the source code into Microsoft Intermediate Language (MSIL). Microsoft Intermediate Language (MSIL) is a CPU-independent set of instructions that can be efficiently converted to the native code. During the runtime the Common Language Runtime (CLR)'s Just In Time (JIT) compiler converts the Microsoft Intermediate Language (MSIL) code into native code to the Operating System.

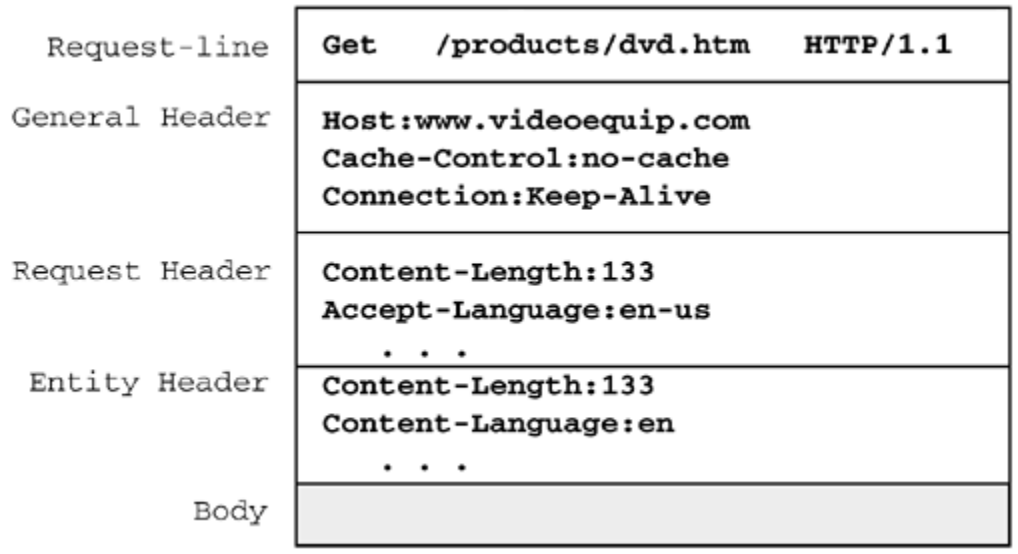
When a compiler produces Microsoft Intermediate Language (MSIL), it also produces Metadata. The Microsoft Intermediate Language (MSIL) and Metadata are contained in a portable executable (PE) file . Microsoft Intermediate Language (MSIL) includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations.

10. Express the format of request and response message format. What is the role of adapter class in database connection?

Solution:

HTTP specifications concisely define messages as "requests from client to server and responses from server to client." At its most elemental level, the role of ASP.NET is to enable server-based applications to handle HTTP messages. The primary way it does this is through HttpRequest and HttpResponse classes. The request class contains the HTTP values sent by a client during a Web request; the response class contains the values returned to the client.

HTTP Request Message Format

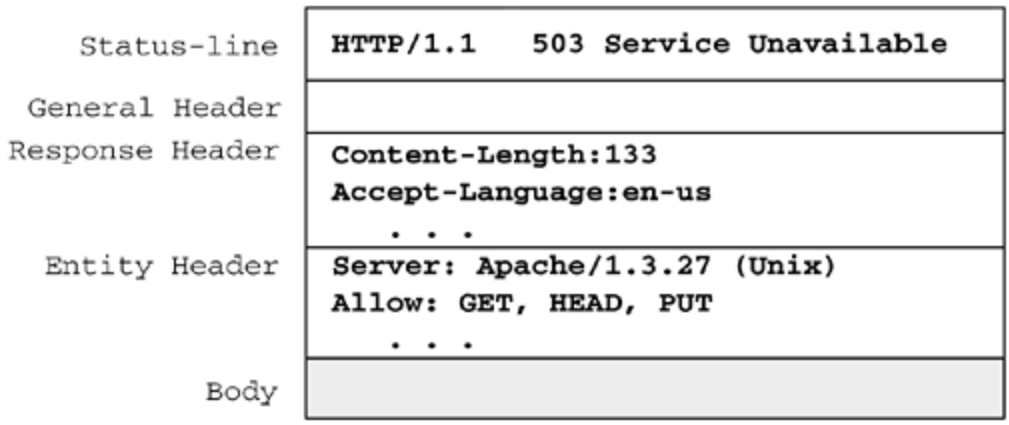


- The *Request-Line* consists of three parts: the method token, the Request-URI, and the protocol version. Several methods are available, the most common being the POST and GET methods. The *Uniform Resource Identifier* (URI) specifies the resource being requested. This most commonly takes the form of a Uniform Resource Locator (URL), but can be a file or other resource. The protocol version closes out the *Request-Line*.

- The *general-header* is used to pass fields that apply to both requests and responses. The most important of these is the *cache-control* field that specifies, among other things, what can be cached and how long it can be cached.

- The *request-header* is used by the client to pass additional information about the request, as well as the client. Most of the *HttpRequest* object's properties correspond to values in this header.

HTTP Response Message Format



The server responds with a *status line* that includes the message's protocol version, a success or error code, and a textual description of the error code. This is followed by the *general header* and the *response header* that provides information about the server. The *entity header* provides metainformation about the body contents or the resource requested.

Role of adapter class in database connection

The DataAdapter serves as a bridge between a DataSet and a data source for retrieving and saving data. The DataAdapter provides this bridge by mapping Fill, which changes the data in the DataSet to match the data in the data source, and Update, which changes the data in the data source to match the data in the DataSet.

11. How do you render HTML with views? Explain.
Solution:

In MVC pattern, the view handles the app's data presentation and user interaction. A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client. In ASP.NET Core MVC, views are *.cshtml* files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in a Views folder at the root of the app.

Views that are specific to a controller are created in the *Views/[ControllerName]* folder. To create a view, add a new file and give it the same name as its associated controller action with the *.cshtml* file extension.

The Home controller is represented by a Home folder inside the Views folder. The Home folder contains the views for the About, Contact, and Index (homepage) webpages. When

a user requests one of these three webpages, controller actions in the Home controller determine which of the three views is used to build and return a webpage to the user.

Example: The following program shows how a view is displayed

1. HTML View Defined as /Views/Home/Index in file Index.cshtml

```
<h1>This is Index Page </h1>
```

2. Action Method Index() defined at Controller/Home in file HomeController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace MyMVcApplication.Controller

{

  public class HomeController : Controller

  {

    public IActionResult Index()

      {

        return View();

      }

  }

}
```

12. Write short notes on (Any TWO)

a. Single page application

b. Hidden fields

c. Await patterns

Solution:

a) Single Page Application:

A single-page application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages. The goal is faster transitions that make the website feel more like a native app.

In a single-page application, all necessary HTML, JavaScript, and CSS code is either retrieved by the browser with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does it transfer control to another page, although the location hash or the HTML5 History API can be used to provide the perception and navigability of separate logical pages in the application.

b) Hidden Fields

HiddenField, as the name implies, is hidden. Sometimes we require some data to be stored on the client side without displaying it on the page. This is non visual control in Asp.net Core where we can save the value. This is one of the client-side state management tools. It stores the value between the roundtrip. We can save data in hidden form fields and send it back in the next request. For example:

```
[HttpGet]

public IActionResult SetHiddenFieldValue() {

User newUser = new User() {

   Id = 210, Name = "Sushma", Age = 23

};

return View(newUser);

}

[HttpPost]

public IActionResult SetHiddenFieldValue(IFormCollection keyValues) {

   var id = keyValues["Id"];

   return View();

}
```

In the View, we can create a hidden field and bind the Id value from Model: @Html.HiddenFor(model =>model.Id)

Then we can use a submit button to submit the form: <input type="submit" value="Submit" />

The general html code for the hidden field look like this when it is inspected:

```
<input type = "hidden" id = "Id" name = "Id" value="210">
```

c) Await Patterns

The await keyword provides a non-blocking way to start a task, then continue execution when that task completes.

The await keyword is used to asynchronously wait for a Task or Task<T> to complete. It pauses the execution of the current method until the asynchronous task that's being *await*ed completes. The difference from calling .Result or .Wait() is that the *await keyword sends the current thread back to the thread pool*, instead of keeping it in a *blocked* state. For example:

```
class Program
{
   static void Main(string[] args)
   {
      Method1();
      Method2();
      Console.ReadKey();
   }

   public static async Task Method1()
   {
      await Task.Run(() =>
      {
         for (int i = 0; i < 100; i++)
         {
```

```
            Console.WriteLine(" Method 1");
            // Do something
            Task.Delay(100).Wait();
        }
    });
}

public static void Method2()
{
    for (int i = 0; i < 25; i++)
    {
        Console.WriteLine(" Method 2");
        // Do something
        Task.Delay(100).Wait();
    }
}
}
```