Razor Views

Razor views are an integral part of the ASP.NET Core MVC framework. They are used to define the user interface of web pages in an ASP.NET Core application. Razor views combine HTML markup with C# code to enable dynamic rendering of content and data-driven views. Razor views provide a simple and intuitive syntax that allows developers to seamlessly mix server-side C# code with client-side HTML.

Key features of Razor views include:

1. **Syntax**: Razor views use a combination of HTML and C# code within the same file. C# code is encapsulated within `@` symbols, allowing developers to switch between HTML and C# easily.

2. **Model Binding**: Razor views support model binding, which enables the automatic transfer of data from a model (C# class) to the view. This allows developers to pass data from the controller to the view without explicitly setting each data element in the view.

3. **Code Reuse**: Razor views support the use of partial views, which are reusable components that can be shared across multiple views. Partial views enable code modularity and help in reducing duplication.

4. **Layouts**: Razor views can be structured using layout files that define the common structure of multiple pages. Layouts allow developers to maintain consistent headers, footers, and navigation across the application.

5. **Tag Helpers**: Razor views support tag helpers, which are special attributes used to simplify HTML generation for specific ASP.NET Core components. Tag helpers resemble HTML elements but have additional functionalities on the server-side.

Here's a simple example of a Razor view:

```html
@model EmployeeManagement.Models.Employee

<!DOCTYPE html>
<html>
<head>
    <title>Employee Details</title>
</head>
<body>
    <h1>Hello, @Model.Name!</h1>
    <p>Age: @Model.Age</p>
    <p>Designation: @Model.Designation</p>
    <p>Department: @Model.Department</p>
</body>
</html>
```

In this example, the `@model` directive indicates that the view is strongly typed to the `EmployeeManagement.Models.Employee` class. The `@Model` object allows us to access properties of the `Employee` class (e.g., `@Model.Name`, `@Model.Age`, etc.) to render dynamic content.

Razor views play a vital role in ASP.NET Core applications, as they enable developers to build dynamic and data-driven user interfaces, making web

development more efficient and productive.

HTML Helper Functions

HTML Helper functions are a feature in ASP.NET Core MVC that provides a set of methods to generate HTML elements and other components in Razor views using C# code. HTML Helpers simplify the process of generating HTML markup and make it easier to work with dynamic data, form elements, URLs, and more. They allow you to write cleaner and more concise Razor views.

ASP.NET Core MVC provides various built-in HTML Helper functions that can be used within Razor views to generate HTML elements and perform common HTML-related tasks. These HTML Helper functions are accessible through the `Html` property within Razor views.

Here are some commonly used HTML Helper functions and their purposes:

1. `Html.DisplayNameFor`:
   - Generates the display name for a property based on the model's metadata.
   - Example: `@Html.DisplayNameFor(model => model.FirstName)`

2. `Html.DisplayFor`:
   - Generates the HTML representation of a model property based on the model's metadata.
   - Example: `@Html.DisplayFor(model => model.LastName)`

3. `Html.TextBoxFor`:

   - Generates an HTML input element of type `text` for a model property.

   - Example: `@Html.TextBoxFor(model => model.Email, new { @class = "form-control" })`

4. `Html.DropDownListFor`:

   - Generates an HTML `select` element (drop-down list) for a model property.

   - Example: `@Html.DropDownListFor(model => model.Department, departmentsList, "Select Department", new { @class = "form-control" })`

5. `Html.ActionLink`:

   - Generates an HTML hyperlink for a specific action method.

   - Example: `@Html.ActionLink("Click here", "About", "Home", null, new { @class = "btn btn-primary" })`

6. `Html.BeginForm`:

   - Generates an HTML `form` element to encapsulate form elements.

   - Example: `@using (Html.BeginForm("SubmitForm", "Home", FormMethod.Post)) { ... }`

7. `Html.ValidationMessageFor`:

   - Generates an HTML `span` element to display validation error messages for a model property.

   - Example: `@Html.ValidationMessageFor(model => model.Email)`

8. `Html.Raw`:

   - Outputs raw HTML content without HTML encoding.

- Example: `@Html.Raw("<strong>Hello</strong>")`

These are just a few examples of the HTML Helper functions available in ASP.NET Core MVC. They significantly simplify the process of generating HTML markup and provide a convenient way to interact with data and form elements within Razor views.

HTML Helper functions are especially useful when working with strongly-typed models as they leverage C# expressions to access properties and perform tasks dynamically. Using HTML Helper functions promotes code consistency, reduces errors, and improves the maintainability of your web applications.

ASP.NET MVC Core state management mechanisms

In the context of ASP.NET Core MVC, state management refers to the process of preserving data or information across multiple requests in a web application. Since HTTP is a stateless protocol, each HTTP request-response cycle is independent, and the server doesn't maintain any information about the previous requests made by the same client.

However, in many web applications, it is essential to maintain data or state across multiple requests to provide a seamless user experience and track user interactions. ASP.NET Core MVC offers several mechanisms for state management:

In ASP.NET Core MVC, you can use several state management mechanisms to preserve data or information across multiple requests in your web application. Here are the state management mechanisms available in ASP.NET Core MVC:

1. **TempData**: `TempData` is a dictionary-like object provided by ASP.NET Core MVC that is used to store data temporarily between two consecutive

requests. It is typically used for passing data from one action method to another or for displaying messages after a redirect. Data stored in `TempData` persists only for one subsequent request and is automatically cleared after that.

2. **View Bag and View Data**: Both `ViewBag` and `ViewData` are mechanisms to pass data from controllers to views. They are used for temporary data transfer between the controller and the corresponding view. While `ViewBag` uses dynamic properties, `ViewData` uses a dictionary. However, like `TempData`, data stored in these objects only persists during the current request.

3. **Query Parameters**: You can pass data between requests using query parameters in the URL. Data is appended to the URL as key-value pairs, and the server can access this data using the `Request.Query` collection.

4. **Route Data**: Route data is used in ASP.NET Core MVC to extract values from the URL route. You can define route patterns that capture segments of the URL, and these segments can be used to determine the controller and action to execute.

5. **Form Data**: Data can be submitted to the server using HTML forms. The server can access this data using the `Request.Form` collection.

6. **Cookies**: Cookies are small pieces of data stored on the client-side and sent with each HTTP request. You can use cookies to store user-specific information, such as preferences or authentication tokens.

7. **Session State**: ASP.NET Core provides session state management to store and retrieve user-specific data across multiple requests. Session state allows you to store data on the server-side and associate it with a specific user. By default, it uses in-memory storage, but you can also use distributed caching or other storage providers for scalability.

Each of these state management mechanisms has its strengths and use cases. The choice of which mechanism to use depends on the specific requirements of your application and the scope of data you need to preserve across requests. For example, if you need to pass data from one action to another during a redirect, you can use `TempData`. If you want to store small amounts of user-specific data on the client-side, you can use cookies. For more complex scenarios, such as user-specific data that should persist across multiple requests, session state might be more appropriate.

Partial Views

Partial views are a feature in ASP.NET Core MVC that allows you to create reusable components that can be shared across multiple views. They are similar to user controls or widgets in other web development frameworks. Partial views enable code reusability, modularity, and help in reducing duplication of code and layout.

A partial view is a self-contained view with its own Razor markup (HTML, C#, etc.), and it can have its own associated model. It does not have a layout, meaning it won't include the standard HTML tags like `<html>`, `<head>`, or `<body>`. Instead, partial views are used to render specific sections of a view or to encapsulate a specific functionality within a larger view.

Partial views are typically used for the following purposes:

1. **Reusable Components**: You can create partial views for common UI components, such as headers, footers, sidebars, navigation menus, etc. These components can then be reused across multiple views, ensuring a consistent user interface.

2. **Complex UI Elements**: In complex views, you can use partial views to break down the UI into smaller, more manageable pieces. This enhances code readability and maintainability.

3. **Widgets**: Partial views can be used to create reusable widgets or custom controls that can be included on different pages of the application.

Here's an example of how to create and use a partial view:

Step 1: Create a Partial View

Create a new Razor partial view file (e.g., `_PartialView.cshtml`) in the `Views/Shared` folder or any other folder you prefer.

```html
<!-- Views/Shared/_PartialView.cshtml -->
<p>Hello from the partial view!</p>
```

Step 2: Use the Partial View in a View

Now, you can use the created partial view within any other view by using the `Partial` method.

```html
<!-- Views/Home/Index.cshtml -->
<h1>Welcome to the Home Page</h1>

<div>
    @await Html.PartialAsync("_PartialView")
```

```

In this example, the `PartialAsync` method is used to include the partial view `_PartialView.cshtml` within the `Index.cshtml` view. The content of the partial view will be rendered at the location where the `PartialAsync` method is called.

Remember that partial views do not have a layout, so they are often used for rendering specific components without duplicating the overall layout structure. They allow developers to promote code reusability and create a more organized and maintainable web application.

---

Action methods and Child Action

**Action Methods**:

Action methods are C# methods within a controller in ASP.NET Core MVC that handle incoming HTTP requests and generate the corresponding HTTP responses. Each action method corresponds to a specific HTTP verb (GET, POST, PUT, DELETE, etc.) and is responsible for processing the request, performing any necessary business logic, and returning the appropriate response, typically in the form of a View or JSON data.

Key Points:

1. Action methods are public methods within a controller class.
```

2. Action methods are decorated with HTTP verb attributes (e.g., `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, `[HttpDelete]`) to specify which type of HTTP request they handle.

3. Action methods can return different types of ActionResult, such as `ViewResult`, `JsonResult`, `RedirectResult`, etc., based on the desired response.

4. Action methods can receive data from the client through parameters, query strings, or form data.

5. Action methods are responsible for invoking the appropriate Model logic and then passing the results to the corresponding View to generate the final HTML output.

Example:

```csharp
public class HomeController : Controller
{
    // GET: /Home/Index
    [HttpGet]
    public IActionResult Index()
    {
        // Business logic, data retrieval, etc.
        var data = GetSomeDataFromDatabase();

        // Return the View with the data
        return View(data);
    }

    // POST: /Home/SaveData
```

```
    [HttpPost]
    public IActionResult SaveData(DataModel model)
    {
        // Validate and save the data to the database
        if (ModelState.IsValid)
        {
            // Save the data
            SaveDataToDatabase(model);
            return RedirectToAction("Index");
        }

        // If the model is not valid, return the same view with validation errors
        return View(model);
    }
}
```

Child Action:

A child action is a specialized type of action method in ASP.NET Core MVC that can be called from within a view to render a partial view. Child actions are typically used to generate specific parts of a view that can be reused across multiple views. They allow developers to encapsulate reusable components or widgets within a controller, and then include those components in different views.

Key Points:

1. Child actions are decorated with the `[ChildActionOnly]` attribute, which means they can only be called from within a view and cannot be directly invoked as an HTTP request.

2. Child actions do not participate in the normal request-response cycle; instead, they are invoked using the `Html.Action` or `Html.RenderAction` helper methods from within a view.

3. Child actions can return a partial view (e.g., `_PartialView.cshtml`) that contains the HTML markup and presentation logic for the reusable component.

4. Child actions can receive data as parameters, allowing them to be dynamic and reusable with different data in different views.

Example:

Controller:

```csharp
public class WidgetController : Controller
{
    [ChildActionOnly]
    public IActionResult RecentPosts(int count)
    {
        // Retrieve the recent posts from the database based on the count parameter
        var recentPosts = GetRecentPostsFromDatabase(count);

        // Return the partial view with the recent posts data
        return PartialView("_RecentPosts", recentPosts);
    }
}
```

```

View:

```html
<!-- Views/Home/Index.cshtml -->
<h1>Welcome to the Home Page</h1>

<!-- Call the child action to render the recent posts widget -->
<div>
    @Html.Action("RecentPosts", "Widget", new { count = 5 })
</div>
```

In this example, the `RecentPosts` child action in the `WidgetController` is used to generate a recent posts widget. The child action is called from within the `Index.cshtml` view using the `Html.Action` helper method, passing the `count` parameter to specify the number of recent posts to display. The child action renders the `_RecentPosts.cshtml` partial view, which contains the HTML markup for the widget and the presentation logic to display the recent posts.

Cookies used for state management

Cookies are small pieces of data that are stored on the client-side (in the user's browser) and are sent with every HTTP request to the same domain. They are commonly used for state management in web applications to store user-specific information or preferences that need to persist across multiple requests or sessions. Cookies are an essential tool for maintaining user sessions, personalizing user experiences, and implementing various features in web applications.
```

Here's how cookies are used for state management in ASP.NET Core MVC:

1. **Setting Cookies**:

   - To set a cookie in ASP.NET Core MVC, you use the `Response.Cookies.Append` method. This method takes the cookie name, value, and other optional parameters such as expiration date, path, and domain.

Example:

```csharp
public IActionResult SetCookie()
{
    // Set a cookie with the name "username" and value "john_doe"
    Response.Cookies.Append("username", "john_doe", new CookieOptions
    {
        Expires = DateTime.Now.AddDays(7), // Cookie will expire after 7 days
        Path = "/", // Cookie is accessible across the entire website
    });

    return View();
}
```

2. **Reading Cookies**:

   - To read a cookie in ASP.NET Core MVC, you use the `Request.Cookies` property, which provides access to the cookies sent with the current HTTP request.

Example:

```csharp
public IActionResult ReadCookie()
{
    // Read the value of the "username" cookie
    var username = Request.Cookies["username"];

    // Use the value as needed in your application
    ViewData["Username"] = username;

    return View();
}
```

3. **Updating Cookies**:
   - To update the value or other properties of a cookie, you can set a new cookie with the same name, which effectively overwrites the existing cookie.

Example:

```csharp
public IActionResult UpdateCookie()
{
    // Update the value of the "username" cookie
    Response.Cookies.Append("username", "jane_smith", new CookieOptions
```

```
    {
        Expires = DateTime.Now.AddDays(7), // Cookie will expire after 7 days
        Path = "/", // Cookie is accessible across the entire website
    });

    return View();
}
```

4. **Deleting Cookies**:
   - To delete a cookie, you set its expiration date to a value in the past. The browser will remove the cookie from its storage automatically.

Example:

```csharp
public IActionResult DeleteCookie()
{
    // Delete the "username" cookie by setting its expiration to the past
    Response.Cookies.Append("username", "", new CookieOptions
    {
        Expires = DateTime.Now.AddDays(-1), // Cookie will be deleted
        Path = "/", // Cookie is accessible across the entire website
    });

    return View();
}
```

```

5. **Cookie Options**:

   - When setting or updating cookies, you can specify various options such as the expiration date, the path on which the cookie is accessible, whether it's secure (HTTPS-only), and more. These options provide control over how the cookie behaves and how long it persists on the user's browser.

Cookies are commonly used to implement various features, such as:

- **User Authentication**: Storing authentication tokens or user identifiers in cookies to maintain user sessions.
- **Remember Me**: Allowing users to stay logged in by persisting their session information in cookies.
- **User Preferences**: Storing user preferences, such as theme settings or language selection, in cookies to personalize the user experience.
- **Shopping Carts**: Keeping track of items in a user's shopping cart by storing cart data in cookies.
- **Tracking User Behavior**: Analyzing user behavior by storing analytics data in cookies.

It's essential to be mindful of the information stored in cookies, as they are stored on the client-side and can be accessed or modified by the user. Avoid storing sensitive or confidential data in cookies and use appropriate security measures when dealing with cookies in your ASP.NET Core MVC application.

---

---

Session

**Session State in ASP.NET Core MVC**:
```

Session state is a server-side state management mechanism in ASP.NET Core MVC that allows you to store and retrieve user-specific data across multiple requests and sessions. Unlike cookies, which are stored on the client-side, session state stores data on the server and associates it with a unique session ID sent to the client via a cookie or URL parameter. This makes session state a more secure option for storing sensitive or confidential information.

Key Points about Session State:

1. **Server-Side Storage**: Session state data is stored on the server, making it more secure than client-side storage like cookies.

2. **Session ID**: The session ID is used to track a user's session and link their subsequent requests to the stored session data on the server.

3. **Expiration**: Session data typically has an expiration time, and after that time elapses, the session data is cleared from the server.

4. **Session Persistence**: By default, ASP.NET Core uses in-memory session storage, which means session data is stored in memory on the server. However, you can use distributed caching or other providers for scalability in web farms or cloud environments.

Now, let's see how to use session state in ASP.NET Core MVC:

1. **Enabling Session State**:

In `Startup.cs`, you need to enable session support in the `ConfigureServices` method:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    // Other service configurations...

    services.AddDistributedMemoryCache(); // For in-memory session storage
    services.AddSession(options =>
    {
        options.IdleTimeout = TimeSpan.FromMinutes(20); // Set session expiration time
        options.Cookie.HttpOnly = true;
        options.Cookie.IsEssential = true; // For compliance with GDPR
    });

    // Other configurations...
}
```

2. **Using Session in Controllers**:

To use session state in a controller, you can inject the `ISession` interface and access the session data through it:

```csharp
public class HomeController : Controller
{
    private readonly ISession _session;
```

```
public HomeController(IHttpContextAccessor httpContextAccessor)
{
    _session = httpContextAccessor.HttpContext.Session;
}

public IActionResult Index()
{
    // Storing data in session
    _session.SetString("username", "john_doe");
    _session.SetInt32("age", 30);

    return View();
}

public IActionResult ReadSession()
{
    // Reading data from session
    var username = _session.GetString("username");
    var age = _session.GetInt32("age");

    // Use the data as needed in your application
    ViewData["Username"] = username;
    ViewData["Age"] = age;

    return View();
}
```

}
```

3. **Accessing Session in Views**:

You can access session data in views using the `Session` property:

```html
<h1>Welcome, @Session["username"]</h1>
<p>Your age is @Session["age"]</p>
```

4. **Clearing Session Data**:

To clear specific session data, you can use the `Remove` method:

```csharp
_session.Remove("username");
```

To clear all session data, you can use the `Clear` method:

```csharp
_session.Clear();
```
```

Session state is useful for storing user-specific data that needs to persist across requests, such as user authentication information, shopping cart contents, or user preferences. However, it's essential to use session state judiciously, as storing large amounts of data in sessions can affect server memory usage and performance. Additionally, consider using distributed caching or external session stores for scalability in production environments.

ADO.NET

ADO.NET (ActiveX Data Objects .NET) is a data access technology in the .NET Framework used to interact with relational databases and other data sources. It provides a set of classes and libraries that enable developers to connect to databases, retrieve data, and perform data manipulation operations. ADO.NET is a key component for building data-driven applications in the .NET ecosystem.

Key Features and Components of ADO.NET:

1. **Data Providers**: ADO.NET includes various data providers that serve as bridges between your application and different database systems. The two primary data providers are:

   - `System.Data.SqlClient`: For connecting to Microsoft SQL Server databases.

   - `System.Data.OleDb`: For connecting to any database using OLE DB (e.g., Microsoft Access).

2. **Connection and Command Objects**: ADO.NET provides `SqlConnection` for establishing connections to the database and `SqlCommand` for executing SQL queries or stored procedures.

3. **DataReader**: ADO.NET offers a `DataReader` that provides a fast, forward-only, read-only data stream from the database. It is suitable for scenarios where you need to read large amounts of data efficiently.

4. **DataSets and DataTables**: ADO.NET introduces `DataSet` and `DataTable` classes, which are in-memory representations of the data retrieved from the database. They allow you to work with data in a disconnected manner, making it possible to manipulate data offline before committing changes back to the database.

5. **DataAdapter**: The `DataAdapter` acts as a bridge between `DataSet` and the database, facilitating the populating of `DataSet` with data from the database and updating the database with changes made to the `DataSet`.

6. **Data Binding**: ADO.NET supports data binding, allowing you to bind data from `DataSet` or `DataTable` to UI controls like DataGrid, GridView, or ComboBox in a straightforward manner.

7. **Transactions**: ADO.NET supports transactions, enabling you to perform multiple database operations as a single, atomic unit. This ensures that either all the changes are applied to the database, or none of them are, in case of a failure.

8. **Disconnected Data Architecture**: ADO.NET follows a disconnected data architecture, where data is fetched from the database and stored in the `DataSet`, and then the database connection is closed. This allows for efficient use of resources and reduces the time the database connection remains open.

ADO.NET offers flexibility and performance for data access tasks, making it a popular choice for developers building .NET applications that need to interact with databases. It provides different approaches to data access, allowing you to choose the most appropriate method based on the specific requirements of your application. Whether you need to perform simple data queries or work with complex data structures, ADO.NET offers the tools and classes to meet your data access needs within the .NET Framework.

CRUD Operations

```
namespace DAL.Connected;
using BOL;
using MySql.Data.MySqlClient;
//using inbuilt, external Object Models



public class DBManager{


   public static string conString=@"server=localhost;port=3306;user=root;
password=password;database=transflower";
   public  static List<Department> GetAllDepartments(){
         List<Department> allDepartments=new List<Department>();
         //database connectivity code
         //Connected Data Access Mode
         //MySqlConnection  : establishing connection
         //MySqlCommand      : query execution
         //MySqlDataReader   : result of query to be catured after processing
query
         MySqlConnection con=new MySqlConnection();
         con.ConnectionString=conString;
         try{
            con.Open();
            //fire query to database
            MySqlCommand cmd=new MySqlCommand();
            cmd.Connection=con;
            string query="SELECT * FROM departments";
```

```
            cmd.CommandText=query;
            MySqlDataReader reader=cmd.ExecuteReader();
            while(reader.Read()){
                int id = int.Parse(reader["id"].ToString());
                string name = reader["name"].ToString();
                string location = reader["location"].ToString();

                Department dept=new Department{
                                        Id = id,
                                        Name = name,
                                        Location = location
                };
                allDepartments.Add(dept);
            }
        }
        catch(Exception ee){
            Console.WriteLine(ee.Message);
        }
        finally{
                con.Close();
        }

        return allDepartments;
}

public static Department GetDeparmentDetails(int id){
    Department dept = null;
```

```
MySqlConnection con = new MySqlConnection();
con.ConnectionString = conString;
try
{
   string query = "SELECT * FROM departments WHERE id=" + id;
   con.Open();
   MySqlCommand command = new MySqlCommand(query, con);
   MySqlDataReader reader = command.ExecuteReader();
   if (reader.Read())
   {
      id = int.Parse(reader["id"].ToString());
      string name = reader["name"].ToString();
      string location = reader["location"].ToString();
      dept = new Department
      {
         Id = id,
         Name = name,
         Location = location
      };
   }

}
catch (Exception e)
{
   throw e;
}
finally
```

```
    {
        con.Close();
    }
    return dept;
}

 public static bool Insert(Department dept){
    bool status=false;
    string query = "INSERT INTO departments(name,location)" +
                    "VALUES('" + dept.Name + "','" + dept.Location + "')";

    MySqlConnection con = new MySqlConnection();
    con.ConnectionString = conString;
    try{
        con.Open();
        MySqlCommand command = new MySqlCommand(query, con);
        command.ExecuteNonQuery();  //DML
        status = true;
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        con.Close();
    }
```

```
        return status;
    }

    public static bool Update(Department dept)
    {
        bool status = false;
        MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;
        try
        {
            string query = "UPDATE departments SET location='" + dept.Location +
"', name='" + dept.Name + "' WHERE id=" + dept.Id;
            MySqlCommand command = new MySqlCommand(query, con);
            con.Open();
            command.ExecuteNonQuery();
            status = true;
        }
        catch (Exception e)
        {
            throw e;
        }
        finally
        {
            con.Close();
        }
        return status;
    }
```

```
    public static bool Delete(int id){
        bool status=false;
        MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;
        try
        {
            string query = "DELETE FROM departments WHERE id=" + id;
            MySqlCommand command = new MySqlCommand(query, con);
            con.Open();
            command.ExecuteNonQuery();
        }
        catch (Exception e)
        {
            throw e;
        }
        finally
        {
            con.Close();
        }
      return status;
    }



//Employee Operations CRUD
    public static bool DoesEmployeeExists(int id)
```

```
    {
        MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;
        bool status = false;
        try
        {
            string query = "SELECT EXISTS (SELECT * FROM employees where id="
+ id + ")";
            con.Open();
            MySqlCommand cmd = new MySqlCommand(query, con);
            MySqlDataReader reader = cmd.ExecuteReader();
            reader.Read();

            if ((Int64)reader[0] == 1)
            {
                status = true;
            }

            reader.Close();
        }
        catch (Exception e)
        {
            throw e;
        }
        finally
        {
            con.Close();
```

```
        }
        return status;
    }
    public static List<Employee> GetAllEmployees()       //get all rows of
employee table
    {

        List<Employee> employees = new List<Employee>();
        MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;
        try
        {
            string query = " SELECT * FROM employees";
            con.Open();
            MySqlCommand cmd = new MySqlCommand(query, con);
            MySqlDataReader reader = cmd.ExecuteReader();
            while (reader.Read())
            {
                int id = int.Parse(reader["id"].ToString());
                string firstname = reader["firstname"].ToString();
                string lastname = reader["lastname"].ToString();
                string email = reader["email"].ToString();
                string address = reader["address"].ToString();
                int deptid = int.Parse(reader["deptid"].ToString());
                int managerid = int.Parse(reader["managerid"].ToString());

                Employee emp = new Employee
```

```
            {
                Id = id,
                FirstName = firstname,
                LastName = lastname,
                Email = email,
                Address = address,
                DeptId = deptid,
                ManagerId = managerid,

            };
            employees.Add(emp);
        }
        reader.Close();
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        con.Close();
    }
    return employees;
}
public static Employee GetById(int id)                //show employee data by id
{
    Employee foundEmployee = null;
```

```
MySqlConnection con = new MySqlConnection();
con.ConnectionString = conString;
try
{
    string query = "SELECT * FROM employees WHERE id =" + id;
    con.Open();
    MySqlCommand cmd = new MySqlCommand(query, con);
    MySqlDataReader reader = cmd.ExecuteReader();
    if (reader.Read())
    {
        id = int.Parse(reader["id"].ToString());
        string firstName = reader["firstName"].ToString();
        string lastName = reader["lastName"].ToString();
        string email = reader["email"].ToString();
        string address = reader["address"].ToString();
        int deptid = int.Parse(reader["deptid"].ToString());
        int managerid = int.Parse(reader["managerid"].ToString());



        foundEmployee = new Employee
        {
            Id = id,
            FirstName = firstName,
            LastName = lastName,
            Email = email,
```

```
                Address = address,
                DeptId = deptid,
                ManagerId = managerid,

            };
        }
        reader.Close();
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        con.Close();
    }
    return foundEmployee;
}
public static List<Role> GetRolesOfEmployee(int empId){
    List<Role> roles=new List<Role>();
    //get all roles belong to empid
    //*****************************
    MySqlConnection con = new MySqlConnection();
    con.ConnectionString = conString;
    try
    {
        //Query to return roles belong to employee id
```

```
        string query = "select rolename from roles"+
                       "where roleid IN("+
                       "select roleid from emp_roles"+
                       "where empid="+empId+");";
        Console.WriteLine(query);

        con.Open();
        MySqlCommand cmd = new MySqlCommand(query, con);
        MySqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            Role theRole=new Role();
            theRole.Id=int.Parse(reader["roleid"].ToString());
            theRole.Name=reader["rolename"].ToString();
            roles.Add(theRole);
        }
    }
    catch(Exception ee){

    }
    finally{
        con.Close();
    }
    //*******************************
    return roles;
}
```

```
   public static bool Insert(Employee emp)        //insertion of employee table
data
   {
      bool status = false;
      MySqlConnection con = new MySqlConnection();
      con.ConnectionString = conString;
      try
      {
         string query = "Insert into employees
(firstName,lastName,email,address,password, deptid,managerid) Values( '" +
emp.FirstName + "','"
         + emp.LastName + "','" + emp.Email + "','" + emp.Address + "',  '" +
emp.Password + "' ,'" + emp.DeptId + "','" + emp.ManagerId + "')";
         MySqlCommand cmd = new MySqlCommand(query, con);
         con.Open();
         cmd.ExecuteNonQuery();

         status = true;
      }
      catch (Exception e)
      {
         throw e;
      }
      finally
      {
         con.Close();
      }
      return status;
```

```
    }
    public static bool SetPassword(string email, string password)
    {
        bool status = false;
        //set password for existing employee whoes email id matches
        //call ado.net code to update password field of employee
        return status;
    }
    public static bool Update(Employee emp)          //updating employee table
data
    {

        MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;
        bool status = false;

        try
        {
            string query = "Update employees SET firstName ='" + emp.FirstName +
"'," + "lastName ='" + emp.LastName + "',"
                    + "email='" + emp.Email + "'," + "address='" + emp.Address + "'," +
"managerid=" + emp.ManagerId + "," +
                    "deptid=" + emp.DeptId + " WHERE id =" + emp.Id;

            MySqlCommand cmd = new MySqlCommand(query, con);
            con.Open();
            cmd.ExecuteNonQuery();
```

```
        }
        catch (Exception e)
        {
            throw e;
        }
        finally
        {
            con.Close();
        }
        return status;
    }
    public static void Delete(Employee emp)                //delete employee table
data
    {
        MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;

        try
        {
            string query = " DELETE FROM employees WHERE id =" + emp.Id;
            MySqlCommand cmd = new MySqlCommand(query, con);
            con.Open();
            cmd.ExecuteNonQuery();
        }
        catch (Exception ee)
        {
            throw ee;
```

```
        }
        finally
        {
            con.Close();
        }

    }
    public static List<Employee> GetEmployeesByDepartment(int deptid)   //get all rows of employee table
    {
        List<Employee> employees = new List<Employee>();
        MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;
        try
        {
            string query = " SELECT * FROM employees where deptid=" + deptid;
            con.Open();
            MySqlCommand cmd = new MySqlCommand(query, con);
            MySqlDataReader reader = cmd.ExecuteReader();
            while (reader.Read())
            {
                int id = int.Parse(reader["id"].ToString());
                string firstname = reader["firstname"].ToString();
                string lastname = reader["lastname"].ToString();
                string email = reader["email"].ToString();
                string address = reader["address"].ToString();
                int dptid = int.Parse(reader["deptid"].ToString());
```

```
            int managerid = int.Parse(reader["managerid"].ToString());

            Employee emp = new Employee
            {
                Id = id,
                FirstName = firstname,
                LastName = lastname,
                Email = email,
                Address = address,
                DeptId = dptid,
                ManagerId = managerid,

            };
            employees.Add(emp);
        }
        reader.Close();
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        con.Close();
    }
    return employees;
}
```

```
public static bool Transfer(int empId, int deptId)
{
    bool status = false;

    // create connection objet
    // set connection string to connection

    MySqlConnection con = new MySqlConnection();
    con.ConnectionString = conString;

    // define query to update existing empoyees department id
    string query = "Update employees SET deptid=" + deptId + " WHERE id=" +
empId;
    try
    {
        con.Open();
        // create command object
        //
        // associate connection and query
        MySqlCommand cmd = new MySqlCommand(query, con);
        // execute command
        cmd.ExecuteNonQuery();
        status = true;
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
```

```
    }
    // set status true on success
    return status;
}

//Roles crude operations
public static List<Role> GetAllRolesOfEmployee(int empId){
    List<Role> roles = new List<Role>();
    MySqlConnection con = new MySqlConnection();
    con.ConnectionString = conString;
    try
    {
        string query = "select * from roles " +
                       "where roleid IN ( "+
                       "select roleid from emp_roles "+
                       "where empid="+ empId+ ")";
        Console.WriteLine(query);
        con.Open();
        MySqlCommand cmd = new MySqlCommand(query, con);
        MySqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            int roleId = int.Parse(reader["roleid"].ToString());
            string roleName = reader["rolename"].ToString();
            Role theRole= new Role
            {
                Id = roleId,
```

```
                Name= roleName
            };
            roles.Add(theRole);
        }
            reader.Close();
    }
    catch(Exception e)
    {
        throw e;
    }
    finally
    {
        con.Close();
    }
    return roles;
}
public static List<Role> GetAllRoles()
{
    List<Role> roles = new List<Role>();
    MySqlConnection con = new MySqlConnection();
    con.ConnectionString = conString;
    try
    {
        string query = "SELECT * FROM roles order by roleid asc";
        con.Open();
        MySqlCommand cmd = new MySqlCommand(query, con);
        MySqlDataReader reader = cmd.ExecuteReader();
```

```
        while (reader.Read())
        {
            int roleId = int.Parse(reader["roleid"].ToString());
            string roleName = reader["rolename"].ToString();

            Role role1= new Role
            {
                Id = roleId,
                Name= roleName
            };
            roles.Add(role1);
        }
            reader.Close();
    }
    catch(Exception e)
    {
        throw e;
    }
    finally
    {
        con.Close();
    }
    return roles;
}
public static List<Employee> GetAllEmployeesByRole(string rolename)
{
     List<Employee> employees = new List<Employee>();
```

```
MySqlConnection con = new MySqlConnection();
con.ConnectionString = conString;
try
{
   string query = " select * from employees"+
               " where id IN("+
                     "select  empid from emp_roles "+
                     "where  roleid=(" +
                                 "select roleid from roles "+
                                 "where rolename='"+rolename+"'))";
   con.Open();
   MySqlCommand cmd = new MySqlCommand(query, con);
   MySqlDataReader reader = cmd.ExecuteReader();
   while (reader.Read())
   {
      int id = int.Parse(reader["id"].ToString());
      string firstname = reader["firstname"].ToString();
      string lastname = reader["lastname"].ToString();
      string email = reader["email"].ToString();
      string address = reader["address"].ToString();
      int deptid = int.Parse(reader["deptid"].ToString());
      int managerid = int.Parse(reader["managerid"].ToString());

      Employee emp = new Employee
      {
         Id = id,
         FirstName = firstname,
```

```
                LastName = lastname,
                Email = email,
                Address = address,
                DeptId = deptid,
                ManagerId = managerid,

            };
            employees.Add(emp);
        }
        reader.Close();
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        con.Close();
    }
    return employees;
}

//Add role for existing employee
public static bool AssignRole(int empid,int roleid)
{
    bool status=false;
     MySqlConnection con = new MySqlConnection();
```

```
        con.ConnectionString = conString;
        try
        {
           string query = "Insert into emp_roles(empid,roleid)
values("+empid+","+roleid+")";
            con.Open();
            MySqlCommand cmd=new MySqlCommand(query,con);
            cmd.ExecuteNonQuery();
            status=true;
        }
        catch (Exception e)
        {
           throw e;
        }
        finally
        {
           con.Close();
        }
        return status;
    }

    //Remove role of existing employee from RoleMapping
    public static bool UnAssignRole(int empid,int roleid)
    {
        bool status=false;
         MySqlConnection con = new MySqlConnection();
        con.ConnectionString = conString;
```

```
        try
        {
            //*****************************"
            //Query for removing roles assigned to emp id
            string query = "delete from emp_roles WHERE empid=" + empid+
                           " AND roleid="+roleid;
             con.Open();
             MySqlCommand cmd=new MySqlCommand(query,con);
             cmd.ExecuteNonQuery();
             status=true;
        }
        catch (Exception e)
        {
            throw e;
        }
        finally
        {
            con.Close();
        }
        return status;
    }}



//

            //DisConnected Data Access Mode
```

//MySqlConnection : establishing connection

//MySqlCommand : query execution

//MySqlDataApater

//DataSet

//DataTable

//DataRow

//DataColumn

//DataRealtion

ADO.NET most frequent question

In an interview or technical examination focused on ADO.NET, the questions may cover various aspects of data access, data manipulation, and data management using ADO.NET. Here are some of the most probable and important questions you may encounter:

1. What is ADO.NET, and how does it differ from classic ADO (ActiveX Data Objects)?

2. Explain the basic architecture of ADO.NET and its key components.

3. What are the different data providers available in ADO.NET? How do they differ from each other?

4. Describe the role and usage of `SqlConnection`, `SqlCommand`, `SqlDataReader`, and `DataSet` in ADO.NET.

5. What are the differences between a connected and disconnected data architecture in ADO.NET?

6. How do you handle exceptions and errors in ADO.NET applications?

7. What are the advantages and disadvantages of using a `DataReader` over a `DataAdapter`?

8. How do you use a `DataReader` to read data from a database, and what is the significance of its forward-only nature?

9. Explain how to perform CRUD (Create, Read, Update, Delete) operations using a `DataSet` and `DataAdapter`.

10. Describe the role of `DataTable`, `DataRow`, and `DataColumn` in a `DataSet`.

11. What is the purpose of a `DataView`, and how is it used for sorting and filtering data in a `DataTable`?

12. What is optimistic concurrency, and how can you implement it in ADO.NET?

13. Explain the use of transactions in ADO.NET, and how can you ensure data consistency using transactions?

14. How do you handle large datasets in ADO.NET to optimize performance and memory usage?

15. What are parameterized queries, and why are they essential for secure data access in ADO.NET?

16. What is the significance of using stored procedures in ADO.NET applications?

17. Describe the process of handling binary data (BLOBs) in ADO.NET.

18. How do you work with XML data in ADO.NET, and what is the role of `XmlReader` and `XmlWriter`?

19. Explain the difference between synchronous and asynchronous data access, and how do you implement asynchronous commands in ADO.NET?

20. What is connection pooling, and how does ADO.NET utilize it to optimize database connections?

These questions cover a broad range of topics related to ADO.NET, from basic concepts and components to more advanced data access techniques and best practices. Preparing for these questions will help you demonstrate a comprehensive understanding of ADO.NET and its usage in building data-driven applications. Additionally, it is advisable to practice hands-on coding exercises and work with real-world scenarios to reinforce your knowledge and skills.

Sloutions to the above questions

Sure, here's a list of solutions to the questions listed sequentially:

1. ADO.NET is a data access technology in .NET used to interact with databases, while classic ADO (ActiveX Data Objects) is an older data access technology in pre-.NET Microsoft platforms. ADO.NET is more robust, efficient, and suitable for modern data access needs.

2. The basic architecture of ADO.NET includes data providers (e.g., `SqlConnection`, `OleDbConnection`), commands (e.g., `SqlCommand`, `OleDbCommand`), data readers (e.g., `SqlDataReader`, `OleDbDataReader`), datasets (`DataSet`), and data adapters (e.g., `SqlDataAdapter`, `OleDbDataAdapter`).

3. The different data providers available in ADO.NET include `SqlClient` for SQL Server, `OleDb` for OLE DB data sources, `OracleClient` for Oracle databases, and `ODBC` for ODBC data sources.

4. `SqlConnection` is used to establish a connection to a database, `SqlCommand` represents an SQL command to be executed, `SqlDataReader` is used to read data from the database, and `DataSet` is an in-memory representation of data.

5. ADO.NET supports both connected and disconnected data architectures. Connected data access maintains an open connection to the database throughout the operation, while disconnected data access fetches data into a `DataSet` and then disconnects from the database, allowing data manipulation without an active connection.

6. Exception handling in ADO.NET can be done using `try-catch` blocks to catch and handle exceptions that might occur during data access operations.

7. Advantages of `DataReader` over `DataAdapter` include efficiency for reading large result sets and forward-only, read-only access. Disadvantages include the inability to modify data and lack of random access.

8. To use a `DataReader`, you execute a command (`ExecuteReader`) to obtain a `DataReader` instance. You can then use its `Read()` method to iterate through the result set and retrieve data.

9. CRUD operations can be performed using `DataSets` and `DataAdapters`. For example, to update data, you modify the data in the `DataSet`, and then call `Update` on the `DataAdapter` to apply changes to the database.

10. `DataTable` holds data in a `DataSet`, while `DataRow` represents a single row of data in a `DataTable`. `DataColumn` represents a column in a `DataTable`.

11. `DataView` allows you to sort and filter data in a `DataTable` based on specified criteria.

12. Optimistic concurrency is a technique where the data is updated without locking the record in advance. You can implement it by using timestamp columns or comparing values before updating.

13. Transactions are used to group multiple database operations into a single unit of work, ensuring that either all the operations succeed or none of them do.

14. For large datasets, you can use paging techniques to fetch data in smaller chunks, reducing memory consumption and improving performance.

15. Parameterized queries use placeholders for dynamic values in SQL commands, reducing the risk of SQL injection attacks.

16. Stored procedures are precompiled SQL statements that are stored on the database server and can be called from ADO.NET applications.

17. BLOBs (Binary Large Objects) can be handled using `byte[]` arrays to store binary data in databases.

18. XML data can be handled using `XmlReader` and `XmlWriter` classes to read and write XML data.

19. Asynchronous data access in ADO.NET can be achieved using `async/await` for methods that support asynchronous execution, such as `OpenAsync`, `ExecuteReaderAsync`, etc.

20. Connection pooling is a technique in ADO.NET that maintains a pool of open database connections to improve performance by reusing connections instead of creating new ones for each request.

Asysnchronous Command Execution

Asynchronous command execution in ADO.NET allows you to execute database commands (such as queries or stored procedures) asynchronously, meaning that the execution of the command does not block the main thread of the application. This is particularly useful for applications that require responsiveness and need to perform multiple database operations concurrently without waiting for each operation to complete before moving on to the next one.

In traditional synchronous execution, when a command is executed, the application waits for the command to complete and return the result before proceeding to the next statement. With asynchronous execution, the command

is executed in the background, and the application can continue doing other tasks while waiting for the command to complete.

Key Points about Asynchronous Command Execution in ADO.NET:

1. **Asynchronous Methods**: Asynchronous command execution is achieved through the use of asynchronous methods, which have the "async" keyword in their method signature. These methods return a `Task` or `Task<T>` representing the asynchronous operation.

2. **Async and Await**: Asynchronous methods use the "await" keyword to asynchronously wait for the operation to complete without blocking the main thread. This allows the application to continue executing other tasks while the asynchronous operation is in progress.

3. **Concurrency**: Asynchronous command execution allows multiple commands to be executed concurrently, improving the overall performance of the application.

4. **Non-Blocking**: Asynchronous execution ensures that the main thread is not blocked while waiting for the database operation to complete. This enhances the responsiveness of the application, especially in scenarios like web applications handling multiple user requests simultaneously.

Example - Asynchronous Command Execution in ADO.NET:

Let's see an example of how to perform asynchronous command execution in ADO.NET using C#. We'll use the `SqlConnection` and `SqlCommand` classes to execute a simple query asynchronously.

```csharp
```

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Threading.Tasks;

public class AsynchronousCommandExecutionExample
{
    public static async Task GetDataAsync()
    {
        string connectionString = "Data Source=YourServer;Initial
Catalog=YourDatabase;Integrated Security=True;";
        string sqlQuery = "SELECT Id, Name, Age FROM Employees";

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            await connection.OpenAsync();

            using (SqlCommand command = new SqlCommand(sqlQuery, connection))
            {
                using (SqlDataReader reader = await command.ExecuteReaderAsync())
                {
                    while (await reader.ReadAsync())
                    {
                        int id = reader.GetInt32(0);
                        string name = reader.GetString(1);
                        int age = reader.GetInt32(2);
```

```
                Console.WriteLine($"ID: {id}, Name: {name}, Age: {age}");
            }
        }
    }
  }
}
```

In this example, we created an asynchronous method `GetDataAsync()` that fetches data from the "Employees" table in the database using asynchronous command execution. The `await` keyword is used to asynchronously wait for the `OpenAsync()` and `ExecuteReaderAsync()` methods to complete.

Key Points to Remember:

- Asynchronous command execution is useful for applications that need to perform multiple database operations concurrently while keeping the application responsive.
- Asynchronous methods use the "async" keyword and return a `Task` or `Task<T>`.
- The "await" keyword is used to asynchronously wait for the result of an asynchronous operation without blocking the main thread.
- Asynchronous execution can significantly improve the performance of applications, especially in scenarios with high concurrency and long-running operations, such as web applications and services.

It's essential to use asynchronous command execution judiciously and ensure that the application design and architecture support concurrent execution of

asynchronous tasks. Asynchronous programming can be beneficial when used correctly but can also introduce complexities if not managed properly.

Asysnchronous connections

In ASP.NET Core MVC, asynchronous connections refer to the ability to use asynchronous programming techniques to perform operations that involve external resources, such as databases, web services, or file systems, without blocking the main thread of the application. Asynchronous connections are especially useful in scenarios where the application needs to handle multiple concurrent requests efficiently and maintain responsiveness.

When performing operations that may take some time to complete, such as reading data from a database or making an HTTP request to an external API, the traditional synchronous approach can lead to thread blocking, which can negatively impact the application's performance and scalability. Asynchronous connections, on the other hand, enable the application to initiate an operation and continue processing other requests or tasks while waiting for the operation to complete.

Key Points about Asynchronous Connections in ASP.NET Core MVC:

1. **Async/Await Pattern**: Asynchronous connections in ASP.NET Core MVC utilize the async/await pattern, which allows developers to write asynchronous code in a more readable and sequential manner. Asynchronous methods are marked with the "async" keyword, and the "await" keyword is used to await the result of an asynchronous operation.

2. **Task-Based Asynchronous Programming (TAP)**: Asynchronous connections are built on the Task-Based Asynchronous Programming model, where asynchronous methods return a `Task` or `Task<T>` that represents the

ongoing operation. The main thread can continue its work while the asynchronous operation is in progress.

3. **Avoiding Thread Blocking**: Asynchronous connections prevent thread blocking, which means the application can handle multiple requests concurrently, making better use of available resources and improving the overall responsiveness.

4. **I/O Bound Operations**: Asynchronous connections are particularly suitable for I/O bound operations, where the application is waiting for an external resource, such as a database or web service, to return data.

Example - Asynchronous Connection in ASP.NET Core MVC:

Let's see an example of an asynchronous action method in ASP.NET Core MVC that demonstrates how asynchronous connections can be used for database access:

```csharp
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using YourProject.Models;

public class EmployeeController : Controller
{
    private readonly YourDbContext _dbContext;
```

```
    public EmployeeController(YourDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    // Asynchronous action method to retrieve employees from the database
    public async Task<IActionResult> Index()
    {
        List<Employee> employees = await _dbContext.Employees.ToListAsync();
        return View(employees);
    }
}
```

In this example, the `Index` action method in the `EmployeeController` is marked as asynchronous using the "async" keyword. Inside the method, we use `await` to asynchronously wait for the database operation `ToListAsync()` to complete. This allows the main thread to continue processing other requests or tasks while the database query is being executed asynchronously.

Key Points to Remember:

- Asynchronous connections in ASP.NET Core MVC are achieved using the async/await pattern and the Task-Based Asynchronous Programming model.
- Asynchronous connections are particularly suitable for I/O bound operations, such as database access, web service calls, or file I/O, where the application spends a significant time waiting for an external resource.
- Asynchronous connections improve application performance, scalability, and responsiveness, especially when handling multiple concurrent requests.

- Asynchronous programming should be used judiciously and is most beneficial when used for long-running I/O bound operations.

When using asynchronous connections in ASP.NET Core MVC, it's important to ensure that the underlying components, such as the database provider or external APIs, support asynchronous operations to fully benefit from the performance gains. Asynchronous connections can significantly improve the overall user experience in web applications by ensuring responsiveness, especially under heavy load or during long-running operations.
