Stored Procedures

Button_1 Click:

```
SqlCommand sqlcommand = dBCon.UDI(Stored Procedure :
        "dbo.Employee_Insert");
sqlCommand.Parameters.Add("@Id", sqlDbType.Int).Value
                = int.Parse(TID.Text);
SqlCommand.Parameter.Add("@Name", SqlDbType.VarChar).
        Value = TName.Text;
SqlCommand.Parameter.Add("@Email", SqlDbType.VarChar).Value
                = TEmail.Text;
if(sqlCommand.ExecuteNonQuery()>0)
{
  LInfo.Text = "Data Inserted";
}
else
{
  LInfo.Text = "Data Not Inserted";
}
```

Button 5 click:-

```
SqlCommand sqlcommand = dBCon.UDI(storedprocedure : "dbo.Employee
        _Search");
SqlDataReader sqldatareader = sqlcommand.ExecuteReader();
  if (sqldatareader.HasRows)
  {
    while(sqlDatareader.Read())
    {
      TID.Text = (sqlDatareader["ID"].ToString());
      TName.Text = (sqlDatareader["Name"].ToString());
      TEmail.Text = (sqldatareader["Email"].ToString());
      LInfo.Text = "Data Found";
    }
  else
    {
      LInfo.Text = "No rows found";
    }
```

DBCon (Stored Procedures)

```
Public DBcon{
  SQLConnection sqlconnection = null;
  Public Void ConnectiontoDataBase()
  {
    String connection = "  _  _____  ";
    sqlConnection = new SqlConnection(connection);
    sqlConnection.Open()
    sqlConnection.Close;
  }
  public void ConnectionClose()
  {
    sqlConnection.Close();
  }
  Public SqlCommand UDI(string storedprocedure)
  {
    Connection to DataBase();
    SqlCommand sqlcommand = new SqlCommand
                (storedprocedure, sqlconnection)
    sqlCommand.CommandType = CommandType.StoredProcedure;
    return sqlCommand;
  }
```

LINQ TO XML

Employee.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
  <Employee Id="1">
    <FirstName> Aqeel </First Name>
    <LastName> Ijaz </Last Name>
      //
      //
      //
  </Employee>
</Employees>
```

EmployeeForm.aspx.cs

```
public Partial Class Employee Form
{
  XDocument xDocument;
  public void PageLoad()
  {
  XDocument = XDocument.Load (Server.MapPath ("Employe
                                                .xml"))

  Load GridView();
  }
```

```
public btnInsert
{
XElement xelement= XDocument.Descendants("Employee").
FirstorDefault(x => x.Attribute("Id").Value.Equals(TID.Text));
if (employee==null)
{
  XDocument.Element("Employees").Add(
new XElement("Employee",
   new XAttribute("Id", txtID.Text),
   new XElement("Name", txtName.Text),
   new XElement("Education", txtedu.Text),
   new XElement("Department", txtdpt.Text),
   new XElement("Salary", txtsalary.Text),
   )
  );
XDocument.Server.SavePath("Employee.xml(")");
LBLinfo.Text = "Data Added";
Load Gridview();
}
else
{
  LBLinfo.Text = "Data Already exists";
}
```

```
Public btn Update
{
    XElement employee = XDocument.Descendants("Employee")
    .FirstOrDefault(x => x.Attribute("Id").Value.Equals(txtID.Text));
    XElement employee.Element("FirstName").Value = txtFname.Text;

    //XDocument.Save(Server.MapPath("Employee.xml"));
    LoadGridView();
    LBLinfo.Text = "Data Updated";

Public btn Delete
    if (employee != null)
    {
        employee.Remove();
    }

Btn Search
{   if (employee != null)
    {
        txtFname.Text = employee.Element("FirstName").Value;
    }
```

```
LoadGridView()
    DataSet dataset = new DataSet();
    dataSet.ReadXml(Server.MapPath("Employee.xml"));
    if(dataset.Table.Count>0)
    {
      dataGridView.DataSource = dataset;
      dataGridView.DataBind();
    }
```

LINQ TO JSON

```
Partial class ProductForm
{
  List<Dictionary<string, object>> rows;
JavaScriptSerializer serializer = new JavaScriptSerialize();
  public PageLoad()
  {
    StreamReader reader = new StreamReader(Server.MapPath("product
          .json"));
    string json = reader.ReadtoEnd();
    rows = serializer.Deserialize&List<Dictionary<string,Object
        >>>(json);
    reader.Close();
```

```
if (!IsPostBack)
{
    txtMain.Text = serializer.Serialize(rows);
}

gridview.DataSource = DisplayJsonData();
gridview.DataBind();
}

}

public void btnInsert_Click()
{

Dictionary<string, object> row = new Dictionary<string, object>();
row.Add("P_ID", txtID.Text);
row.Add("P_Title", txtTitle.Text);
row.Add("P_Expiry", txtExp.Text);
row.Add("P_Manufacturer", txtmanuf.Text);
row.Add("P_Price", txtprice.Text);
row.Add("P_stock", txtstock.Text);
row.Add(rows);
txtMain.Text = serializer.Serialize(rows);
}
```

```
Public void btn_Update ( )
{ Dictionary<string,Object) row = new Dictionary <string, Object>();
  rows.Add("P_ID"} = txtID.Text);
  row = rows.(Single orDefault(x => row.All (x.contains));
  row["P_title"] = txttille.Text;
  row ["P_Expiry"] = txtExp.Text;
  row["P_Manufacturer"] =txt[Manuf. Text;
  row ["P_ Stock"] = txtStock.Text;
  row ["P_ Price"] = txtPrice.Text;
  txtMain.Text = serializer.Serialize(rows);
}

Public void btn_Delete()
  {

   Dictionary<string, object> row = new Dictionary <string, object>()
    row.Add("P_ID", txtID.Text);
   row = rows. Single orDefault (x => row.All (x.Contains));
   ~~txtTitle~~rows. Remove (row);
   txtMain.Text = $serializer. Serialize (rows);
  }
```

```
Public void btn_Search
{
Dictionary <string, object> row = new Dictionary <string, object>
        ();
  row.Add ("P_ID", txtID.Text);
 row = rows.SingleOrDefault (x => row.All(x.contains));
 txtTitle = Prow["P_Title"].ToString ();
 txtExpiry = row["P_Expiry"].ToString ();
 txtMauf = row["P_Manufacturer"].ToString ();
 txtPrice = row["P_Price"].ToString ();
 txtStock = row["P_Stock"].ToString ();
 }

Public DataTable DisplayJSONData ()
 {

 StreamReader sr = new StreamReader (Server.MapPath("~/Product.js
 String json = sr.ReadToEnd ();
 sr.Close ();
Var table = JsonConvert.DeserializeObject<DataTable> (json);
 return table;
}
```

```
Public void SaveToFile()
{
 System.IO.StreamWriter myFile = new System.IO.Stream
    Writer (Server.MapPath("Product.json");
   if (!txtMain.Text.Equals(" "))
   {
     myFile.WriteLine(txtmain.Text);
   }
   myFile.Close();
   }
  }
}
```

SHOPPING CART

Product DAL.cs,

```
Public class ProductDAL
{

  DBCon dbcon = new DBCon();
  Public DataTable GetAllProducts()
  {

    String query = "Select * from SCproducts";
    return dbcon.Search(query);
  }

  Public DataTable SearchProductDA(Product product)
  {

    String query = "Selet * from SCproducts where
     Product_ID = " + product_id + "";
    return dbcon.Search(query);
  }
}
```

Product BLL.cs

```
public class productBLL
{
    ProductDAL productdal = new ProductDAL();
    public DataTable GetAllProduct()
    {
        return productdal.GetAllProductsDAL();
    }
    public DataTable SearchProduct()
    {
        return productdal.SearchProductDAL(product);
    }
}
```

Default.aspx.cs

```
Public Page_Load()
{
if(!Page.IsPostBack)
{
    ProductBLL productbll = new ProductBLL();
    DataTable datatable = productbll.GetAllproduct();
    DatalistProduct.DataSource = datatable;
    DataListProduct.DataBind();
}
```

```
Public btnAddtoCart()
{
  int id = int.Parse(e.CommandArgument.ToString());
  Product product = new Product();
  product.Id = ID;
  ProductBLL productBLL = new ProductBLL();
  DataTable datatable = productBLL.SearchProduct(product);
  product.Name = datatable.Rows[0]["Name"].ToString();
  product.Image = datatables.Rows[0]["Image"].ToString();
  product.Description = dataTable.Rows[0]["Description"].ToString();
  product.Price = float.Parse(dataTable.Row[0]["Price"].ToString());

  Cart cart = null;
 if [Session["cart"] != null]
  {
    cart (Cart)Session["cart"];
  }
    Cart cart = new Cart();
 if (new Cart.AddtoCart(Product, 1))
  {
     Session["cart"] = new Cart;
     Response.Redirect(Request.RawUrl);
  }
 else{ Response.Write(@"<script language='javascript'>alert
```

```
Cart cart = null;
if (Session["cart"] != null)
{
    cart = (Cart) Session["cart"];
}
Cart newCart = new Cart(cart);
if (newCart.AddtoCart(product, 1))
{
    Session["cart"] = newCart;
    Response.Redirect(Request.RawUrl);
```

Cart.cs :

```
public class Cart
{
    public List<MyProducts> items = null
    public int totalquantity=0
    public int total.price =0
    public Cart (cart OldCart)
    {
        if (OldCart1 = null)
        { items = Old cart. items;
          totalquantity = Old cart. totalquantity;
          total price = Old cart. total price;
    }
```

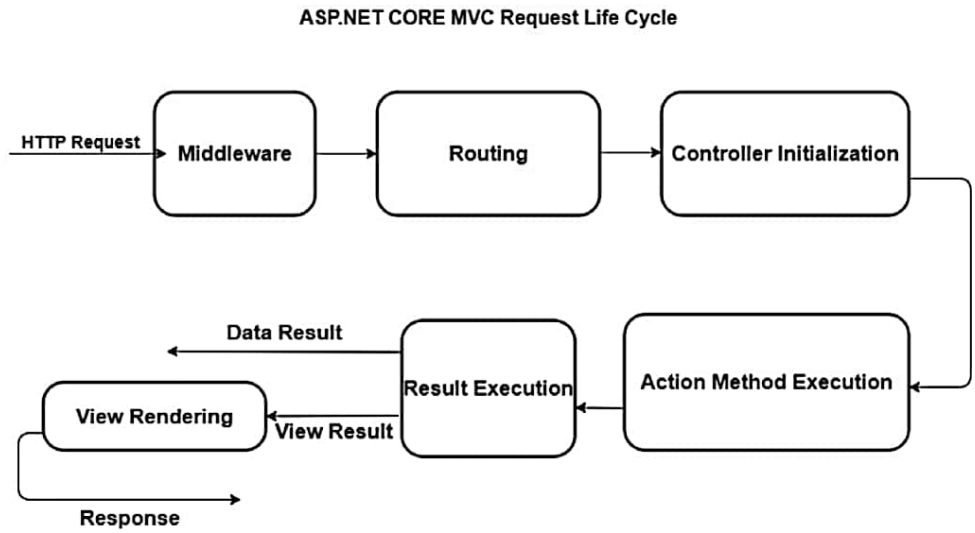
```
Public bool RemoveFromCart()
{
MyProduct previousitem = items.FirstorDefault(x=> x.product.id
                        == product.Id)
if (previousitem != null)
{
  total price = 1 if(previous item.SubQuantity == quantity
          || quantity = -1)
  {
    total price = totalprice - (previous item.SubQuantity
        *previous item.product.price);
    totalprice--;
    items.Remove(previous item);
    return true;
  }
  if else
    {
      total price = total price - (previous item.product.
        price*quantity);
    Previous item.SubTotal = previous item.SubTotal - (previous
      item.product.price *quantity)
previous item.SubQuantity = Previous Item.SubQuantity - quantity;
  return true;
}
```

```
    }
    else
    {
        return false;
    }
}
public bool Add to Cart()
{
  if (items != null)
  {
    MyProduct myproduct = new myProduct();
    myproduct.productproduct = product;
    myproduct.Subquantity = quantity;
    myproduct.SubTotal = product*price*quantity;
    item = new List<MyProduct>;
    items.Add(myproduct);
    totalquantity++;
    totalprice = myproduct.SubTotal;
    return true;
  }
```

```
else
{
  MyProduct previousitem = items.FirstorDefault (x=> x.productid==
        Product.id)
  if (previous item != null)
    {
      total price = quantity *previous item.product.price;
    previousitem.Subquantity= previous item.Subquantity +quantity;
    previous item. SubTotal= previous item. SubTotal + previous item.
        product.price *quantity;
    }
  else
    {
      MyProduct. product = new MyProduct();
      Product. product = product;
      Product. SubQuantity= quantity;
      product. Sub Total = product.price *quantity;
      items. Add (product);
      totalQuantity ++;
    totalprice = product. sub. Total
    ~~product. price * SubTotal~~ + totalprice;
    return true;
```

Introduction

The ASP.NET Core MVC Request Life Cycle is a sequence of events, stages or components that interact with each other to process an HTTP request and generate a response that goes back to the client. In this article, we will discuss each and every stage of ASP.NET Core MVC Request Life Cycle in detail.



The request life cycle consists of various stages which are
:

Middleware

Middleware component forms the basic building block of application HTTP pipeline. These are a series of components that are combined to form a request pipeline in order to handle any incoming request.

request.

Routing

Routing is a middleware component that implements MVC framework. The routing Middleware component decides how an incoming request can be mapped to Controllers and actions methods, with the help of convention routes and attribute routes.

Controller Initialization

At this stage of ASP.NET MVC Core Request Life Cycle, the process of initialization and execution of controllers takes place. Controllers are responsible for handling incoming requests. The controller selects the appropriate action methods on the basis of route templates provided.

Action method execution

After the controllers are initialized, the action methods are executed and returns a view which contains Html document to be returned as response to the browser.

Result Execution

During this stage of ASP.NET MVC Core Request Life Cycle, result i.e. the response generated to the original HTTP request, is executed. If an action method returns a view result, the MVC view engine
renders a view and returns the HTML respon

Life Cycle, result i.e. the response generated to the original HTTP request, is executed. If an action method returns a view result, the MVC view engine
renders a view and returns the HTML response. If r

Now, we will discuss each stage briefly

Middleware

Middleware component forms the basic building block of the application'a HTTP pipeline. These are aseries of components that are combined to form a request pipeline in order to handle any incoming request. Whenever a new request comes, it is passed to the first middleware component. The component then decides, whether to generate a response after handling that incoming request or to pass it to the next component. The response is sent back along these components, once the request has been handled.

Most middleware components are implemented as standalone classes, which are content generating middleware. Now, we will be creating a content generating middleware component by adding ContentComponent.cs class.

```
01. using System.Text;
02. using System.Threading.Tasks;
03. using Microsoft.AspNetCore.Http;
04. namespace WebApplication
05. {
06.
    public class ContentComponent
07.     {
08.
    private RequestDelegate nextC
09.
```

This ContentComponent middleware component is defining a constructor, and inside constructor its taking RequestDelegate object. This RequestDelegate object represents the next middleware component. ContentComponent component also defines an Invoke method. When ASP.NET Core application receives an incoming request, the Invoke method is called.

The HttpContext argument of the Invoke method provides information about the incoming HTTP request and the generated response that will be sent back to the client. The invoke method of ContentComponent class checks whether incoming request is sent to /edit URL or not. If the request has been sent to /edit url then, in response a text message is sent.

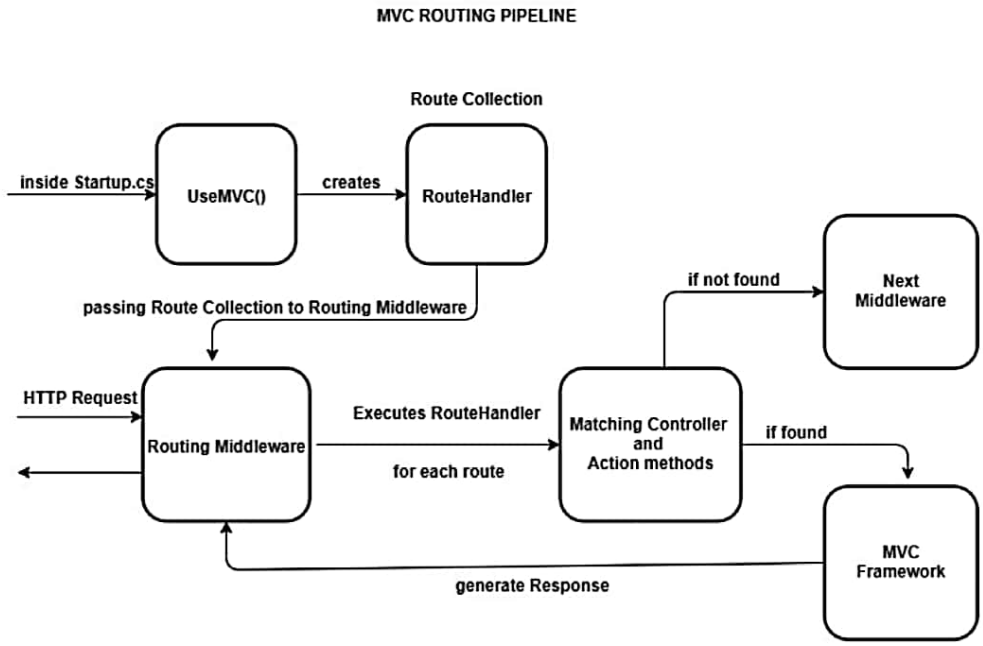
Routing

Routing is a middleware component that implements MVC framework. In ASP.NET Core, the routing system is used to handle MVC URLs. The routing Middleware component decides how an incoming request can be mapped to Controllers and actions methods, with the help of convention routes and attribute routes. Routing bridges middleware and MVC framework by mapping incoming request to controller action methods.

In a particular application, MVC registers one or more routes using MapRoute method. MVC provides default routes that are given a name and a template to match incoming request URLs. Controllers and action variables are placeholders

se routes are passed into and consumed by routing middleware.

The MVC routing pipeline



In ASP.NET Core, routing maps an incoming request to RouteHandler class, which is then passes as the collection of routes to the routing middleware. The routing middleware executes MVC RouteHandler for each route. If a matching controller and action method is found on a particular route, then the requested data is forwarded to the rest of the MVC framework which will generate a response.

There are two types of routing available in MVC which are:

Convention routing

This routing type uses application wide patterns to match a different URL to various controller a methods. Convention routes represent defau

methods. Convention routes represent default sible routes that can be defined with the help of controller and action methods.

Attribute routing

This type of routing is implemented through attributes and applied directly to a specific controller or action method.

The convention routing methodology is implemented inside startup.cs file. In startup.cs, the UseMvc() method registers the routes, that are being supplied to it as a parameter inside MapRoute method.

```
01.  public void Configure(IApplicatic
02.
03.   app.UseMvc(routes => {
04.      routes.MapRoute(
05.     name: "default",
06.     template: "
     {controller=Home}/{action=Index}/
07.   });
08.  }
```

A route attribute is defined on top of an action method inside controller class.

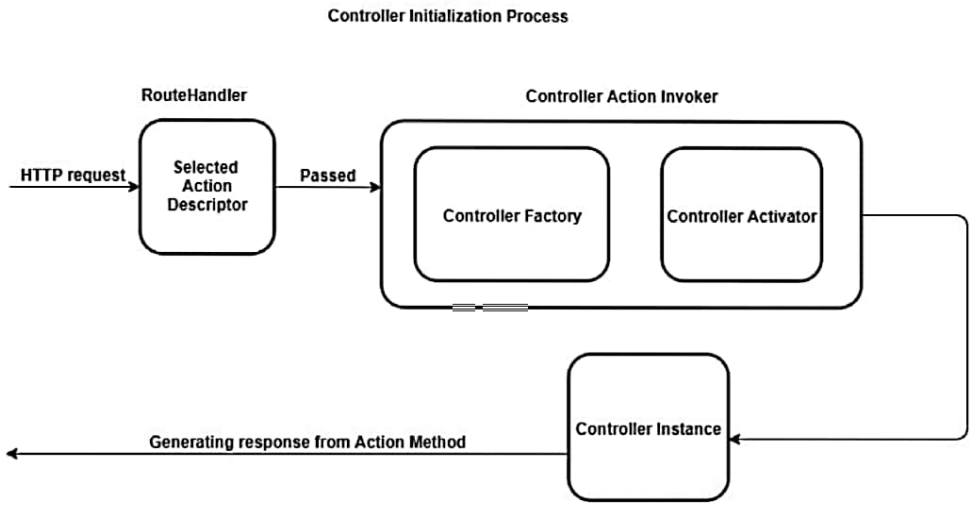
```
01.  public class MoviesController: Cc
02.    [Route("movies / released / {ye
03.      public ActionResult ByRelease
04.       return Content(year + "/"+mc
05.       }
06.      }
```

Controller Initialization

At this stage, the process of initialization and execution of controllers takes place. Controll

Controller Initialization

At this stage, the process of initialization and execution of controllers takes place. Controllers are responsible for handling incoming requests which is done by mapping request to appropriate action method. The controller selects the appropriate action methods (to generate response) on the basis of route templates provided. A controller class inherits from controller base class. A controller class suffix class name with the Controller keyword.



The MVC RouteHandler is responsible for selecting an action method candidate in the form of action descriptor. The RouteHandler then passes the action descriptor into a class called Controller action invoker. The class called Controller factory creates instance of a controller to be used by controller action method. The controller factory class depends on controller activator for controller instantiation.

antiation.

After action method is selected, instance of controller is created to handle request. Controller instance provides several features such as action methods, action filters and action result. The activator uses the controller type info property on action descriptor to instantiate the controller by name. once the controller is created, the rest of the action method execution pipeline can run.

The controller factory is the component that is responsible for creating controller instance. The controller factory implements an interface called IControllerFactory. This interface contains two methods which are called CreateController and ReleaseController.

Action Method Execution Process

Authorization Filters

Authorization filters authorize or deny any incoming request.

Controller Creation

Instantiate Controller object.

Model Binding

Model Binding bind incoming request to Action method parameters.

Action Filters

action method is invoked.

Action Method Execution

Action method inside controller classes executes logic to create Action Result. Action method returns action results to generate response.

Action Filters

OnActionExecuted method called is always called after the action method has been invoked.

Authorization Filters

In ASP.NET Core MVC, there is an attribute called Authorize. Authorize is a filter, which can be applied to an action and it will be called by a MVC Core framework before and after that action or its result are executed.

These filters are used to provide security to application, including user authorization. If authorize attribute is applied to an action method, before the action is executed, the attribute will check if the current user is logged in or not. If not it will redirect the user to the login page. Authorization filters authorize or deny any incoming request.

In the below code example, Authorize Authorization filter(predefined filter) has been used over Index action method.

```
01. [Authorize]
02. public ViewResult Index() {
03.  return View();
```

methods.

Action Filters

Action filters are used to execute tasks instantly before and after action method execution is performed. In order to apply some logic to action methods, without having to add extra code to the controller class, action filters can be used either above the action method itself or above the Controller class.

Action filter implements two types of interfaces which are IActionFilter and IAsyncActionFilter.

The IActionFilter interface which defines action filter is as follows

```
01. public interface IActionFilter: I
02.  void OnActionExecuting(ActionExe
03.  void OnActionExecuted(ActionExec
04. }
```

The method OnActionExecuting is always called before the action method is invoked, and the method OnActionExecuted is always called after the action method has been invoked, whenever an action filter is applied to an action method.

Whenever an action filter is created, the data is provided using two different context classes, ActionExecutingContext and ActionExecutedContext.

Now I will demonstrate, how to create a custom action filter by deriving a class from the ActionFilterAttribute class. A class file called ActionAttribute.cs has been created which de...

```
attribute");
  }
}
```

Action Method Execution

Action methods return objects that implements the IActionResult interface from the Microsoft.AspNetCore.Mvc namespace. The various types of response from the controller such as rendering a view or redirecting the client to another URL, all these responses are handled by IActionResult object, commonly known as action result.

When an an action result is returned by a specific action method, MVC calls its ExecuteResultAsync method, which generates response on behalf of the action method. The ActionContext argument provides context data for generating the response, including the HttpResponse object.

Controllers contain Action methods whose responsibility is to generate a response to an incoming request. An action method is any public method inside a controller that responds to incoming requests.

```
Public class HomeController: Cont
 Public IActionResult Edit() {
  Return View();
 }
}
```

Here, Edit action method is defined inside Home controller.

```
using Microsoft.AspNetCore.Mv
```