Abstracting SQL

Injecting environment variables or dynamic values into an SQL query in a Razor page requires careful handling for security (to avoid SQL injection) and maintainability.

Here's a structured approach:

Contents

☑ Best Practice: Use Parameterized Queries

Never concatenate raw environment variables into SQL strings. Instead, use **parameterized queries** with SqlCommand or EF Core.

Option 1: Using Environment Variables in Razor Page

You can access environment variables in Razor via:

```
@{
   var serverName = Environment.GetEnvironmentVariable("DB_SERVER");
}
```

Then, when building your connection string:

```
var connectionString = $"Server={serverName};Database=MyDb;User Id=…;Password=…";
```

Option 2: Injecting into SQL Query Safely

If you need to query dynamically:

```
@{
   var serverName = Environment.GetEnvironmentVariable("DB_SERVER");
   var query = "SELECT * FROM MyTable WHERE ServerName = @ServerName";

   using (var conn = new SqlConnection(connectionString))
   using (var cmd = new SqlCommand(query, conn))
   {
      cmd.Parameters.AddWithValue("@ServerName", serverName);
      conn.Open();
      var reader = cmd.ExecuteReader();
      // Process results
   }
}
```

Option 3: Use Configuration Instead of Environment Variables

For ASP.NET Core Razor Pages, prefer IConfiguration:

```
@inject IConfiguration Configuration


@{
   var serverName = Configuration["DB_SERVER"];
}
```

Security Note

- Avoid string interpolation in SQL queries.
- Validate environment variable values before use.
- Consider using **stored procedures** or **ORM (EF Core)** for better abstraction.

Part 1: Full Razor Page Example with Environment Variable Injection

Here's a minimal Razor Page that:

- Reads an environment variable (DB_SERVER)
- Builds a connection string dynamically
- Executes a parameterized SQL query
- Displays results safely

```
@page
@using System.Data.SqlClient
@model IndexModel
@{
    var serverName = Environment.GetEnvironmentVariable("DB_SERVER");
    var connectionString = $"Server={serverName};Database=MyDb;User Id=myuser;Password=mypassword;";
    var results = new List<string>();

    using (var conn = new SqlConnection(connectionString))
    {
        conn.Open();
        var query = "SELECT Name FROM MyTable WHERE ServerName = @ServerName";
        using (var cmd = new SqlCommand(query, conn))
        {
            cmd.Parameters.AddWithValue("@ServerName", serverName);
            using (var reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    results.Add(reader.GetString(0));
                }
            }
        }
```

```
    }
}


<h2>Results from @serverName</h2>
<ul>
@foreach (var item in results)
{
    <li>@item</li>
}
</ul>
```

Key Points:

- Environment.GetEnvironmentVariable("DB_SERVER") pulls the server name.
- Parameterized query prevents SQL injection.
- Results are rendered in a simple <ul> list.

☑ Part 2: Cleaner Approach Using EF Core

Instead of raw SQL, use EF Core with dynamic connection strings:

Step 1: Configure EF Core in Program.cs

```
var serverName = Environment.GetEnvironmentVariable("DB_SERVER");
var connectionString = $"Server={serverName};Database=MyDb;User
Id=myuser;Password=mypassword;";


builder.Services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer(connectionString));
```

Step 2: Razor Page Code

```
@page
@model IndexModel
@inject MyDbContext DbSERVER")</h2>@inject MyDbContext Db
<ul>
```

```
@foreach (var item in Db.MyTable.Where(x => x.ServerName == Environment.GetEnvironmentVariable("DBSERVER")))
{
    <li>@item.Name</li>
}
</ul>
```

Why cleaner?

- No manual connection handling.
- LINQ queries are safer and easier to maintain.
- EF Core handles parameterization automatically.

Full Walkthrough

✅ **Program.cs setup**\ ✅ **DbContext configuration**\ ✅ **Razor Page with dynamic dropdown**\ ✅ **Async query execution**\ ✅ Optional: Switching servers dynamically at runtime

---

Step 1: Project Setup

Start with a new ASP.NET Core Razor Pages project:

```
dotnet new webapp -n RazorDynamicSQLDemo
cd
```

Install EF Core SQL Server provider:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Step 2: Create the Model

Create a simple model in Models/MyTable.cs:

```
namespace RazorDynamicSQLDemo.Models
{
    public class MyTable
    {
        public int Id { get; set; }
```

```
        public string Name { get; set; }
        public string ServerName { get; set; }
    }
}
```

Step 3: Create DbContext

In Data/MyDbContext.cs:

```
using Microsoft.EntityFrameworkCore;
using RazorDynamicSQLDemo.Models;

namespace RazorDynamicSQLDemo.Data
{
    public class MyDbContext : DbContext
    {
        public MyDbContext(DbContextOptions<MyDbContext> options) :
base(options) { }

        public DbSet<MyTable> MyTable { get; set; }
    }
}
```

Step 4: Configure Program.cs

Inject dynamic connection string using environment variable:

```
using RazorDynamicSQLDemo.Data;using);

// Get DB server from environment variable
var serverName = Environment.GetEnvironmentVariable("DB_SERVER") ??
"localhost";
var connectionString = $"Server={serverName};Database=MyDb;User
Id=myuser;Password=mypassword;";
```

```
builder.Services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer(connectionString));

builder.Services.AddRazorPages();

var app = builder.Build();

app.MapRazorPages();
app.Run();
using Microsoft.EntityFrameworkCore;
```

Step 5: Razor Page with Dynamic Dropdown

Create Pages/Index.cshtml:

```
@page @model IndexModel @inject RazorDynamicSQLDemo.Data.MyDbContext Db

# Select Server

@if (Model.Results?.Any() == true) {

## Results from @Model.SelectedServer

    @foreach (var item in Model.Results) {
- @item.Name
    }

}
```

Index.cshtml.cs (Async + Dynamic Query)

```
using Microsoft.AspNetCore.Mvc;usingDynamicSQLDemo.Data;
using RazorDynamicSQLDemo.Models;
using Microsoft.EntityFrameworkCore;

public class IndexModel : PageModel
{
    private readonly MyDbContext db;

    public IndexModel(MyDbContext db)
    {
        db = db;
    }

    [BindProperty]
    public string SelectedServer { get; set; }

    public List<string> Servers { get; set; } = new() { "ServerA",
"ServerB", "ServerC" };
    public List<MyTable> Results { get; set; }

    public async Task OnPostAsync()
    {
        Results = await _db.MyTable
            .Where(x => x.ServerName == SelectedServer)
            .ToListAsync();
    }
}
using Microsoft.AspNetCore.Mvc.RazorPages;
```

✅ Optional Enhancements

- **Dynamic server switching at runtime**: Instead of hardcoding servers, fetch from config or DB.
- **Async everywhere**: Already implemented in OnPostAsync.
- **Security**: Validate SelectedServer before querying.

Training Flow

This tutorial covers:

- Reading environment variables
- Configuring EF Core dynamically
- Razor Pages with dropdown selection
- Async queries
- Safe parameterization via LINQ

Here's your **complete copy-paste-ready solution** followed by a **step-by-step markdown guide** with explanations.

✅ Full Solution (All Files in One Block)

```
// ======================
// Program.cs
// ======================
using RazorDynamicSQLDemo.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Get DB server from environment variable or fallback
var serverName = Environment.GetEnvironmentVariable("DBSERVER") ?? "localhost";
var connectionString = $"Server={serverName};Database=MyDb;User Id=myuser;Password=mypassword;";

builder.Services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer(connectionString));

builder.Services.AddRazorPages();

var app = builder.Build();

app.MapRazorPages();
app.Run();

// ======================
// Data/MyDbContext.cs
// ======================
using Microsoft.EntityFrameworkCore;
using RazorDynamicSQLDemo.Models;
```

```
namespace RazorDynamicSQLDemo.Data
{
    public class MyDbContext : DbContext
    {
        public MyDbContext(DbContextOptions<MyDbContext> options) :
base(options) { }

        public DbSet<MyTable> MyTable { get; set; }
    }
}
```

```
// ======================
// Models/MyTable.cs
// ======================
namespace RazorDynamicSQLDemo.Models
{
    public class MyTable
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string ServerName { get; set; }
    }
}

// ======================
// Pages/Index.cshtml
// ======================
@page
@model IndexModel
@inject RazorDynamicSQLDemo.Data.MyDbContext Db

<h2>Select Server</h2>
<form method="post">
    <select name="SelectedServer">
        @foreach (var server in Model.Servers)
        {
            <option value="@server">@server</option>
        }
    </select>
    <button type="submit">Query</button>
</form>
```

```
@if (Model.Results?.Any() == true)
{
    <h3>Results from @Model.SelectedServer</h3>
    <ul>
        @foreach (var item in Model.Results)
        {
            <li>@item.Name</li>
        }
    </ul>
}

// ======================
// Pages/Index.cshtml.cs
// ======================
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorDynamicSQLDemo.Data;
using RazorDynamicSQLDemo.Models;
using Microsoft.EntityFrameworkCore;

public class IndexModel : PageModel
{
    private readonly MyDbContext db;

    public IndexModel(MyDbContext db)
    {
        db = db;
    }
```

```
    [BindProperty]
    public string SelectedServer { get; set; }


    public List<string> Servers { get; set; } = new() { "ServerA",
"ServerB", "ServerC" };
    public List<MyTable> Results { get; set; }


    public async Task OnPostAsync()
    {
        Results = await db.MyTable
            .Where(x => x.ServerName == SelectedServer)
            .ToListAsync();
    }
}
```

✅ Step-by-Step Markdown Guide

Step 1: Create the Project

dotnet new webapp -n RazorDynamicSQLDemo

cd RazorDynamicSQLDemo

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

Step 2: Add the Model

Create Models/MyTable.cs:

- Represents a table with Id, Name, and ServerName.

Step 3: Add DbContext

Create Data/MyDbContext.cs:

- Inherits DbContext.
- Exposes DbSet<MyTable> for EF Core queries.

Step 4: Configure Program.cs

- Reads DB_SERVER from environment variables.
- Builds dynamic connection string.
- Registers MyDbContext with EF Core.

Step 5: Razor Page UI

Pages/Index.cshtml:

- Dropdown for selecting server.
- Displays query results dynamically.

Step 6: Razor Page Code-Behind

Pages/Index.cshtml.cs:

- Handles form submission.

- Executes async EF Core query filtered by SelectedServer.

☑ Optional Enhancements

- **Dynamic server list from config or DB** instead of hardcoded.
- **Async everywhere** (already implemented).
- **Validation** for SelectedServer before querying.
- **Switching DB at runtime**: Requires scoped DbContext factory (advanced scenario).

☑ Why This Is Best Practice

- Uses **environment variables** for server configuration.
- Avoids **SQL injection** by using EF Core LINQ.
- Implements **async** for scalability.
- Keeps Razor Page clean and tutorial-friendly.

In classic ASP VBScript

The primary method for abstracting SQL statements and protecting against injection is to move the SQL logic to a separate function or file and, more importantly, **use ADO Parameterized Queries or stored procedures**. This approach treats user input as data, not executable code, which is the only sure way to prevent SQL injection.

1. Identify the Vulnerable Code

First, identify where the user input from the query string is concatenated directly into the SQL string. A common vulnerable pattern looks like this:

vbscript

Dim conn, rs, sql, userID

userID = Request.QueryString("id")

' VULNERABLE: Direct concatenation of user input

sql = "SELECT * FROM Users WHERE UserID = " & userID

Set conn = Server.CreateObject("ADODB.Connection")

conn.Open "your_connection_string"

Set rs = conn.Execute(sql)

2. Abstract and Parameterize the Query

Move the data access logic into a reusable function or sub-routine, and use ADO **Command** objects with **Parameters** to safely handle user input.

vbscript

```
<%
' Function to safely retrieve user data using parameterized query
Function GetUserData(p_userID)
    Dim cmd, param, rs, conn

    Set conn = Server.CreateObject("ADODB.Connection")
    ' IMPORTANT: Replace with your actual connection string
    conn.Open "your_connection_string"

    Set cmd = Server.CreateObject("ADODB.Command")
    Set cmd.ActiveConnection = conn
    cmd.CommandText = "SELECT * FROM Users WHERE UserID = @UserID"
    cmd.CommandType = 1 ' adCmdText (or 4 for stored procedure, if applicable)

    ' Create and append a parameter for the user ID
    ' adInteger (3) is used here assuming UserID is an integer type in the database
    ' Adjust the type, direction, and size based on your database schema
    Set param = cmd.CreateParameter("@UserID", 3, 1, , p_userID) ' adTypeInteger (3), adParamInput (1)
    cmd.Parameters.Append param

    ' Execute the command and return the recordset
    Set rs = cmd.Execute()
    Set GetUserData = rs
```

```
    ' Clean up the command and connection objects when finished
    ' (Connection needs to remain open for the recordset to be used outside the function)
End Function

' Main page logic
Dim userDataRS, requestedUserID
' Retrieve input from query string
requestedUserID = Request.QueryString("id")

' Use the safe, abstracted function
Set userDataRS = GetUserData(requestedUserID)

' Now you can safely display the data
If Not userDataRS.EOF Then
    Response.Write "Welcome, " & userDataRS("UserName")
Else
    Response.Write "User not found."
End If

' Don't forget to close the recordset and connection when the page finishes
userDataRS.Close
userDataRS.ActiveConnection.Close
%>
```

Key Benefits of this Abstraction

- **Security**: Parameterized queries are the primary defense against SQL injection, ensuring user input is treated as literal values, not part of the SQL command.
- **Maintainability**: SQL logic is centralized in functions/data access layers, separate from the presentation (HTML) code, making it easier to manage and debug.
- **Reusability**: The GetUserData function can be reused across different page

In C#

The concept of "abstracting SQL statements" from a query string variable or request body generally refers to securing your application by **separating user-provided data from executable SQL commands**. The best practice for achieving this in ASP.NET (C#) is to use **parameterized queries** or **stored procedures**, not by attempting to manually extract or parse potential SQL statements from the raw user input.

Why Direct Extraction is a Bad Idea

Attempting to parse user input to "abstract" or "clean" SQL statements is strongly discouraged because:

- It is nearly impossible to create a foolproof sanitizing routine that covers every potential SQL injection technique, leaving your application vulnerable.
- The fundamental issue is treating user input as part of the SQL command itself. The solution is to ensure user input is *always* treated as data (literal values), never as executable code.

The Correct Approach: Use Parameterized Queries

Parameterized queries are the most effective defense against SQL injection. When you use parameters, the database engine receives the SQL command and the data separately.

Here is how you can implement this using SqlCommand in C# (for classic ASP.NET) or Entity Framework (for modern ASP.NET Core):

Classic ASP.NET / ADO.NET Example (C#)

This example shows how to take input from a query string (Request.QueryString["username"]) and use it safely:

csharp

```
using System.Data.SqlClient;

// ... other using statements


// 1. Retrieve the user input from the request (e.g., query string or form body)

string userInputName = Request.QueryString["username"];


// 2. Define the SQL query with a named parameter placeholder (e.g., @Name)

string sqlQuery = "SELECT * FROM Users WHERE Name = @Name";


// 3. Use the SqlCommand object and add the user input as a parameter

using (SqlConnection connection = new SqlConnection(yourConnectionString))
```

```
{
    using (SqlCommand command = new SqlCommand(sqlQuery, connection))
    {
        // Add the parameter and its value. The framework handles the rest.
        command.Parameters.AddWithValue("@Name", userInputName); // This automatically handles escaping and typing

        connection.Open();
        SqlDataReader reader = command.ExecuteReader();

        // Process the results...
        reader.Close();
    }
}
```

ASP.NET Core / Entity Framework Core Example (C#)

If you are using Entity Framework Core, string interpolation with FromSqlInterpolated automatically creates parameters, making it safe by default:

csharp

```
public IActionResult GetUser(string name)
{
    // EF Core automatically handles parameterization with string interpolation
    var user = context.Users
                .FromSqlInterpolated($"SELECT * FROM Users WHERE Name = {name}")
                .ToList();
    return View(user);
}
```

By adopting parameterized queries or an Object-Relational Mapper (ORM) like Entity Framework, you "abstract" the SQL execution logic from the user-provided data, thus preventing SQL injection vulnerabilities. You can find more information on preventing attacks in the OWASP SQL Injection Prevention Cheat Shee