Contact Us Form System – Repository + Service Pattern (ADO.NET)

Task 1: Create the Project and Structure

Problem Statement:
Set up an ASP.NET Core Web API using ADO.NET with clean architecture.

Solution Outline:

1. Run `dotnet new webapi -n ContactUsSystem`
2. Add folders:
   - `Models`
   - `DTOs/ContactMessage`
   - `Controllers`
   - `Repositories`
   - `Services`
   - `Interfaces/Repositories`
   - `Interfaces/Services`

Task 2: Define the `ContactMessage` Model

Problem Statement:
Create the core model for contact message data.

Solution Outline:

Create `Models/ContactMessage.cs`:

- `Id` (int)
- `Name` (string)
- `Email` (string)
- `Subject` (string)
- `Message` (string)
- `CreatedAt` (DateTime)

Task 3: Create DTOs for Contact Message

Problem Statement:
Use DTOs to transfer data between layers.

Solution Outline:

In `DTOs/ContactMessage`, define:

- `CreateContactMessageDTO`
- `ContactMessageDTO`

Used for API input/output formatting.

Task 4: Create SQL Table

Problem Statement:
Create the table to store contact messages in SQL Server.

Solution Outline:

```sql
CREATE TABLE ContactMessages (
  Id INT PRIMARY KEY IDENTITY,
  Name NVARCHAR(100) NOT NULL,
  Email NVARCHAR(100) NOT NULL,
  Subject NVARCHAR(200),
  Message NVARCHAR(MAX) NOT NULL,
  CreatedAt DATETIME NOT NULL DEFAULT GETDATE()
);
```

Task 5: Create Repository Interface and Implementation

Problem Statement:
Handle database access using ADO.NET.

Solution Outline:

1. `Interfaces/IContactRepository.cs`
2. `Repositories/ContactRepository.cs` and `Repositories/DataAdapterContactRepository.cs`
3. Use:
   - `SqlConnection`
   - `SqlCommand`
   - `SqlDataReader`
   - and in `DataAdapterContactRepository`: (We have two implementations)
   - `SqlDataAdapter`
   - `DataTable`
   - `SqlDataBuilder`
4. Methods:

Inject `IConfiguration` into the repository to access the connection string:

```
private readonly IConfiguration _configuration;

public ContactRepository(IConfiguration configuration)
{
    _configuration = configuration;
}
```

Task 6: Create Service Interface and Implementation

Problem Statement:
Business logic and mapping layer between controller and repository.

Solution Outline:

1. `Interfaces/Services/IContactService.cs`
2. `Services/ContactService.cs`
3. Logic:
    - Validate
    - Map DTOs to models
    - Call repository methods

Task 7: Implement the `ContactController`

Problem Statement:
Handle API requests using controller → service → repo.

Solution Outline:

1. Create `ContactController.cs`
2. Inject `IContactService`
3. Add endpoints:
   - `POST /api/contact`
   - `GET /api/contact`
   - `GET /api/contact/{id}`
   - `DELETE /api/contact/{id}`

Task 8: Add Input Validation

Problem Statement:
Validate incoming request data.

Solution Outline:

1. Use:
   - `[Required]`, `[EmailAddress]`, `[StringLength]` in DTOs
2. In controller:

```
if (!ModelState.IsValid) return BadRequest(ModelState);
```

Task 9: Secure Against SQL Injection

Problem Statement:
Prevent injection vulnerabilities in queries.

Solution Outline:

Use parameterized queries:

```
cmd.Parameters.AddWithValue("@Email", dto.Email);
```

Never build raw SQL strings directly with user input.

Task 10: Add Logging and Error Handling

Problem Statement:
Log application errors and show friendly error responses.

Solution Outline:

1. Inject `ILogger<ContactController>`
2. Wrap logic in `try-catch` blocks
3. Return:
   - `500 InternalServerError` on exceptions
   - `404 NotFound` when needed

Task 11: Test the Contact API

Problem Statement:
Manually test the endpoints.

Solution Outline:

1. Open Swagger `/swagger`
2. Test all endpoints:
    - Add message
    - View messages
    - View by ID
    - Delete message

Task 12: Prepare for Deployment

Problem Statement:
Get ready to publish and deploy to production.

Solution Outline:

1. Enable CORS & HTTPS redirection
2. Use `dotnet publish -c Release`
3. Host on IIS, Azure, or other platform

✅ Contact Us System – Repository + Service + ADO.NET