**Checklist for ASP.NET Web API Best Practices**

**1. Project Structure**

* **Organize by Feature**: Structure project by feature rather than by layer. This helps in maintaining and scaling the project.
  + Controllers/
  + Models/
  + Services/
  + Dtos/

**2. Controllers**

* **Keep Controllers Thin**: Controllers should only handle HTTP requests and responses. Business logic should be placed in services.

public class CreditCardController : ControllerBase

{

private readonly ICreditCardService \_service;

public CreditCardController(ICreditCardService service)

{

\_service = service;

}

[HttpPost("validate")]

public ActionResult<bool> ValidateCreditCard([FromBody] CreditCardModel model)

{

if (string.IsNullOrWhiteSpace(model.CreditCardNumber))

{

return BadRequest("Credit card number is required.");

}

bool isValid = \_service.ValidateCreditCard(model.CreditCardNumber);

return Ok(isValid);

}

}

**3. Services**

* **Business Logic**: Encapsulate all business logic in services.

public interface ICreditCardService

{

bool ValidateCreditCard(string creditCardNumber);

}

public class CreditCardService : ICreditCardService

{

public bool ValidateCreditCard(string creditCardNumber)

{

// Luhn validation logic

}

}

**4. Dependency Injection**

* **Use Dependency Injection**: Register services, repositories, and other dependencies in the DI container.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddSwaggerGen();

services.AddScoped<ICreditCardService, CreditCardService>();

}

**5. DTOs and Mappers**

* **Use DTOs**: Data Transfer Objects (DTOs) should be used to define the structure of request and response bodies.

public class CreditCardModel

{

public string CreditCardNumber { get; set; }

}

* **Automapper**: Use Automapper for mapping between entities and DTOs.

**6. Validation**

* **Model Validation**: Use data annotations and custom validators to ensure incoming data is valid.

public class CreditCardModel

{

[Required]

[CreditCard]

public string CreditCardNumber { get; set; }

}

* **Fluent Validation**: Consider using FluentValidation for complex validation scenarios.

**7. Error Handling**

* **Global Error Handling**: Implement global error handling middleware to manage exceptions and return meaningful error responses.

public class ErrorHandlingMiddleware

{

private readonly RequestDelegate \_next;

public ErrorHandlingMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

try

{

await \_next(context);

}

catch (Exception ex)

{

await HandleExceptionAsync(context, ex);

}

}

private static Task HandleExceptionAsync(HttpContext context, Exception exception)

{

context.Response.ContentType = "application/json";

context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;

return context.Response.WriteAsync(new ErrorDetails()

{

StatusCode = context.Response.StatusCode,

Message = "Internal Server Error from the custom middleware."

}.ToString());

}

}

**8. Security**

* **HTTPS**: Enforce HTTPS for all requests.
* **Authentication and Authorization**: Implement robust authentication and authorization mechanisms using JWT tokens or OAuth.
* **Data Protection**: Use proper data protection mechanisms for sensitive information.

**9. Logging and Monitoring**

* **Logging**: Use a logging framework (e.g., Serilog, NLog) to log information, warnings, and errors.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env, ILogger<Startup> logger)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

app.Use(async (context, next) =>

{

logger.LogInformation("Handling request: " + context.Request.Path);

await next.Invoke();

logger.LogInformation("Finished handling request.");

});

}

* **Monitoring**: Integrate monitoring tools like Application Insights or Prometheus.

**10. API Documentation**

* **Swagger/OpenAPI**: Use Swagger to generate API documentation and make it accessible.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "CreditCardValidationApi v1");

c.RoutePrefix = string.Empty;

});

}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

**11. Testing**

* **Unit Testing**: Unit tests for controllers CreditCardControllerTests

using CreditCardValidationApi.Controllers;

using CreditCardValidationApi.Models;

using CreditCardValidationApi.Repository;

using Microsoft.AspNetCore.Mvc;

using Moq;

using Xunit;

namespace CreditCardValidationApi.Tests.ControllerTests

{

public class CreditCardControllerTests

{

[Fact]

public void ValidateCreditCard\_ValidCard\_ReturnsTrue()

{

// Arrange

var mockService = new Mock<ICreditCardService>();

mockService.Setup(service => service.ValidateCreditCard(It.IsAny<string>())).Returns(true);

var controller = new CreditCardController(mockService.Object);

var model = new CreditCardModel { CreditCardNumber = "4111111111111111" };

// Act

var result = controller.ValidateCreditCard(model);

// Assert

var okResult = Assert.IsType<OkObjectResult>(result.Result);

Assert.True((bool)okResult.Value);

}

[Fact]

public void ValidateCreditCard\_InvalidCard\_ReturnsFalse()

{

// Arrange

var mockService = new Mock<ICreditCardService>();

mockService.Setup(service => service.ValidateCreditCard(It.IsAny<string>())).Returns(false);

var controller = new CreditCardController(mockService.Object);

var model = new CreditCardModel { CreditCardNumber = "1234" }; // Invalid card number

// Act

var result = controller.ValidateCreditCard(model);

// Assert

var okResult = Assert.IsType<OkObjectResult>(result.Result);

Assert.False((bool)okResult.Value);

}

[Fact]

public void ValidateCreditCard\_EmptyCardNumber\_ReturnsBadRequest()

{

// Arrange

var mockService = new Mock<ICreditCardService>();

var controller = new CreditCardController(mockService.Object);

var model = new CreditCardModel { CreditCardNumber = null }; // Empty card number

// Act

var result = controller.ValidateCreditCard(model);

// Assert

var badRequestResult = Assert.IsType<BadRequestObjectResult>(result.Result);

Assert.Equal("Credit card number is required.", badRequestResult.Value);

}

}

}