**Technical Design Document**

**API Architecture and Framework**

**Overview:** This document outlines the technical design of the API architecture and framework, focusing on design principles, return types, error handling, validation, and best practices for implementation.

**1. API Architecture Overview**

**1.1 Goals:**

* Build a scalable, maintainable, and secure API architecture.
* Ensure consistent error handling, validation, and response formatting.
* Support extensibility to meet evolving business needs.

**1.2 Components:**

1. **Controller Layer:**
   * Handles incoming requests and routes them to the appropriate service layer.
   * Performs basic request validation.
2. **Service Layer:**
   * Encapsulates business logic.
   * Coordinates with repositories or external APIs.
3. **Repository Layer:**
   * Handles data access, querying, and persistence.
4. **Middleware:**
   * Manages cross-cutting concerns like authentication, logging, and exception handling.
5. **DTOs and Models:**
   * Data Transfer Objects (DTOs) for request/response encapsulation.
   * Domain models for internal representation.

**2. Design Rules and Best Practices**

**2.1 Asynchronous Programming:**

* Use Task<IActionResult> for controller action return types instead of custom user classes.
  + **Reason:**
    - Task<IActionResult> supports asynchronous programming, enabling non-blocking I/O operations.
    - It integrates seamlessly with ASP.NET Core’s middleware and pipeline.
    - Avoids tight coupling with custom classes, promoting flexibility and maintainability.

**Example:**

[HttpGet]

public async Task<IActionResult> GetUserById(int id)

{

var user = await \_userService.GetUserByIdAsync(id);

if (user == null)

return NotFound(new { message = "User not found." });

return Ok(user);

}

**2.2 Error Handling:**

* Implement a global exception handler using middleware.
* Define a common error response class:
* {
* "errorCode": "string",
* "errorMessage": "string",
* "details": "string"

}

* Use standardized HTTP status codes:
  + 200: Success
  + 400: Bad Request (validation errors)
  + 401: Unauthorized
  + 404: Not Found
  + 500: Internal Server Error

**Example Global Exception Middleware:**

public class ExceptionMiddleware

{

private readonly RequestDelegate \_next;

public ExceptionMiddleware(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)

{

var response = new

{

errorCode = "500",

errorMessage = "An unexpected error occurred.",

details = exception.Message

};

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

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

return context.Response.WriteAsync(JsonConvert.SerializeObject(response));

}

}

**2.3 Validation:**

* Perform validation at the controller level using DataAnnotations or FluentValidation.
* Return meaningful error messages for validation failures.

**Example Validation Using Data Annotations:**

public class CreateUserRequest

{

[Required]

[StringLength(50, MinimumLength = 3)]

public string Name { get; set; }

[Required]

[EmailAddress]

public string Email { get; set; }

}

[HttpPost]

public async Task<IActionResult> CreateUser([FromBody] CreateUserRequest request)

{

if (!ModelState.IsValid)

{

return BadRequest(new { errors = ModelState.Values.SelectMany(v => v.Errors).Select(e => e.ErrorMessage) });

}

var result = await \_userService.CreateUserAsync(request);

return CreatedAtAction(nameof(GetUserById), new { id = result.Id }, result);

}

**3. Framework Components**

**3.1 API Response Wrapper:**

* Wrap responses in a consistent format for success and failure cases.

**Example Success Wrapper:**

{

"data": "object",

"message": "Operation successful."

}

**Example Error Wrapper:**

{

"errorCode": "400",

"errorMessage": "Validation error.",

"details": "Name is required."

}

**3.2 Dependency Injection:**

* Use constructor injection for all dependencies.
* Register services and repositories in Startup.cs or Program.cs.

**3.3 Logging:**

* Integrate a logging framework (e.g., Serilog or NLog) for tracking errors and debugging.
* Log critical information, including:
  + Request and response payloads.
  + Exception stack traces.

**4. Conclusion**

This design document provides a robust framework for building modern APIs with scalable, maintainable, and secure architecture. The guidelines and best practices ensure consistent error handling, validation, and response formatting, fostering a seamless developer and user experience.