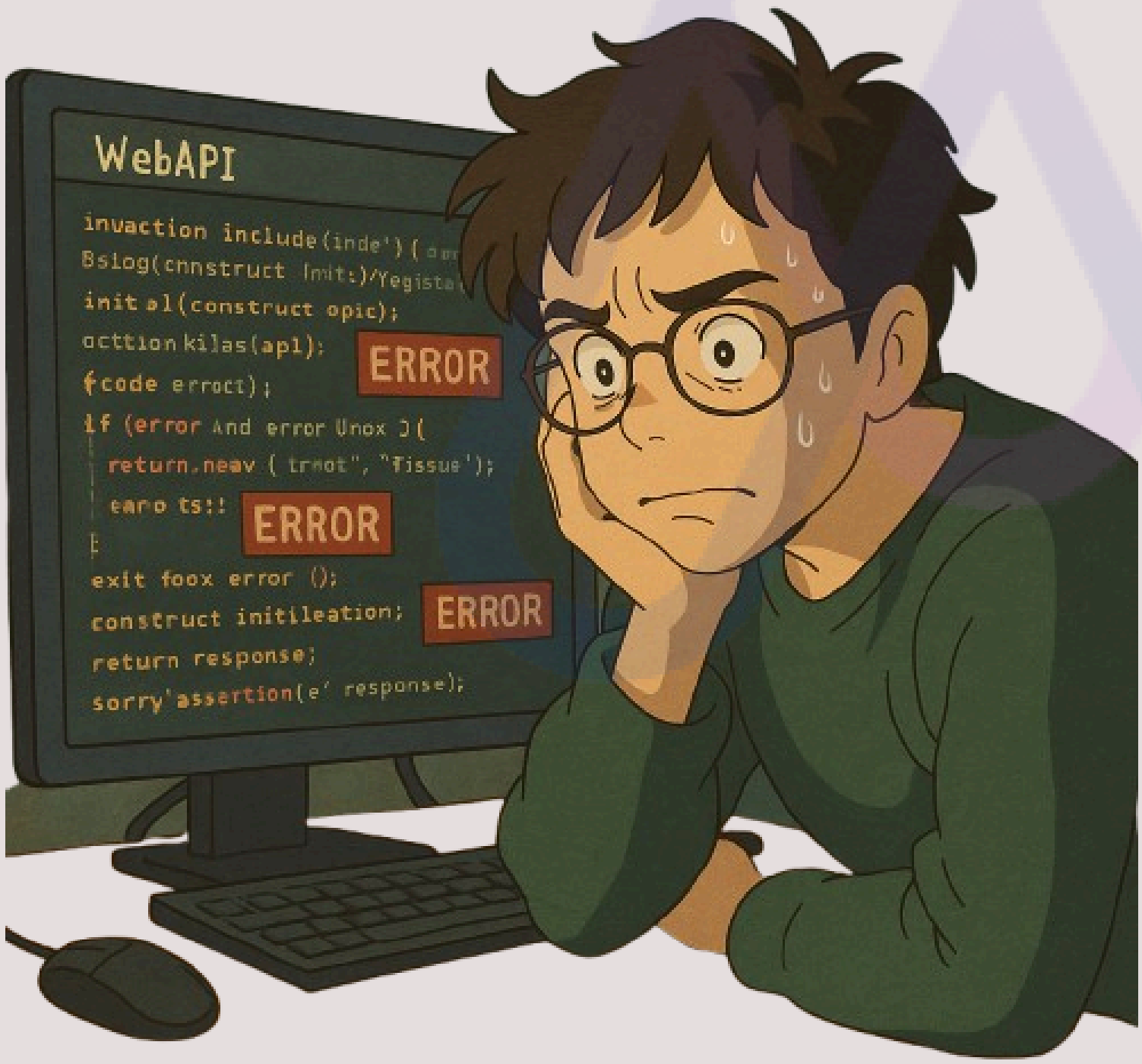
Ajay Patel

.NET

20 Mistakes You Makes When Creating Web APIs



Ajay Patel

1. Not Using HttpClientFactory

Creating HttpClient manually can lead to socket exhaustion, which causes APIs to become unresponsive under load.

❌ Bad Practice

```
using (HttpClient client = new HttpClient())
{
}

//Or
private static readonly HttpClient client = new HttpClient();
```

✅ Good Practice

```
//Register service
builder.Services.AddHttpClient();

// Use construction injection
public HttpClientFactoryService(IHttpClientFactory httpClientFactory)
{
    _httpClientFactory = httpClientFactory;
}

//Use and create new HttpClient
var httpClient = _httpClientFactory.CreateClient();
```

Ajay Patel

2. Not Using OpenTelemetry for Observability

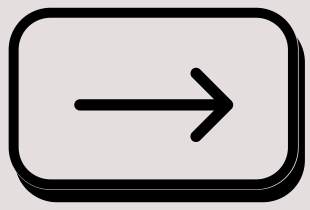
Monitoring only exceptions or logs isn't enough. Without proper observability, debugging production issues is like chasing ghosts.

❌ Bad Practice

```
//No trace context or metrics across services
_logger.LogInformation("Order created");
```

✅ Good Practice

```
builder.Services.AddOpenTelemetry()
    .WithTracing(builder ⇒
    {
        builder
            .AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddConsoleExporter();
    });
```



Ajay Patel

3. Not Implementing Rate Limiting

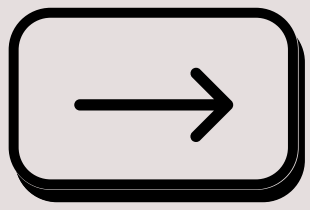
Without rate limiting, your API is exposed to abuse—malicious users or faulty clients can overwhelm it.

❌ Bad Practice

```
// No restriction on API usage per user
[HttpGet("resource")]
public IActionResult GetResource() { ... }
```

✅ Good Practice

```
builder.Services.AddRateLimiter(options => options
    .AddFixedWindowLimiter("fixed", config =>
    {
        config.PermitLimit = 100;
        config.Window = TimeSpan.FromMinutes(1);
    }));
```



Ajay Patel

4. Ignoring CORS (Cross-Origin Resource Sharing)

Blocking API requests across different origins? CORS configuration matters

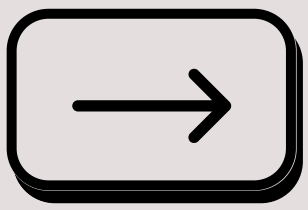
❌ Bad Practice

```
builder.Services.AddCors();
```

✅ Good Practice

```
services.AddCors(options ⇒
{
    options.AddPolicy("AllowSpecificOrigin",
        builder ⇒ builder.WithOrigins("https://myclient.com")
        .AllowAnyMethod()
        .AllowAnyHeader());
});
```

Restrict origins smartly—don’t just allow everything.



5. Not Enabling Response Compression

Sending large JSON payloads without compression leads to high bandwidth usage and slower responses.

✅ Good Practice

```
builder.Services.AddResponseCompression(options ⇒
{
    options.EnableForHttps = true;
    options.Providers.Add<GzipCompressionProvider>();
});
```

Ajay Patel

6. Ignoring API Versioning

APIs evolve—version them properly to avoid breaking changes

❌ Bad Practice

```
[Route("api/users")]
public class UsersController : ControllerBase
{

}
```

✅ Good Practice

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/users")]
public class UsersController : ControllerBase {}
```

Use URL-based or header-based versioning.

Ajay Patel

7. Not Using Global Exception Handler

Uncaught exceptions can lead to crashes, security vulnerabilities, and difficult debugging. APIs should always handle exceptions gracefully and provide meaningful responses.

❌ Bad Practice

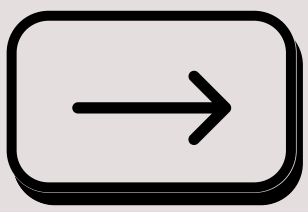
```
// No Global Exception Handler to handle it.
throw new Exception("Exception while fetching all the users.");
```

✅ Good Practice

```
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();
builder.Services.AddProblemDetails();

app.UseExceptionHandler();
```

Implement centralized exception handling to keep APIs robust and maintainable.



Ajay Patel

8. Forgetting to Cache Responses

Every request shouldn't hit the database—use caching.

❌ Bad Practice

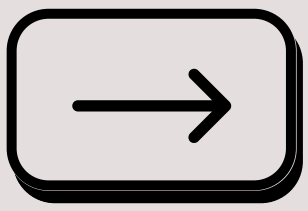
```
//No caching used. So, fetched from database.

return await _dbContext.Users.
    SingleOrDefaultAsync((user) ⇒ user.ProductId == id, token);
```

✅ Good Practice

```
builder.Services.AddHybridCache();

//apply Hybrid Cache
var user = await _cache.GetOrCreateAsync(
$"user-{id}",
async token ⇒
{
    return await _dbContext.Users
        .SingleOrDefaultAsync(user ⇒ user.ProductId == id, token);
}
```



Ajay Patel

9. Not Using Asynchronous Calls

Blocking operations can degrade API performance, causing slow responses and thread starvation. When handling I/O-bound operations, always prefer async/await.

❌ Bad Practice

```
public IActionResult GetUser(int id)
{
    var user = _dbContext.Users.Find(id);
    return Ok(user);
}
```

✅ Good Practice

```
public async Task<IActionResult> GetUser(int id)
{
    var user = await _dbContext.Users.FindAsync(id);
    return Ok(user);
}
```

Ajay Patel

10. Not Supporting Pagination in Large Responses

A massive dataset shouldn’t be dumped all at once—always paginate.

❌ Bad Practice

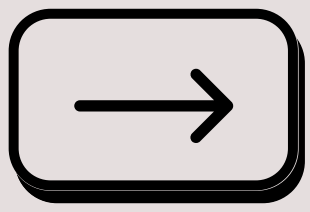
```
return Ok(await _dbContext.Users.ToListAsync());
```

✅ Good Practice

```
var result = await _dbContext.Users
                .Skip(page * size)
                .Take(size)
                .ToListAsync();

return Ok(result);
```

Improve performance with paging, filtering, and sorting



Ajay Patel

11. Exposing Sensitive Data in DTOs

Exposing sensitive fields like passwords, API keys, or payment details in DTOs can lead to serious security risks.

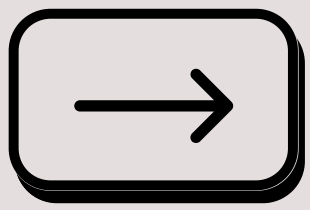
❌ Bad Practice

```
public class UserDto
{
    public string Username { get; set; }
    public string Email { get; set; }
    public string Password { get; set; } // 🚨
}
```

✅ Good Practice

```
public class UserDto
{
    public string Username { get; set; }
    public string Email { get; set; }
}
```

Apply DTOs properly by keeping them lean and secure.



Ajay Patel

12. Not Using Dependency Injection

Tightly coupled database calls in controllers, making testing impossible.

❌ Bad Practice

```
public class OrderService
{
    private PaymentService _paymentService = new PaymentService();

    public void ProcessOrder(Order order)
    {
        _paymentService.ProcessPayment(order);
    }
}
```

✅ Good Practice

```
public class OrderService
{
    private readonly IPaymentService _paymentService;

    public OrderService(IPaymentService paymentService)
    {
        _paymentService = paymentService;
    }

    public void ProcessOrder(Order order)
    {
        _paymentService.ProcessPayment(order);
    }
}
```

Ajay Patel

13. Not Documenting the API

A well-designed API is useless if developers can’t understand how to use it. Lack of documentation leads to confusion, wasted time, and frustration.

✅ Good Practice

```
builder.Services.AddSwaggerGen(c ⇒
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "Users API", Version = "v1" });
});
```

Always document APIs using OpenAPI (Swagger), Postman, etc.

Ajay Patel

14. Ignoring Proper HTTP Status Codes

Returning incorrect HTTP status codes leads to confusing client behavior and makes debugging harder. APIs should follow standard conventions for better clarity.

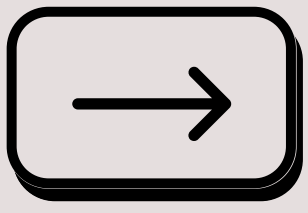
❌ Bad Practice

```
return Ok(new { error = "Invalid parameters." });
```

✅ Good Practice

```
return BadRequest(new { error = "Invalid parameters." });
```

Use 200 OK, 201 Created, 204 No Content, 400 Bad Requests, 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict, and 500 for Internal Server Errors.



Ajay Patel

15. Using GET for Actions That Modify Data

GET is safe and idempotent—it shouldn't change state.

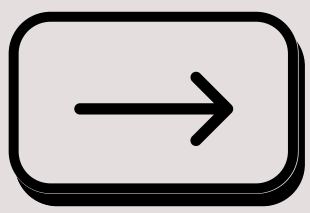
❌ Bad Practice

```
[HttpGet("api/users/{id}")]
public IActionResult DeleteUser(int id)
{
    /* Delete logic */
}
```

✅ Good Practice

```
[HttpDelete("api/users/{id}")]
public IActionResult DeleteUser(int id)
{
    /* Delete logic */
}
```

Use POST, PUT, or DELETE for modifications



Ajay Patel

16. Forgetting to Log API Calls

No logs = no way to debug issues.

❌ Bad Practice

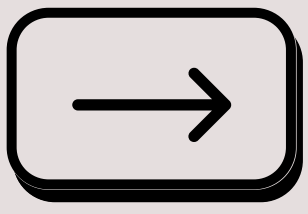
```
return BadRequest("Error Message");
```

✅ Good Practice

```
_logger.LogError("Error Message");

return BadRequest("Error Message");
```

Use structured logging with Serilog or Application Insights.



Ajay Patel

17. Overcomplicating Authentication & Authorization

API security should be simple yet robust. Don’t reinvent the wheel—use JWT, OAuth, or API Keys.

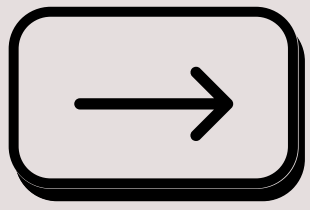
❌ Bad Practice

```
public bool IsAuthenticated(string username, string password)
{
   /* Custom auth logic */
}
```

✅ Good Practice

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options ⇒ { /* Configure JWT */ });
```

Leverage frameworks for security, not custom hacks.



Ajay Patel

18. Not Validating Input Data

Skipping validation leads to broken requests and potential security risks. Always validate incoming data.

❌ Bad Practice

```
public IActionResult CreateUser(UserDto user)
{
    await _userService.SaveAsync(user);
    return CreatedAtRoute(...);
}
```

✅ Good Practice

Use FluentValidation, DataAnnotation to ensure correctness.

```
public IActionResult CreateUser(UserDto user)
{
    var result = await _validator.ValidateAsync(user);

    if (!result.IsValid)
    {
        return BadRequest(result);
    }

    await _userService.SaveAsync(user);
    return CreatedAtRoute(...);
}
```

Ajay Patel

19. Poorly Designed Endpoint Naming

APIs should be predictable and logical.

❌ Bad Practice

```
GET /getUserDetails
POST /createNewUserEntry
DELETE /removeAccount
```

✅ Good Practice

```
GET /users/{id}
POST /users
DELETE /users/{id}
```

Follow RESTful conventions for clarity.

Ajay Patel

20. Hardcoding Configuration Value

Never embed secrets or configuration in code.

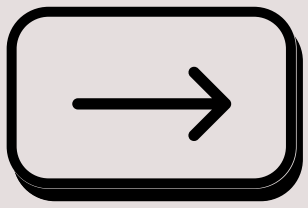
❌ Bad Practice

```
string connectionString = "Server=localhost;Database=mydb;User=myuser;Password=mypassword;";
```

✅ Good Practice

```
"ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=mydb;User=myuser;Password=mypassword;"
}
```

Use IConfiguration or Options pattern for flexible setups



Ajay Patel

.NET

Knowledge is contagious, let's spread it!

DO YOU LIKE THIS POST?

REPOST IT!

THANKS FOR READING