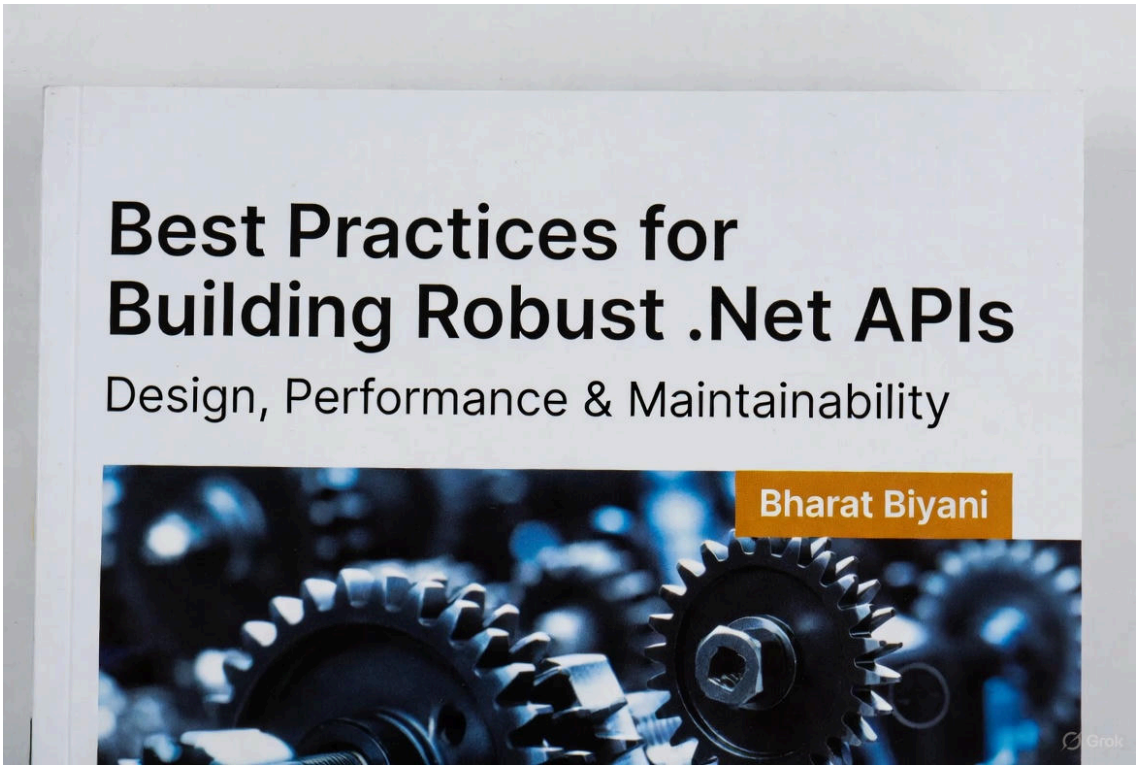
Best Practices and Patterns for Building Scalable .NET Backend APIs

By Bharat Biyani



Modern backend systems must be scalable, reliable, observable, and maintainable. With .NET's maturity and ecosystem, developers can build production-grade APIs efficiently—but following good architectural and operational practices is key.

1. **Architectural Patterns**

Clean Architecture (Onion):

- Philosophy: Business logic should be independent of frameworks, UI, and databases
- Dependency Rule: Dependencies point inward—domain knows nothing about other layers
- Testability: Domain logic can be tested without infrastructure
- Trade-off: Higher initial complexity, more abstractions

Vertical Slice Architecture:

- Philosophy: Organize around features rather than technical concerns

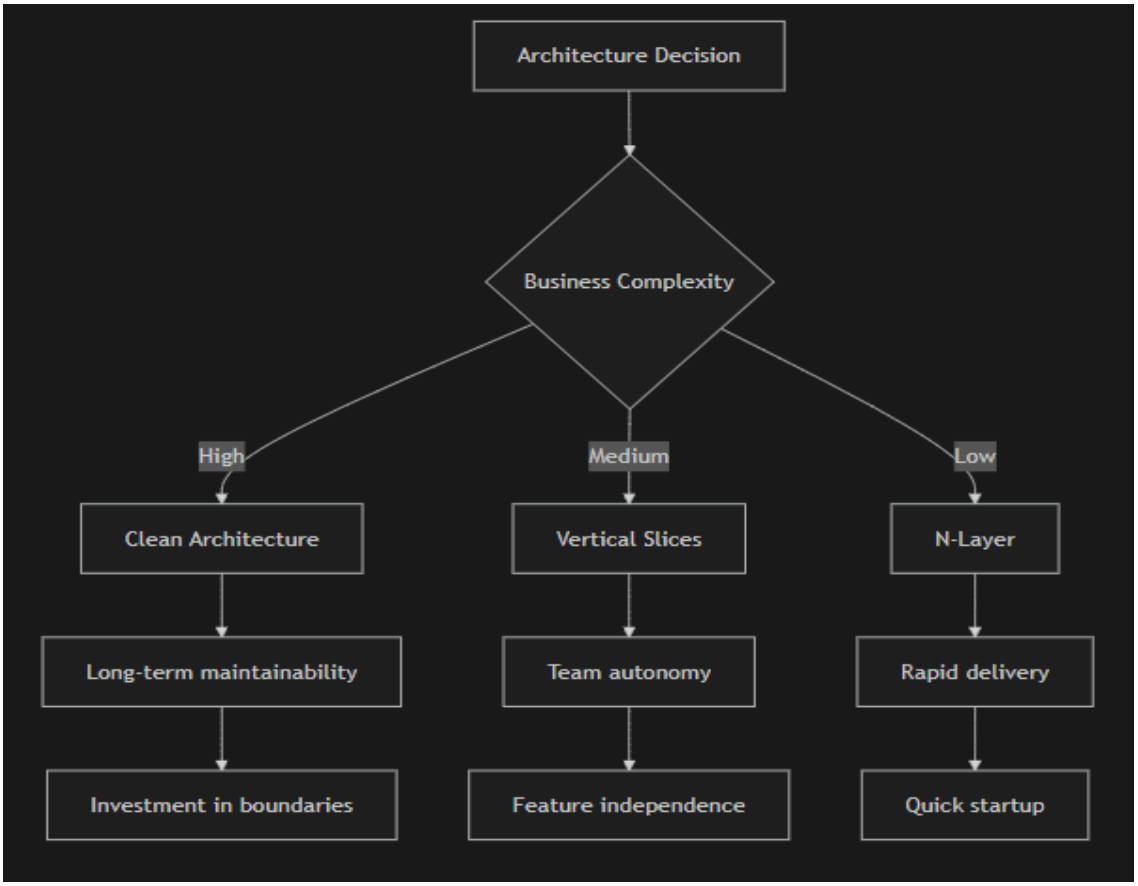
- Benefit: Features are independent, reducing coupling
- Drawback: Potential code duplication across features
- Ideal For: Feature teams, microservices, rapid evolution

Traditional N-Layer:

- Approach: UI → Business Logic → Data Access
- Advantage: Simple, familiar to most developers
- Risk: Business layer often gets coupled to data layer
- Suitable For: Simple CRUD applications with stable requirements

Strategic Recommendation:

- Start with Clean Architecture for complex domains
- Use Vertical Slices when you have independent feature teams
- Consider N-Layer only for simple, short-term projects



Clear Architecture (API / Application / Domain / Infrastructure)

Clean architecture (Layered architecture) promotes separation of concerns, allowing different parts of the system to evolve independently. This approach aligns with Clean Architecture principles, ensuring dependencies flow inward toward the domain core. It keeps business logic isolated from external concerns like databases or UI frameworks, making the system easier to test and adapt—for example, switching from SQL Server to PostgreSQL without changing core logic.

A common structure in .NET projects includes four layers:

- **MyApp.Api**: Handles presentation, such as HTTP endpoints or gRPC services.
- **MyApp.Application**: Manages use cases, workflows, and orchestration.
- **MyApp.Domain**: Encapsulates pure business logic and rules.
- **MyApp.Infrastructure**: Implements technical details like persistence and external integrations.

What Goes in Each Layer?

Domain Layer (MyApp.Domain)

Pure business logic. Avoid dependencies on HTTP, databases, or frameworks.

- Entities/Aggregates: e.g., Order, Customer.
- Value Objects: e.g., Money, EmailAddress.
- Domain Services: Operations spanning multiple entities.
- Domain Events: e.g., OrderPlaced.
- Domain Exceptions: e.g., BusinessRuleViolationException.

The domain should:

- Not reference EF Core, HTTP, or messaging libraries.
- Be unit-testable without infrastructure.

Application Layer (MyApp.Application)

Handles use cases and workflows. Coordinates domain logic with infrastructure via interfaces.

- Use Case Handlers: e.g., CreateOrderCommandHandler (using CQRS).
- DTOs/Commands/Queries.
- Interfaces for Infrastructure: e.g., IOrderRepository.
- Application Services: Alternative to CQRS.
- Mappings: Between DTOs and domain types.

References Domain; defines ports for Infrastructure.

Infrastructure Layer (MyApp.Infrastructure)

Implements technical details.

- EF Core DbContext and configurations.
- Repositories: Implementing interfaces.
- Messaging/External Clients.

References Application and Domain; implements ports.

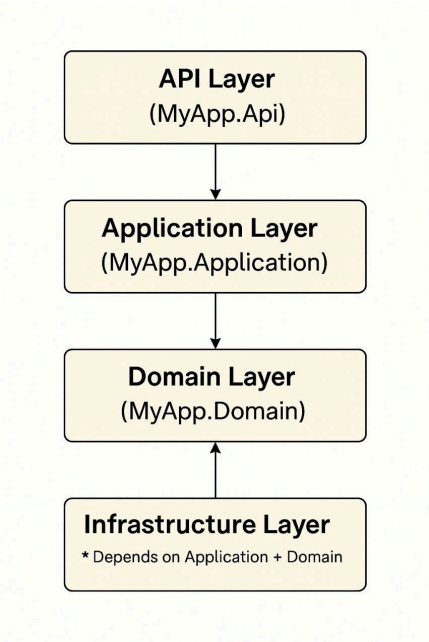
API Layer (MyApp.Api)

Exposes endpoints over HTTP or gRPC.

- Controllers/Minimal APIs.
- Filters/Middleware.
- Input Models.

References Application.

This layering ensures clear boundaries, testability, and flexibility (e.g., swapping databases). For visual representation, here's a **diagram** of Clean Architecture in .NET:



2. Dependency Injection (DI) in Practice

.NET's built-in DI container is robust for most scenarios, with .NET 9 adding better support for keyed services.

Lifetimes

- Transient: New instance per resolution. Ideal for stateless services.
- Scoped: Per request. Default for DbContexts.
- Singleton: App-wide. Use for thread-safe, shared resources.

Patterns & Practices

- Use constructor injection exclusively.
- Group registrations with extension methods.

3. Configuration & Secrets Management

Secure and flexible configuration is a cornerstone of.NET application.

Configuration Hierarchy

Follow the built-in layering system for clean overrides (lowest to highest priority):

1. appsettings.json – base configuration.
2. appsettings.{Environment}.json – environment-specific overrides.
3. Environment variables – for deployment/runtime configs.
4. User Secrets – for local dev (via dotnet user-secrets).
5. Secure vaults – for production secrets.

Access via IConfiguration and validate with strongly typed options at startup (IOptions<T>).

Secrets Management

Never hardcode or commit secrets.
For on-prem environments, consider these approaches:

- Development:
    - Use the built-in User Secrets feature (dotnet user-secrets).
    - Store temporary credentials in environment variables.
- Production:
    - Store secrets in an encrypted file (e.g., protected JSON).
    - Consider a self-hosted vault such as HashiCorp Vault, if you need central management and rotation.

4. Logging & Monitoring

Robust logging and monitoring are essential for diagnosing issues, ensuring stability, and maintaining observability in on-prem .NET applications.
Structured logs make it easy to query, correlate, and analyze events across distributed systems.

Structured Logging

Prefer structured logs over plain text for better parsing and filtering:

Log.Information("Order {OrderId} created for {CustomerId}", order.Id, order.CustomerId);

Recommended Setup

- Libraries: Serilog (most popular), or NLog for simpler setups.
- Sinks / Targets:
    - Elastic Stack (ELK) – mature and open-source.
    - Seq – lightweight and developer-friendly UI.
    - Loki + Grafana – efficient log aggregation with metrics integration.
    - Graylog – enterprise-grade on-prem log management.

Integrate health checks via AspNetCore.HealthChecks and expose /health endpoints for monitoring.

Cross-Service Correlation

For multi-service systems, generate and propagate correlation IDs via middleware:

- Add a unique ID per request (using middleware or libraries like CorrelationIdMiddleware).
- Pass the ID in headers (X-Correlation-ID) to track requests end-to-end.
- Enrich logs automatically with correlation context:
- Log.ForContext("CorrelationId", correlationId)
- .Information("Processing request...");

What to Log

Keep logs actionable and secure:

Level	Purpose	Example
Information	App lifecycle and business events	"User {UserId} logged in"
Warning	Unexpected but recoverable conditions	"Payment delay detected"

Level	Purpose	Example
Error	Failures that affect functionality	"Order creation failed"
Critical	System-wide or security-critical issues	"Database unreachable"

Avoid:

- Logging sensitive data (passwords, tokens, PII).
- Excessive debug logs in production.

5. Performance & Diagnostics

Performance tuning in .NET APIs isn't a one-time effort — it's an ongoing cycle of measuring, diagnosing, and optimizing.
Combine runtime tools, observability, and profiling to detect bottlenecks before users notice them.

Profiling & Diagnostics Tools

Use built-in .NET CLI tools when performance degrades or anomalies appear:

Tool	Purpose	When to Use
dotnet-counters	Live metrics (GC, threads, requests/sec)	Monitor during load tests or production spikes
dotnet-trace	CPU/memory profiling	When requests slow down or CPU is high
dotnet-gcdump	Analyze GC & allocations	When memory keeps growing or leaks suspected
dotnet-dump	Inspect threads & stacks	When app hangs or stops responding

How: Run dotnet-counters monitor -p <PID> or dotnet-trace collect -p <PID> on a running process, then analyze in Visual Studio, PerfView, or WinDbg.

Observability Stack

Adopt OpenTelemetry (OTel) for unified metrics, traces, and logs.
Export to on-prem tools like Prometheus + Grafana (metrics) and Jaeger/Zipkin (traces).

Setup Example (Program.cs):

```
builder.Services.AddOpenTelemetry()
   .WithMetrics(m => m.AddAspNetCoreInstrumentation().AddPrometheusExporter())
```

```
    .WithTracing(t => t.AddAspNetCoreInstrumentation().AddHttpClientInstrumentation());
```

Expose /metrics for Prometheus scraping and visualize latency, error rates, and throughput in Grafana.

Continuous Performance Process

Follow a repeatable process:

1. Baseline: Run load tests (e.g., with k6) to establish normal response times.
2. Measure: Use metrics and traces to detect slow endpoints or heavy allocations.
3. Profile: Collect runtime data with dotnet-trace or Visual Studio Profiler.
4. Optimize: Fix async blocking, caching, query inefficiencies, or GC pressure.
5. Re-test: Confirm improvements under the same load.

When Logs Are Enough — and When They Aren't

- Use logs for request flow and exception context.
- Use metrics/traces to spot performance trends.
- Use dumps and profilers when memory leaks, hangs, or unexplained slowdowns occur.

6. Messaging & Asynchronous Processing

Asynchronous messaging improves scalability, reliability, and responsiveness in .NET APIs by decoupling long-running or cross-service work from request handling.

When to Use a Message Broker

Use a broker when:

- Operations are time-consuming or I/O-heavy (emails, file generation, external APIs).
- You need durability and retries so work isn't lost if the API restarts.
- Multiple services must react to the same event (e.g., OrderCreated).
- You want to smooth traffic spikes via background processing.

If work is quick, isolated, and non-critical, keep it inline — simplicity wins.

Broker Options

Broker	Strengths	Best For
RabbitMQ	Reliable, easy queues, routing	Work queues, command/event bus
Kafka	High throughput, replayable log	Event sourcing, streaming data
NATS	Lightweight pub/sub	Real-time notifications, control events

For .NET, use frameworks like MassTransit or Rebus to abstract brokers and manage retries, dead-letter queues, and serialization.

Simplified Messaging Flow

1. API receives request.
2. Validates input and publishes a message to the broker.
3. Returns 202 Accepted (or 200) after enqueue succeeds.
4. A background service or separate worker processes messages asynchronously.

This approach keeps the API responsive and avoids blocking users while background workers handle heavy or slow operations.

Outbox Pattern — When and Why

Only use an outbox when API updates the database and publishes a message that must stay consistent.

When to use:

- Both DB write and message publish are part of a critical transaction (e.g., order + payment event).
- You can't afford message loss or duplicates.

How it works (conceptually):

- Write domain entity and an OutboxMessage record in the same DB transaction.
- A BackgroundService later reads unsent outbox records and publishes them to the broker.
- Mark them as sent once delivery succeeds.

If the API only enqueues work and doesn't commit DB changes first, you can skip the outbox — the broker itself provides reliability.

Internal Asynchronous Work

Not all async work requires a broker. For lightweight background tasks (e.g., cache refresh, cleanup):

- Use BackgroundService or IHostedService.
- Use System.Threading.Channels for in-memory queuing.
  Note: This is non-durable and resets on restart — suitable only for transient workloads.

Delegates, Events, and Messaging

C# events and delegates provide in-process decoupling — great for triggering actions *inside* a single service.
Message brokers extend this concept across processes and machines with durability and reliability.

7. Long-Running Jobs & Scheduling

Long-running or scheduled work (reports, invoices, cleanups, syncs) shouldn't block API requests. Offload them to background jobs with persistence, retries, and visibility.

When to Use a Job Scheduler

Use a scheduler like Hangfire or Quartz.NET when:

- Work takes seconds/minutes (reports, exports, batch processing).
- You need retries, persistence, and a dashboard to see job status.
- Jobs must run on a schedule (hourly, daily, cron-based).
- You want jobs to survive app restarts and machine reboots.

If work is very short-lived and only tied to the incoming HTTP request, a simple BackgroundService or queue may be enough.

Hangfire – Fire-and-Forget + Scheduled Jobs

Hangfire is a common, on-prem friendly choice with a dashboard and persistent storage (SQL Server, PostgreSQL, etc.).

Make Jobs Safe: Idempotency, Retries, and Timeouts

Because background jobs will be retried, they must be idempotent:

- Use natural keys: e.g., don't create a new invoice if Invoice for OrderId already exists.
- Record job execution status (e.g., JobRuns table) if needed.
- Wrap external calls with resilience (retries, circuit breakers via Polly).

Set retry rules in Hangfire/Quartz instead of hand-rolling loops inside the job.

Choosing Between Jobs, Queues, and Simple Background Services

- Hangfire / Quartz.NET
    - Use for durable, visible, scheduled or long-running work.
- Message broker + consumer
    - Use when work is part of a distributed system, or multiple services react to events.
- BackgroundService / IHostedService
    - Use for simple internal recurring tasks (cache warm-up, small housekeeping) that don't need dashboards or persistence.

8. Caching & State Management

Caching improves performance and reduces load on databases or APIs.
For .NET APIs, use a cache-aside strategy and prefer distributed caches (like Redis) for scalability across multiple instances.

When to Use Caching

Apply caching when:

- The data is expensive to compute or fetch.
- The same data is requested frequently.
- Slightly stale data is acceptable.

Avoid caching:

- Highly dynamic or security-sensitive data.
- User-specific or transactional data (unless carefully scoped).
-

Cache-Aside Pattern (How It Works)

The API first checks the cache; if data is missing, it loads from the database and stores it for next time.

Distributed Caching Options

Cache Type	Strengths	Best For
Redis	Fast, distributed, supports pub/sub invalidation	Shared cache across servers
NCache	.NET native, on-prem friendly	Enterprise, offline environments

Cache Type	Strengths	Best For
MemoryCache	Simple, in-process	Single-instance, dev/test workloads

Use IDistributedCache abstraction to swap backends without code changes.

State Management Principles

- APIs should remain stateless — no per-user data in memory.
- Use cache for shared data, not session state.
- For session-like scenarios (e.g., shopping carts), prefer:
    - Redis or SQL-backed session store (AddDistributedRedisCache + AddSession).
    - Avoid sticky sessions — keep the app horizontally scalable.

9. Testing & Quality

Quality in .NET APIs comes from targeted testing.
Follow a testing pyramid: wide unit coverage, focused integration tests, and minimal end-to-end checks. Each layer serves a distinct purpose.

Testing Strategy Overview

Test Level	Purpose	When to Apply	Recommended Tools / Patterns
Unit Tests	Validate isolated logic (calculations, rules, validations) quickly and deterministically.	Core business logic, domain models, helper libraries.	xUnit / NUnit / MSTest with Moq or NSubstitute for mocking. Use FluentAssertions for readability.
Integration Tests	Verify full request-to-response flow — DI, controllers, filters, middleware, persistence.	Key API endpoints, data access, authentication, core workflows.	WebApplicationFactory, EF Core test DBs (InMemory, SQLite, or Testcontainers), WireMock.NET or MockHttp for external calls.
Contract / Component Tests	Ensure API's request/response contracts or message schemas remain stable.	Public APIs or message queues consumed by other teams/systems.	Pact.NET, schema validation, snapshot tests, or BDD scenarios.
End-to-End (E2E)	Validate the system as a user sees it — across API, DB, and integrations.	Few representative flows (e.g., "Place Order", "Complete Payment").	Postman/Newman, Playwright, Cypress, or SpecFlow (BDD scenarios for business flows).

Test Level	Purpose	When to Apply	Recommended Tools / Patterns
Performance / Load Tests	Detect latency regressions and throughput degradation.	Critical endpoints or workloads before release or infrastructure changes.	k6, JMeter, dotnet-counters, dotnet-trace.

BDD & SpecFlow — Where It Fits

Behavior-Driven Development (BDD) tools like SpecFlow sit at the intersection of integration and E2E layers.
Use them when:

- Business rules are complex and benefit from plain-language scenarios shared with non-technical stakeholders.
- You want automated acceptance criteria that align with user stories.
- The same steps can be reused across integration and end-to-end tests.

Example: Given an order is placed → When payment succeeds → Then invoice is generated.

BDD doesn't replace unit or integration tests — it complements them for business-critical flows.

Making Tests Predictable

Predictable tests yield the same result every run — locally or in CI — by controlling environment, data, and time.

Key practices:

- Isolated Data: Use disposable databases (InMemory, SQLite, or ephemeral test DBs). For complex schemas, spin up real DB containers or use a dedicated test DB reset before each run.
- Controlled Time: Abstract time (via IClock or similar) so tests don't depend on the current date/time.
- Fakes for External Services: Use mocks or local stubs (e.g., WireMock.NET) for HTTP and message-based integrations.
- Stable Test Data: Seed only what's required per test, not the full production dataset.

Running Tests in CI/CD (Headless & Automated)

Integration and higher-level tests should run fully automated, with no manual setup or external dependencies.

Use:

- Ephemeral or isolated test environments — start API and test database automatically (Testcontainers or Docker).
- Automated migrations and seeding on startup.
- Self-contained test scripts — dotnet test runs everything locally or in CI without manual intervention.

10. CI/CD & DevOps

Reliable CI/CD pipelines are essential for consistent quality, fast delivery, and operational stability.
Core Principles

1. Automate Everything: Builds, tests, and deployments should be reproducible via scripts — no manual steps.
2. Safe Deployments: Deploy gradually and reversibly to minimize downtime and risk.
3. Traceability: Every build should be traceable to source code, tests, and deployment logs.

CI/CD Tooling

Common Choices:

- GitHub Actions / GitLab CI: Cloud-based and YAML-driven; easy to version and share.
- Jenkins / Azure DevOps / TeamCity: On-prem or hybrid setups with strong enterprise integration.
- Docker: Standardize build and runtime environments across machines.

Typical CI pipeline:

1. Checkout and restore dependencies.
2. Build with dotnet build --configuration Release.
3. Run automated tests (dotnet test).
4. Package into a container (docker build).
5. Publish artifact or push image to registry.

Typical CD pipeline:

1. Pull tested image or artifact.
2. Deploy to staging → run smoke tests.

3. Promote to production upon validation.
4. Roll back automatically if health checks fail.

Deployment Hygiene & Observability

- Health checks: Always expose /health endpoints for deployment validation.
- Smoke tests: Run quick sanity checks right after deployment.
- Rollback policy: Automate reversion to the last stable version if startup or health checks fail.
- Monitoring hooks: Integrate build/deploy events with observability stack (Grafana, Prometheus, ELK).

Continuous Feedback Loop

CI/CD isn't just automation — it's an engineering feedback cycle:

- CI gives early feedback on code and tests.
- CD gives feedback on deployment stability and runtime metrics.
- Observability closes the loop by showing the impact of each release.

11. Security Best Practices (General .NET, Not Cloud-Dependent)

Security should be baked into .NET API design from day one — not added on later. Focus on strong authentication, least-privilege authorization, defensive coding, and secure-by-default configuration.

Authentication & Authorization

- Use standardized protocols: Prefer OpenID Connect + OAuth2 with JWT access tokens.
- On-prem / vendor-neutral options:
    - Keycloak (self-hosted identity provider)
    - Enterprise identity servers (e.g., IdentityServer, custom IdP)
- Keep the API focused on resource protection, not user management:
    - Validate JWTs in the API.
    - Enforce authorization with policies and roles/claims (e.g., policy-based auth).

Principles:

- Apply least privilege: only grant the claims/roles required.
- Prefer policy-based authorization over ad-hoc if (User.IsInRole(...)) scattered in code.

Rate Limiting & Abuse Protection

- Enable built-in rate limiting in ASP.NET Core (middleware-based) to:
    - Throttle abusive clients.
    - Protect against brute-force and resource exhaustion.
- Use different policies for:
    - Public vs internal endpoints
    - Authenticated vs anonymous users
- Combine with:
    - IP allowlists/denylists where appropriate.
    - Account lockout / backoff on repeated failed logins (implemented at the IdP).

Transport & Data Protection

- Enforce HTTPS everywhere:
    - Redirect HTTP → HTTPS.
    - Use strong TLS versions and ciphers (configured at reverse proxy/web server).
- Never send sensitive data (tokens, passwords, secrets) over plain HTTP.
- Use HSTS, secure cookies, and modern security headers (via reverse proxy or middleware).

Input Validation & Defensive Coding

- Validate all inputs:
    - Use model validation attributes and custom validators where needed.
    - Enforce length limits, allowed values, and formats.
- Treat all external input as untrusted:
    - Avoid string concatenation in SQL; use parameterized queries / EF Core.
    - Sanitize or encode any content that might end up in logs, headers, or HTML.
- Avoid exposing internal details:
    - Generic error responses to clients.
    - Detailed exception info only in logs, not in API responses.

12. Documentation & Developer Experience

Invest in documentation so other developers can get productive quickly and consumers can integrate API without guesswork.

API Documentation (OpenAPI / Swagger)

- Expose a Swagger/OpenAPI document for every API:
    - Use Swashbuckle to generate OpenAPI from controllers and models.
    - Enable Swagger UI in non-production environments (or protect it in prod).
- Enrich the spec:
    - Add XML comments on controllers and models for descriptions and examples.
    - Document status codes, error shapes, and required/optional fields.
    - Tag endpoints by area (Orders, Customers, Admin) for easier navigation.

Swagger/OpenAPI becomes the single source of truth for:

- REST contract (paths, methods, types).
- Client generation (C#, TypeScript, etc.).
- Other tools (API gateways, contract tests, BFFs).

Versioning & Change Management

Versioning is about breaking-change safety.

- Pick one approach and stick to it:
    - URL-based: /api/v1/orders, /api/v2/orders — simplest and most explicit.
    - Header-based: version via custom or Accept header — cleaner URLs but more complex.
- Document:
    - Which versions are supported, deprecated, and retirement timelines.
    - Differences between versions at a high level.
- Avoid breaking changes within a version; add new fields and endpoints in a backward-compatible way when possible.

In short:

Make API self-documenting via OpenAPI, easy to run locally, and easy to consume via examples and clear versioning — that's what turns a correct .NET API into a great developer experience.

👤 About the Author

Bharat Biyani - Senior .NET Developer specializing in scalable backend systems, API architecture, and software design.