[4-Make a linkedIn article about CORS policy](https://www.linkedin.com/posts/ahmed-b-ramzy_dotnetcore-webapi-corspolicy-activity-7341919434531356673--3kB?utm_source=share&utm_medium=member_desktop&rcm=ACoAADuIxXkBdVPXITPJTt0yTVu1y8HgMAABtYE)

**5-mention 3 cases to use GraphQL,gRPC ?**

**When to Use GraphQL:**

1. Client-Driven Data Needs (Frontend Flexibility)
   * Scenario: A mobile app for e-commerce (like Jumia or Amazon) needs to fetch only product name, image, and price to save bandwidth.
   * Why GraphQL: It allows the client to request only the fields it needs from the server, reducing payload size.
2. Aggregating Data from Multiple Sources
   * Scenario: A dashboard application needs user profile from one microservice, orders from another, and preferences from a third.
   * Why GraphQL: GraphQL can combine responses from multiple services into a single query result.
3. Evolving Frontends without Breaking APIs
   * Scenario: A React app is being redesigned, and new UIs need new fields, while older versions of the app still exist.
   * Why GraphQL: You can extend your GraphQL schema without versioning or breaking existing queries.

**When to Use gRPC:**

1. High-Performance Internal Service-to-Service Communication
   * Scenario: A ride-sharing backend system (like Uber) has multiple microservices (e.g., pricing, geolocation, driver tracking).
   * Why gRPC: It uses HTTP/2 and binary Protobuf, which makes it ultra-fast and efficient between internal services.
2. Low Latency, Real-time Streaming Needs
   * Scenario: A chat app backend using bi-directional streaming between users and the server.
   * Why gRPC: gRPC supports streaming out-of-the-box with better performance than REST or GraphQL.
3. Strongly Typed APIs with Contract-First Development
   * Scenario: FinTech company with strict requirements and code generation for multiple languages.
   * Why gRPC: Protobuf IDL (interface definition language) ensures strict contracts and generates code for multiple platforms (Java, Go, C#, etc.)

**6-what api versioning? How to implement this practice?**

**What is API Versioning?**

API versioning is the practice of managing changes to your API without breaking existing clients. When you change the structure or behavior of the API, you release a new version.

**Why is it Important?**

* Avoid breaking existing frontend/mobile apps
* Manage gradual feature rollouts
* Maintain multiple consumer support (e.g., different client apps)

**How to Implement API Versioning?**

**1. URI Versioning (Most Common)**

* URL: /api/v1/products
* Easy to understand and cache
* **Used in ASP.NET Core by**: [Route("api/v1/[controller]")]

**2. Header Versioning**

* Send version in HTTP header: api-version: 1.0
* Cleaner URLs, but harder to debug in browser

**3. Query String Versioning**

* Example: /api/products?version=1.0
* Easy to test, not preferred for production

**4. Accept Header (Media Type Versioning)**

* Accept: application/vnd.myapp.v1+json
* Clean and RESTful, but harder to configure

**7-Is there any difference between onion and clean architecture ?!**

**So tell when use each in 3 cases mentioning business scenarios.**

there is a difference between Onion Architecture and Clean Architecture, even though they seem very similar at first glance. Both aim to separate concerns, make the code testable, and ensure the core business logic is independent from external concerns (like databases, UI, frameworks). But their focus and philosophy differ slightly.

**First, the core difference between Onion and Clean Architecture:**

**Onion Architecture:**

* Introduced by Jeffrey Palermo.
* Focuses on layering around the domain.
* The domain model (entities) is at the center.
* It’s data-centric: the layers are organized based on how data flows through the system.
* Application services orchestrate business logic around domain entities.
* The outer layers (like infrastructure, UI) depend on the inner ones, never the other way around.

**Clean Architecture:**

* Introduced by Uncle Bob (Robert C. Martin).
* Focuses on use cases and behavior as the core.
* The use cases are the central unit of organization.
* It’s use-case-centric: instead of thinking in terms of entities and data, it organizes the system around user actions and business rules.
* It makes a clearer distinction between application rules and business rules.

**When to Use Each – 3 Business Scenarios Explained:**

**Financial or Banking System – Complex Business Rules**

* In a banking system, you deal with things like money transfers, fraud detection, account rules, regulatory compliance, etc.
* These rules are not just CRUD; they involve lots of behaviors, use cases, validations, and workflows.
* You don’t want your business rules to be affected by the database or the web framework.

**Use Clean Architecture** here because it **places use cases at the center**. That makes it easier to isolate, test, and evolve complex business processes.  
Also, it enforces a clear boundary between business rules and technical details.

**E-commerce System – Many Entities and CRUD Operations**

* In an e-commerce platform, you usually deal with Products, Categories, Orders, Customers, Reviews, etc.
* The logic revolves around creating, updating, retrieving, and deleting data (with some business rules, but not extremely complex).
* The domain (entities and relationships) is central to how the app works.

Use Onion Architecture here because the domain model is rich, and you want to keep infrastructure concerns (like database or APIs) out of the core.  
The layering helps you protect the domain while still handling real-world operations efficiently.

**Startup or MVP Project – Quick Launch, Iterative Development**

* Suppose you’re building a new social media app or a food delivery platform.
* The goal is to launch quickly, validate the idea, and iterate fast.
* Business logic is not too complex yet, and you might rewrite or refactor after validation.

Use Onion Architecture (or even a simplified version) in this case.  
It gives you a clean structure without overengineering and lets you keep your domain clean and infrastructure swappable.  
Clean Architecture can be added later as the system becomes more behavior-driven and complex.