CORS, Authentication, Authorization, JWT

📘 CORS (Cross-Origin Resource Sharing)

What is CORS?

CORS is a **browser security mechanism** that controls whether a web application running on one origin (domain + port + protocol) can access resources from another origin.

Why CORS exists

Browsers enforce the **Same-Origin Policy** to prevent malicious websites from reading sensitive data from another site without permission.

What is an Origin?

An origin consists of:

- Protocol (http / https)
- Domain (localhost / example.com)
- Port (3000 / 5000)

If **any one differs**, it is a cross-origin request.

When does CORS apply?

- ✔ Browser → API (AJAX / Fetch / Axios)
- ❌ Server → Server (CORS does NOT apply)
- ❌ Postman / curl (CORS does NOT apply)

How CORS works (high level)

1. Browser sends request with `Origin` header
2. Server responds with CORS headers
3. Browser allows or blocks the response

Common CORS headers

- `Access-Control-Allow-Origin`
- `Access-Control-Allow-Methods`
- `Access-Control-Allow-Headers`
- `Access-Control-Allow-Credentials`

CORS in ASP.NET Web API (.NET Framework)

CORS must be **explicitly enabled**.

```
config.EnableCors();
```

Enable globally:

```
var cors = new EnableCorsAttribute("*", "*", "*");
config.EnableCors(cors);
```

Enable per controller:

```
[EnableCors("http://localhost:3000", "*", "*")]
```

Key Points

- CORS is enforced by **browser**, not server
- Same origin → no CORS required
- Different origin → CORS must be enabled
- Used mainly with SPA frontends (React, Angular)

🔐 Authentication

What is Authentication?

Authentication answers the question:

> **“Who are you?”**

It verifies the **identity** of the user or client.

Authentication in Web APIs

In APIs, authentication is usually **token-based**, not session-based.

Common methods:

- Cookies (traditional MVC apps)
- JWT Bearer Tokens (modern APIs)

Authentication flow (JWT-based)

1. User sends credentials (login)

2. Server verifies credentials
3. Server issues a JWT
4. Client stores JWT
5. Client sends JWT with every request

Authentication in ASP.NET (.NET Framework)

- Implemented using **OWIN middleware**
- Happens **before controller execution**

`app.UseJwtBearerAuthentication(...)`

`[Authorize]` attribute

`[Authorize]`

- Allows only **authenticated users**
- If token is missing or invalid → `401 Unauthorized`

Key Points

- Authentication verifies identity
- Happens once per request
- JWT replaces server-side sessions
- Stateless (server does not remember users)

🛂 Authorization

What is Authorization?

Authorization answers the question:

> **“What are you allowed to do?”**

It checks **permissions** after authentication.

Difference between Authentication & Authorization

Authentication	Authorization
Who are you?	What can you access?
Login step	Permission step
401 if failed	403 if failed

Authorization in ASP.NET Web API

Basic:

```
[Authorize]
```

Role-based:

```
[Authorize(Roles = "Admin")]
```

How role authorization works

- Role is stored as a **claim**
- Claim type: `ClaimTypes.Role`
- ASP.NET checks:

```
User.IsInRole("Admin")
```

Authorization failure responses

- Not authenticated → `401 Unauthorized`
- Authenticated but forbidden → `403 Forbidden`

Key Points

- Authorization depends on authentication
- Uses claims (especially role claims)
- Enforced before controller logic runs

🔑 JWT (JSON Web Token)

What is JWT?

JWT is a **self-contained, digitally signed token** used to securely transmit user identity and claims between client and server.

JWT structure

A JWT has **three parts**:

```
HEADER.PAYLOAD.SIGNATURE
```

JWT Header

Contains metadata:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

JWT Payload

Contains claims:

```
{
  "name": "admin",
  "role": "Admin",
  "iss": "DemoIssuer",
  "aud": "DemoAudience",
  "exp": 1700000000
}
```

JWT Signature

- Created using a secret key
- Ensures token integrity
- Prevents tampering

Why JWT is encoded, not encrypted

- JWT is **Base64 encoded**, not encrypted
- Anyone can read payload
- Security comes from **signature**, not secrecy
- Sensitive data should NOT be stored in JWT

JWT in ASP.NET (.NET Framework)

JWT is validated using **OWIN middleware**:

```
app.UseJwtBearerAuthentication(...)
```

Validation includes:

- Signature validation
- Expiry validation
- Issuer validation
- Audience validation

TokenValidationParameters (important)

```
ValidateIssuerSigningKey = true
ValidateLifetime = true
IssuerSigningKey = secretKey
```

Ensures:

- Token is genuine
- Token is not expired
- Token was issued by your server

Claims in JWT

Claims are key-value pairs describing the user.

Common claims:

- `ClaimTypes.Name` → username
- `ClaimTypes.Role` → role
- Custom claims → app-specific data

Where JWT is checked

- JWT is validated **on every request**
- Happens before controller execution
- Implemented by OWIN middleware

JWT Request Flow

```
Client
 → Authorization: Bearer TOKEN
 → OWIN JWT Middleware
 → Token Validation
 → ClaimsIdentity created
 → [Authorize] check
 → Controller executes
```

Exception handling in ASP.NET Web API:

1) Definition:
- Exceptions are runtime errors that disrupt normal program flow.
- In Web APIs, exceptions must be handled so that appropriate HTTP responses are returned.

2) Basic local handling:
- Use try-catch blocks within code for specific error cases.
- Throw HttpResponseException for controlled HTTP responses.

3) Web API exception filters:
- Derive from ExceptionFilterAttribute and override OnException.
- Can be applied per controller or globally.
- Useful for handling controller-level exceptions.

4) Global error handling:
- Catch all unhandled exceptions with a global exception handler.
- Implement by replacing IExceptionHandler in WebApiConfig.
- Provides centralized responses and logging.

5) Demonstration:
- Create GlobalErrorHandler class inheriting ExceptionHandler.
- Override Handle to return custom error structure.
- Register in WebApiConfig with config.Services.Replace.

6) Best practices:
- Avoid repetitive try-catch blocks.
- Use global handlers and filters.
- Return meaningful HTTP status codes.
- Do not expose sensitive details in responses.
- Log errors for diagnostics.

7) Advanced:
- Define custom exception types for business logic.
- Use structured error responses (ProblemDetails format).

Exception Filters vs Global Exception Handler in ASP.NET Web API:

1) Exception Filters:
- Implemented by deriving from ExceptionFilterAttribute.
- Can be applied at action or controller level.

- Only catch exceptions thrown during controller action execution.
- Registered by decorating controllers/actions or via config.Filters.Add().

2) Global Exception Handler:
- Implements ExceptionHandler (IExceptionHandler).
- Replaces default handler via config.Services.Replace().
- Catches all unhandled exceptions in the entire Web API pipeline, including before/after controller actions.
- Useful for uniform API error responses and logging.

3) Execution order:
Controller action throws → Action exception filters → Global filters → Global Exception Handler.
If any filter handles the exception (sets context.Response), global handler is skipped.

📘 CACHING STRATEGIES IN APIs

🔎 What Is Caching?

Caching is the process of storing copies of frequently accessed data in a temporary storage (cache) so future requests can be served more quickly without recomputing or refetching from the original source. Caching improves performance, scalability, and reduces server load.

In REST APIs, caching can occur at various points: at the client, at intermediaries (like CDNs or proxies), or on the server.

🧠 WHY CACHING MATTERS

Benefits of caching in APIs:

✔ **Improved performance:** Cached responses are faster than recomputing data.
✔ **Reduced server load:** Less frequent database access.
✔ **Better scalability:** Can handle more traffic with fewer resources.
✔ **Lower latency:** Faster response times for end users.

📍 LEVELS OF CACHING

Caching can be broadly categorized into:

1️⃣ CLIENT-SIDE CACHING

- Cache data **in the consumer's browser or app**.
- Uses HTTP headers like `Cache-Control`, `ETag`, `Expires` to tell browsers how long to cache a response.
- Reduces requests to the server.

Use Cases:

- Static content (CSS, images, product lists).
- Responses that don't frequently change.

2️⃣ SERVER-SIDE CACHING

- Server stores computed results (or database query results) in memory or an external cache (Redis, Memcached).
- Good for heavy or expensive computations.

Types:

- In-memory cache — stored in RAM, fastest access.
- Distributed cache — shared cache across multiple servers (Redis, Memcached).

3️⃣ PROXY / CDN CACHING

- Intermediate caches like CDNs (Cloudflare, Akamai) serve cached responses to multiple clients.
- Reduces load on origin servers.

🔧 HTTP CACHING (BUILT-IN PROTOCOL CACHE)

The HTTP protocol itself supports caching using headers like:

- `Cache-Control`
- `Expires`
- `ETag` (entity tag for validation)

These headers instruct:

- **Who can cache**
- **How long to cache**
- **When to revalidate**

Example:

```
Cache-Control: public, max-age=60
ETag: "xyz123"
```

🔍 CACHE VALIDATION & CONDITIONAL REQUESTS

With `ETag`:

- Client sends `If-None-Match: <etag>` in the next request.
- Server compares and returns:
  - `304 Not Modified` (if unchanged)
  - Full response (if changed)

This avoids transferring full data again.

🧠 CACHE INVALIDATION

Key challenge: ensuring data freshness.

Common strategies:

- **TTL (time-to-live):** Cache expires after a set time.
- **Event invalidation:** Clear cache when source data changes.
- **Version keys:** Change cache keys on updates.

🧠 CHOOSING CACHE SCOPE

Decide based on requirements:

Scope	Good for	Example
Client	Static, repeat views	images, site assets
Server	Heavy computations	user dashboards
Proxy/CDN	Public resources	global products list

📌 COMMON CACHING ISSUES

Cache stampede
 Occurs when many requests occur just after a cache entry expires, causing simultaneous recompute. Strategies like *early refresh* or *locking* can help mitigate this.

CACHE VIDEO : https://www.youtube.com/watch?v=TV-xsNjbx_g

📘 Versioning Techniques In APIs

API Versioning: concepts, strategies, pros & cons

1 — What is API versioning and why it matters

API versioning is the practice of assigning version identifiers to an API so you can change the API (add, remove, or change behavior) without breaking existing clients. Without versioning, any breaking change can instantly invalidate client integrations.

Why version:

- Preserve backward compatibility for existing clients.
- Allow evolution (new fields, changed semantics, new resources) safely.
- Give clients control over when to adopt changes.

---

2 — High-level strategies (what every engineer should know)

There are four common ways clients declare which version they want:

1. **URL path (URI) versioning**
   Example: `GET /api/v1/customers/123`
   Pros: easy to see, cache/CDN friendly, simple routing.
   Cons: couples versioning to URL structure; if you want semantic versioning (1.2 → 1.3) it can get noisy.

2. **Query string**
   Example: `GET /api/customers/123?api-version=1.0`
   Pros: easy for testing; compatible with clients that can't set headers.
   Cons: Not as clean as path; some caches treat query strings differently.

3. **Request header**
   Example: `GET /api/customers/123` with header `X-API-Version: 2`
   Pros: Clean URLs; hides versioning details from users; useful for headless clients.
   Cons: Less discoverable; caches and some proxies may need config to vary by header.

4. **Media-type (Accept/Content-Type) versioning** (a.k.a. content negotiation)
   Example: `Accept: application/vnd.myapp.v2+json`
   Pros: Very flexible (lets representation and version be negotiated).
   Cons: More complex to implement, less human readable, harder with generic HTTP clients.

---

3 — Design & operational considerations (advanced)

- **Discoverability**: URI-based versioning is easiest for devs to see. Header/media-type hides it, which is cleaner but less discoverable.

- **Caching & CDNs**: Path and query-string are cache/CDN friendly by default. Header-based or Accept-header versioning require cache configuration to vary by header.

- **Semantic vs breaking versions**: Decide on a versioning policy (major-only vs semver). Many teams use `major` numbers only for breaking changes (e.g., v1, v2) and use feature flags for minor changes.

- **Deprecation policy**: Communicate deprecation windows, return `Deprecation`/`Sunset` headers, and mark controllers as deprecated in docs.
- **Documentation & API discovery**: Keep docs for each version (Swagger/OpenAPI supports multi-version docs).
- **Evolution patterns**: Prefer additive changes (non-breaking) and use new endpoints for breaking changes; use DTO translation layers on server side to map older DTOs to newer internals.
- **Testing matrix**: Maintain automated tests across supported versions.

API versioning in .NET Framework

There is a widely-used, actively maintained library for API versioning in ASP.NET (both Web API and ASP.NET Core): **aspnet-api-versioning** (NuGet package `Microsoft.AspNet.WebApi.Versioning` for Web API). It provides attributes, routing support (URL segment), readers (query/header/media-type), and tooling like version reporting.

Key concepts when using this library in Web API 2

- Annotate controllers with `[ApiVersion("1.0")]` etc.
- Configure API versioning in `WebApiConfig.Register(HttpConfiguration config)` via `config.AddApiVersioning(...)`.
- The library provides `ApiVersionReader` implementations:
  - `UrlSegmentApiVersionReader` (URL path)
  - `QueryStringApiVersionReader` (query)
  - `HeaderApiVersionReader` (custom header)
  - `MediaTypeApiVersionReader` (accept header)
    You can combine readers (e.g., accept either url segment OR header).
- Route templates use the `apiVersion` route constraint when using URL segment routing: e.g. `Route("api/v{version:apiVersion}/values")`.

- Options: `ReportApiVersions`, `AssumeDefaultVersionWhenUnspecified`, `DefaultApiVersion`.