Contents

Thread

- A Thread is a process managed by the operating system that executes code instructions independently. It's like a worker that runs your code instructions.
- Can be created manually (new Thread(...)) → not from the ThreadPool, it's a dedicated OS thread you must manage.A real worker from the OS. You can create one manually, but then you must manage its lifetime yourself.
- **Resource cost**: Threads are *heavy*. Each thread needs memory (~1 MB stack) and CPU scheduling & CPU context,registers.
- B**est for**: CPU-bound tasks (parallelism).
- When you create a new thread, the OS actually allocates a separate stack and CPU context for it.

```csharp
Thread t = new Thread(() => {
    Thread.Sleep(2000); // blocks this thread
    Console.WriteLine("Thread work done");
});
t.Start();
```

Here, that thread is *blocked* for 2 sec → memory + CPU context is wasted.

- **Why use task over thread? Context switching between threads is expensive OS saves registers, stack pointers, program counter, etc. With async/await, instead of switching OS threads, the compiler-generated state machine remembers "where to continue" → no CPU context switch needed.** When the OS scheduler switches the CPU from running **Thread A** to **Thread B.** Save Thread A's context (registers, stack pointer, etc.). Load Thread B's context. Resume B where it left off. This is called **context switching**. It happens at the **OS level** and is **expensive** (microseconds, but adds up with thousands of threads)
- **"A Thread is just an OS worker that runs your code with its own stack and CPU context — heavy to create/manage, so we prefer Tasks + async/await to avoid costly context switching."**

Threadpool

- **ThreadPool** - The **Thread Pool** is a pool of worker reusable threads maintained by the CLR. Instead of creating new threads (expensive), .NET reuses threads from the pool.
- **is main thread is threadpool thread?** No. The Main thread that starts a console app is not a ThreadPool thread, it's a dedicated OS thread.

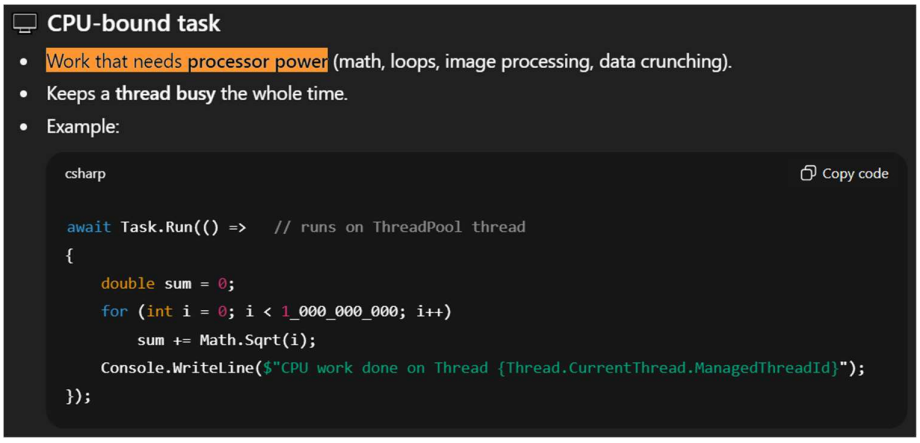
- **ThreadPool is like a bucket of reusable worker threads managed by CLR — faster than creating new threads, but the Main thread of your program is not part of this pool.**

Task

- A Task is a .NET abstraction that represents work to be done. It is Just a .NET wrapper that usually runs code on a ThreadPool thread (so you don't deal with raw threads).
- **Thread is low-level, Task is high-level, both use OS threads.**
- **A Task is a high-level .NET wrapper that represents some work, usually running on a ThreadPool thread — easier to use than raw low-level threads.**

Async / Await

- syntactic sugar over **Tasks + state machine**, resumes on a **ThreadPool thread** after I/O. async/await in Node.js is **syntactic sugar over Promises**. **Async/await ≠ new thread, it's about freeing threads during I/O.**
- When the code hits an await on an I/O-bound operation, the **current thread is freed** (not blocked). The actual work (like DB call, API call, file read) is handled by the OS/CLR asynchronously.
- Once the I/O completes, the continuation is picked up by a **ThreadPool thread**, and the state machine resumes execution from the point after await.
- Async/await does NOT create new threads.
- For CPU-bound work → the work runs directly on a ThreadPool OS thread. The ThreadPool thread stays busy until the work is finished. If I have a CPU-bound operation in async/await, what happens to the main thread and how is the work scheduled? The main thread continues immediately. The CPU-bound work is picked up by a ThreadPool thread, which executes it. CPU-bound task = needs a Thread (from ThreadPool).



- For I/O-bound work → no thread is blocked; the OS kernal does the waiting, and When done, .NET uses a ThreadPool OS thread to continue.

- Work that waits on **external resources** (file read/write, DB query, HTTP call).
  No thread is held while waiting → OS notifies when done → then a ThreadPool thread continues.
  Example:

```csharp
using var http = new HttpClient();

string data = await http.GetStringAsync("https://example.com");
Console.WriteLine($"I/O work done on Thread {Thread.CurrentThread.ManagedThreadId}");
```

  or

```csharp
await Task.Delay(2000); // simulated async wait, no thread held
Console.WriteLine("I/O resumed");
```

- **Best for**: I/O-bound tasks (network, database, file operations).
- **Resource cost**: Very light — no extra thread needed for waiting.
- **Blocking?** Doesn't block a thread while waiting (e.g., for I/O).
- Async/await is just a smart way to pause and resume Tasks using a compiler-made state machine — it doesn't create new threads, it frees the thread during I/O and later continues on a ThreadPool thread.

State Machine (in Async/Await)

- When you write an async method, the C# compiler doesn't execute it as-is.
  It rewrites it into a state machine (hidden code) so the program can *pause* and *resume* execution at each await.
- **Async method is converted by the compiler into a hidden state machine so it can pause at await and resume later from the same point.**
-

```csharp
static async Task DemoAsync()
{
    Console.WriteLine("Step 1");
    await Task.Delay(1000);
    Console.WriteLine("Step 2");
}
```

  You write it like above 👆, but the compiler turns it into something like this (simplified):

```
class DemoAsyncStateMachine : IAsyncStateMachine
{
    int state = -1;  // tracks where we are
    TaskAwaiter awaiter;

    public void MoveNext()
    {
        try
        {
            if (state == -1)  // initial
            {
                Console.WriteLine("Step 1");
                awaiter = Task.Delay(1000).GetAwaiter();
                if (!awaiter.IsCompleted)
                {
                    state = 0; // mark we are at "after await"
                    awaiter.OnCompleted(MoveNext); // resume later
                    return;
                }
            }

            if (state == 0) // resumed after await
            {
                awaiter.GetResult();
                Console.WriteLine("Step 2");
            }
        }
        catch (Exception ex) { /* handle */ }
    }
}
```

-

Asynchronous Programming

Asynchronous programming is long-running tasks (like I/O) that run without blocking a thread. The thread is freed while waiting, and execution resumes when the task get completes, improving responsiveness and scalability.

Asynchronous programming means doing long tasks (like I/O) without blocking a thread — the thread is freed to do other work and continues when the task is done.

Parellel Programming

Parallel programming is Running multiple tasks *at the same time* on different CPU cores/threads to finish CPU-bound work to speed up computations.

Parallel programming means running multiple CPU-bound tasks at the same time on different threads/cores to finish faster.

Question Imagine 5 users hit my API at the same time, and inside I make an HTTP call without async/await.

- Each request grabs a thread from the ThreadPool.
- That thread just waits (blocked) until the HTTP call finishes.
- So if all 5 threads are busy waiting, the 6th user has to wait until a free thread comes back.

Now with async/await:

- When my code hits await http.GetStringAsync(), the thread is released back to the pool.
- The waiting is handled by the OS, not by my thread.
- So while the HTTP call is pending, that freed thread can go handle another user's request.
- When the HTTP call completes, .NET grabs any available ThreadPool thread to continue.

Without async/await, each API request blocks a ThreadPool thread while waiting for I/O, so if all threads are busy, new requests have to wait. With async/await, the thread is freed while waiting for I/O, allowing other requests to use it, and the continuation resumes on a ThreadPool thread when the I/O completes.

If 100,000 users hit your web app at the same time, does the server create 100,000 threads to handle them?

No, The server uses a limited ThreadPool. If I block threads (no async/await), they pile up and the pool runs out → thread starvation. With async/await, threads are released during I/O, so a small pool of threads can handle thousands of requests efficiently.

Thread starvation means: too many blocked threads → slow server.

- All available threads in the ThreadPool are busy (often blocked).
- New incoming requests can't get a free thread. Result: slow responses, timeouts.
- So those requests are forced to wait a long time (or timeout).

The server doesn't create 100,000 threads; without async/await blocked threads cause thread starvation, but async/await frees threads during I/O so a few threads can handle many requests efficiently.

If you run 10 .exe processes on a machine with 4 CPU cores

- Multiple Processes vs CPU Cores
- If 10 processes run on 4 cores → OS time-slices threads.
- Only 4 threads run at the same instant. Others wait and rotate.
- Appears parallel, but actually fast switching.

Question Is the thread ID before and after await guaranteed to be the same?

No, the thread ID before and after await is not guaranteed to be the same; the continuation may resume on a different ThreadPool thread after I/O completes.

Main thread starts execution → Your code begins on the main thread (let's say Thread ID = 1). → When GetStringAsync is called, it *kicks off* an async network I/O operation → No thread is blocked → The HTTP request is handed off to the OS kernel → No .NET thread is held while waiting for the network response it sended to Threadpool to server other request→ OS notifies when I/O is done → ThreadPool thread (ID=9)resumes the async state machine.

```
static async Task Main(string[] args)
{
    Console.WriteLine($"Main started on Thread {Thread.CurrentThread.ManagedThreadId}");

    var http = new HttpClient();

    // Await an async operation
    string data = await http.GetStringAsync("https://example.com");

    // This part runs after await
    Console.WriteLine($"Continuation after await on Thread {Thread.CurrentThread.ManagedThreadId}"
}
```

OUTPUT → Main started on Thread 1 → Continuation after await on Thread 9

Question **If i have 10000 user will only main thread will take the requets?**

- No — the **main thread itself does NOT handle every user request**.
- The main thread's job is just to **start the application and listen for incoming HTTP requests**.
- When a user hits your server, the **request is taken by the main listener** and then **assigned to a ThreadPool thread** to actually process it.
- So even if 10,000 users hit your server at the same time, the main thread just keeps listening — it **never executes the request logic itself**.

No, the main thread only listens for requests; each request is handled by a ThreadPool thread, so the main thread never processes all user requests itself.

Question Output Not awaiting + Not returning Task

```
static void Main(string[] args)
{
    Console.WriteLine("Code 1");
    Console.WriteLine("Code 2");
    SomeMethod();               // async void
    Console.WriteLine("Code 7");
    Console.WriteLine("Code 8");
    Console.Read();
}

static async void SomeMethod()
{
    Console.WriteLine("Code 3");
    Console.WriteLine("Code 4");

    await Task.Delay(60000);   // 60 sec delay
    Console.WriteLine("Code 5");
    Console.WriteLine("Code 6");
}
```

```
Code 1
Code 2
Code 3
Code 4
Code 7
Code 8
(Waits for Console.Read input)
Code 5   ← after 60 seconds
Code 6   ← after 60 seconds
```

Question What is the difference between async void and async Task in C#?

```
async void MyVoid()
{
    await Task.Delay(1000);
    Console.WriteLine("Done");
}

static void Main()
{
    MyVoid();
    Console.WriteLine("After void");
}
```

```
After void
Done  // prints later
```

Async task non void it will wait because task will return.

```
async Task MyVoid()
{
    await Task.Delay(1000);
    Console.WriteLine("Done");
}

static async Task Main()
{
    await MyVoid();
    Console.WriteLine("After void");
}
```

```
Done
After void
```

Q: Why should we return Task or Task<T> instead of async void? What happens if we don't?

- Because await only works on Task/Task<T>. The compiler rewrites async code into a state machine that tracks when the task is paused and when to resume.
- If the method returns Task/Task<T>, the caller can await, so it **pauses until the task completes**, and exceptions are captured.
- If it's void, the state machine runs, but the caller **cannot await or resume properly** — it just fires and forgets.

Thread vs Task vs Async/ Await

Thread: A Thread is an OS worker with its own stack and CPU context — heavy to create/manage, used for CPU-bound work.

ThreadPool: ThreadPool is a bucket of reusable worker threads managed by CLR — faster than new threads; Main thread is not in the pool.

Task: A Task is a high-level .NET wrapper representing work, usually on a ThreadPool thread — easier than raw threads.

Async/Await: Async/await is compiler-made state machine syntax over Tasks that pauses/resumes code during I/O without creating new threads.

State Machine: The compiler rewrites async methods into a hidden state machine to pause at await and resume later from the same point.

Asynchronous Programming: Doing long tasks (like I/O) without blocking threads — the thread is freed to do other work and continues when task completes.

Parallel Programming: Running multiple CPU-bound tasks at the same time on threads/cores to finish faster.

Thread Blocking (without async/await): API requests block ThreadPool threads while waiting for I/O, causing other requests to wait.

Async/await behavior (I/O-bound): Threads are freed during I/O; continuation resumes on a ThreadPool thread when done.

Thread Starvation: Too many blocked threads → slow server, timeouts; async/await avoids this by freeing threads.

Thread ID after await: Not guaranteed — continuation may resume on a different ThreadPool thread.

Main Thread role: Only listens for requests; each request is handled by a ThreadPool thread.

Async void vs Task: async void → fire-and-forget, cannot await; async Task → awaitable, pauses caller, captures exceptions.

Why return Task/Task<T>: Allows caller to await, resumes correctly via state machine, and handles exceptions; void cannot.

```
Thread
  ↓ (heavy, OS worker, CPU-bound)
ThreadPool
  ↓ (reusable threads, faster than creating new threads)
Task
  ↓ (high-level wrapper over ThreadPool threads)
Async/Await
  ↓ (pauses/resumes code, frees thread during I/O)
State Machine
  ↓ (compiler rewrites async methods, remembers where to continue)
I/O-bound vs CPU-bound
  ↓
I/O-bound → thread freed, resumes later on ThreadPool thread
CPU-bound → ThreadPool thread busy until work finishes
```

Q1: Does async/await create a new thread?
A: No, it just pauses the method and frees the current thread; continuation runs later on a ThreadPool thread.

Q2: What happens if you use async void instead of Task?
A: Caller cannot await it, exceptions are unhandled, and it's fire-and-forget — use only for event handlers.

Q3: Is the thread ID guaranteed to be the same after await?
A: No, continuation may resume on a different ThreadPool thread.

Q4: Why return Task or Task<T> instead of void?
A: So the caller can await, the state machine tracks completion, and exceptions are captured.

Q5: What is thread starvation?
A: Too many blocked threads → ThreadPool exhausted → slow server or timeouts; async/await prevents this by freeing threads during I/O.

Q6: Main thread vs ThreadPool thread — who handles requests?
A: Main thread only listens; ThreadPool threads process actual requests.

Q7: Difference between async and parallel programming?
A: Async frees threads during I/O (scalability), parallel runs multiple CPU-bound tasks simultaneously (speed).

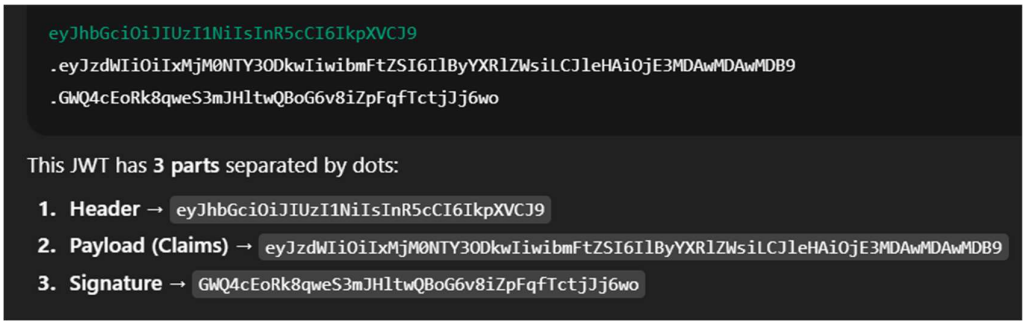
Q8: Can a Task run without a ThreadPool thread?
A: For CPU-bound work, it runs on a ThreadPool thread; for some custom schedulers, yes, but rare.

Q9: What's the advantage of Task over Thread?
A: Tasks are high-level, easier to manage, and avoid expensive OS thread creation/context switching.

Q10: How many threads will the server create for 100,000 simultaneous requests?
A: Only a limited number from ThreadPool; async/await allows many requests to share threads efficiently.

JWT

- **JWT (JSON Web Token)** is a **compact, stateless token** used for authentication/authorization.



-
- What does **Stateless** mean? **Stateless server** = The server does NOT remember anything about you between requests. Each request is **independent**. Example: You call API → server checks your token → gives response → forgets you.

Code Example - How JWT Authentication Works in .NET

(a) Install NuGet Packages

```bash
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

(b) appsettings.json (store secret key)

```json
"Jwt": {
  "Key": "ThisIsASecretKey12345",
  "Issuer": "MyApp",
  "Audience": "MyAppUsers",
  "ExpireMinutes": 60
}
```

STEP 1:→

Step 1: User Logs In

1. User provides credentials (username/password).
2. Server verifies credentials (DB check).
3. If correct → server creates a JWT token:
   - Header → algorithm info (`HS256`)
   - Payload → user data + claims (like `sub` , `role` , `exp`)
   - Signature → HMACSHA256(header.payload, secret)
4. Server returns JWT token to the client.

TOKEN CREATION WIYHOUT LIBRARY

```
// ---------------- CREATE TOKEN ----------------
public static string CreateToken()
{
    // 1. Header
    var header = new { alg = "HS256", typ = "JWT" };
    string headerJson = JsonSerializer.Serialize(header);
    string headerBase64 = Base64UrlEncode(Encoding.UTF8.GetBytes(headerJson));

    // 2. Payload (claims)
    var payload = new
    {
        sub = "12345",
        role = "Admin",
        exp = DateTimeOffset.UtcNow.AddMinutes(5).ToUnixTimeSeconds()
    };
    string payloadJson = JsonSerializer.Serialize(payload);
    string payloadBase64 = Base64UrlEncode(Encoding.UTF8.GetBytes(payloadJson));

    // 3. Signature = HMACSHA256(header.payload, secret)
    string unsignedToken = $"{headerBase64}.{payloadBase64}";
    string signature = CreateSignature(unsignedToken, secretKey);

    // 4. Final token
    return $"{unsignedToken}.{signature}";
}
```

🔧 Mini Example

Let's say:

- Header = `{ alg: "HS256", typ: "JWT" }`
- Payload = `{ sub: "123", role: "Admin" }`
- Secret = `"myKey"`

👉 Flow:

1. JSON → `{"alg":"HS256","typ":"JWT"}` → bytes → Base64Url → `aaa111` .
2. JSON → `{"sub":"123","role":"Admin"}` → bytes → Base64Url → `bbb222` .
3. Unsigned token = `aaa111.bbb222` .
4. Signature = `HMACSHA256("aaa111.bbb222", secretKey)` → bytes → Base64Url → `ccc333` .
5. Final JWT = `aaa111.bbb222.ccc333` .

What is HMACSHA256? Is used to generate hash signature

HMACSHA256 = SHA256 hash + secret key → creates a signature that proves the data is genuine and unchanged.


```
// Signature = HMACSHA256(header.payload, secretKey)
private static string CreateSignature(string data, string secretKey) =>
    Base64UrlEncode(new HMACSHA256(Encoding.UTF8.GetBytes(secretKey))
                    .ComputeHash(Encoding.UTF8.GetBytes(data)));

// Example: data = "aaa111.bbb222", secretKey = "myKey"
// → signature = "ccc333"
```

```
private static string CreateSignature(string data, string key)
{
    var keyBytes = Encoding.UTF8.GetBytes(key);
    using var hmac = new HMACSHA256(keyBytes);
    var hash = hmac.ComputeHash(Encoding.UTF8.GetBytes(data));
    return Base64UrlEncode(hash);
}
```

(d) Login Endpoint → Generate JWT

```csharp
[HttpPost("login")]
public IActionResult Login([FromBody] LoginModel model)
{
    if (model.Username == "admin" && model.Password == "123")
    {
        var claims = new[]
        {
            new Claim(JwtRegisteredClaimNames.Sub, model.Username),
            new Claim("role", "Admin"),
            new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
        };

        var key = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken(
            issuer: _config["Jwt:Issuer"],
            audience: _config["Jwt:Audience"],
            claims: claims,
            expires: DateTime.Now.AddMinutes(
                Convert.ToDouble(_config["Jwt:ExpireMinutes"])),
            signingCredentials: creds);

        return Ok(new { token = new JwtSecurityTokenHandler().WriteToken(token) });
    }
    return Unauthorized();
}
```

Step 2: Client Sends JWT

Step 3: .NET Middleware Validates JWT

(c) Program.cs / Startup.cs → Configure JWT

```csharp
builder.Services.AddAuthentication("Bearer")
    .AddJwtBearer("Bearer", options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,

            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
        };
    });

builder.Services.AddAuthorization();
```

How it works internally (basic):

- Middleware reads `Authorization` header.
- Splits token → header, payload, signature.
- Recalculates signature using secret key (HMACSHA256).
- Compares signature → If mismatch → reject.
- Checks expiry (`exp`) → If expired → reject.
- Reads claims from payload → creates a `ClaimsPrincipal` object.
- Sets `HttpContext.User = ClaimsPrincipal` → now your controllers know who the user is.

Saved memory full

```
// 1. Split JWT
var parts = token.Split('.');
string headerB64 = parts[0];
string payloadB64 = parts[1];
string signature = parts[2];

// 2. Recalculate Signature
string newSig = CreateSignature($"{headerB64}.{payloadB64}", "myKey");
if (newSig != signature) return false; // invalid

// 3. Decode Payload
var payloadJson = Encoding.UTF8.GetString(Base64UrlDecode(payloadB64));
var payloadData = JsonSerializer.Deserialize<Payload>(payloadJson);
if (payloadData.exp < DateTimeOffset.UtcNow.ToUnixTimeSeconds()) return false; // expired

// 4. Create ClaimsPrincipal
var claims = new[]
{
    new Claim("sub", payloadData.sub),
    new Claim(ClaimTypes.Role, payloadData.role)
};
var identity = new ClaimsIdentity(claims, "jwt");
var principal = new ClaimsPrincipal(identity);

// 5. Assign to HttpContext.User
HttpContext.User = principal;

// ✅ Now [Authorize] and HttpContext.User can read claims
```

Mini Example (continuing earlier one)

Token:

```
aaa111.bbb222.ccc333
```

1. Split → header= `aaa111` , payload= `bbb222` , signature= `ccc333` .
2. Recalculate:
   - `unsigned = "aaa111.bbb222"` .
   - `newSig = HMACSHA256(unsigned, secret) → Base64Url` .
3. Compare:
   - If `newSig == ccc333` → valid.
   - Else → tampered.
4. Decode `bbb222` → `{"sub":"123","role":"Admin","exp":...}` .
5. Check `exp` .

Then set to claimprinciple.

Step 4: Authorization

- You can now use [Authorize] or [Authorize(Roles="Admin")] in controllers.
- The middleware already filled HttpContext.User with claims.
- ASP.NET checks these claims to allow or deny access.
-

```
[Authorize(Roles="Admin")]
[HttpGet("admin")]
public IActionResult AdminOnly()
{
    var userId = HttpContext.User.FindFirst("sub")?.Value;
    return Ok($"Hello Admin {userId}");
}
```

What is Claims , Claim Identity and ClaimsPrincipal ?

- Claims: A piece of info about the user.
- Claim Principle : ⍰ It's a **.NET object that represents the current user**. It Holds **claims**. ASP.NET stores it in HttpContext.User automatically after authentication.
    - **HttpContext.User** → The place where ASP.NET stores the current user's ClaimsPrincipal so your controllers/middleware can read it.

```
using System.Security.Claims;

// 1. Create claims
var claims = new[]
{
    new Claim("sub", "12345"),
    new Claim(ClaimTypes.Role, "Admin")
};

// 2. Create identity & principal
var identity = new ClaimsIdentity(claims, "jwt");
var principal = new ClaimsPrincipal(identity);

// 3. Set to HttpContext.User (normally done by middleware)
HttpContext.User = principal;

// 4. Access claims later
var userId = HttpContext.User.FindFirst("sub")?.Value; // 12345
var role = HttpContext.User.FindFirst(ClaimTypes.Role)?.Value; // Admin
```

Q: Do you need ClaimsIdentity / ClaimsPrincipal when creating a token?

No, they are only needed when validating a token and setting the user context.

1. **Base64** → Safe text encoding of JSON (so it can travel in HTTP headers).
    👉 Example: `"{"user":"Prateek"}"` → `eyJ1c2VyIjoiUHJhdGVlayJ9`
2. **UTF8.GetBytes** → Converts text into raw bytes (needed before hashing).
    👉 Example: `"Hello"` → `[72, 101, 108, 108, 111]`
3. **HMACSHA256** → Cryptographic algorithm that uses a **secret key** + **data** to generate a hash.
    👉 Example: `HMACSHA256("data", "secret")` → `abc123hash...`
4. **Signature** → The final hashed value added to the token, so if data is changed, signature won't match.
    👉 Example: Token parts = `Header.Payload.Signature` → If Payload changes, Signature breaks ❌
5. **Claims** → Actual info inside the token (like user ID, expiry, roles/permissions).
    👉 Example: `{"sub":"101", "name":"Prateek", "role":"Admin", "exp":1723456789}`

JWT VAlidation

```
public static bool ValidateJwt(string token, string secret)
{
    var parts = token.Split('.');
    if (parts.Length != 3) return false;

    var header = parts[0];
    var payload = parts[1];
    var signature = parts[2];

    // Recompute signature using secret key
    var key = Encoding.UTF8.GetBytes(secret);
    using var hmac = new HMACSHA256(key);
    var computedHash = hmac.ComputeHash(Encoding.UTF8.GetBytes($"{header}.{payload}"));
    var computedSignature = Base64UrlEncode(computedHash);

    // Compare with given signature
    return signature == computedSignature;
}

private static string Base64UrlEncode(byte[] input)
{
    return Convert.ToBase64String(input)
        .TrimEnd('=').Replace('+', '-').Replace('/', '_');
}
```

Common Interview Questions

Is JWT stored on server?

❌ No → it's stateless. Token is validated using secret key only.

Difference: JWT vs Cookies?

- Cookie = server session stateful.
- JWT = client-side stateless.

How to invalidate JWT early? After logout how to invalidate(store in session)

- Use a **refresh token mechanism** or maintain a **token blacklist** in DB/Redis.

Q: Are JWT header and payload secure?

No, they are Base64Url-encoded, not encrypted. Anyone can decode and read them.

Q What are refresh token?

A **refresh token** is a long-lived token used to get a new **access token** after the original access token expires, without asking the user to log in again.

```
1. User logs in → Server issues Access Token + Refresh Token
2. Client calls API with Access Token → works until expiry
3. Access Token expires → Client sends Refresh Token to /refresh endpoint
4. Server validates Refresh Token (DB/Redis)
5. If valid → Server issues new Access Token (and optionally new Refresh Token)
6. If invalid → Client must log in again
```

JWT TOKEN

1. **What is JWT?**

   JWT (JSON Web Token) is a compact token used for authentication/authorization. It has 3 parts:

   - **Header** (algorithm info),
   - **Payload** (claims like user id, role, expiry),
   - **Signature** (Header+Payload hashed with secret).

2. **How it works?**
   - When user logs in → server creates JWT by encoding header & payload and signing with secret.
   - Token is sent to client (browser/mobile).
   - On each request → client sends `Authorization: Bearer <token>`.
   - Server verifies signature + expiry → if valid, sets **HttpContext.User = ClaimsPrincipal**.
3. **Authentication vs Authorization:**
   - **Authentication** = Check who you are (token valid?).
   - **Authorization** = Check what you can do (claims like `role=Admin`).
4. **Claims & ClaimsPrincipal:**
   - Payload → converted into `ClaimsIdentity` → wrapped into `ClaimsPrincipal`.
   - This is stored in `HttpContext.User`.
   - `[Authorize(Roles="Admin")]` works by checking these claims.
5. **Access vs Refresh Token:**
   - **Access Token** = short-lived, sent on each request.
   - **Refresh Token** = long-lived, used only to get a new access token.
6. **Security Note:**
   - Header & Payload are just **Base64Url**, anyone can read them.
   - Only **Signature** ensures data is untampered.