Contents

[Thread vs Task vs Async/ Await 8](#_Toc208757117)

[JWT TOKEN 17](#_Toc208757118)

**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.
* **Best for**: CPU-bound tasks (parallelism).
* When you create a new thread, the OS actually allocates a separate stack and CPU context for it.
* **A computer screen shot of a computer code

  AI-generated content may be incorrect.**
* **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).
* A computer screen shot of a program

  AI-generated content may be incorrect.
* 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.
* A screen shot of a computer

  AI-generated content may be incorrect. A screen shot of a computer

  AI-generated content may be incorrect.
* **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.**
* A screenshot of a computer program

  AI-generated content may be incorrect.
* A computer screen shot of code

  AI-generated content may be incorrect.

**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.

**A screen shot of a computer program

AI-generated content may be incorrect.**

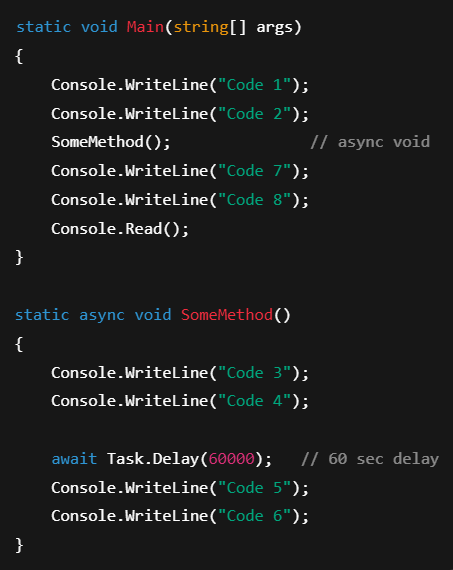
**OUTPUT 🡪 Main started on Thread 1 🡪 Continuation after await on Thread 9**

**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**

**** **A screen shot of a computer code

AI-generated content may be incorrect.**

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

A computer screen shot of text

AI-generated content may be incorrect. A black background with white text

AI-generated content may be incorrect.

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

**A computer screen shot of text

AI-generated content may be incorrect.** A black background with white text

AI-generated content may be incorrect.

**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.

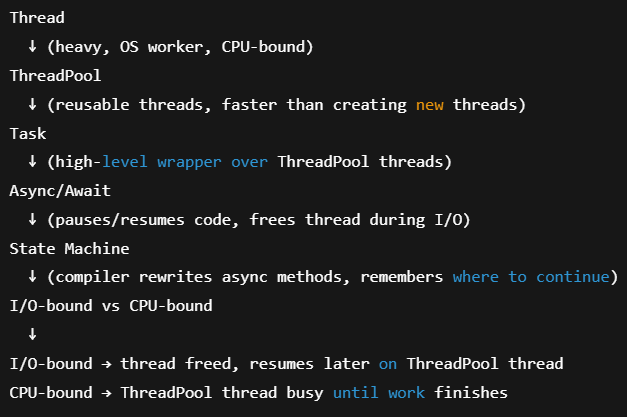
**Thread Starvation:** Too many blocked threads → slow server, timeouts; async/await avoids this by freeing threads.

**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.



**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.
* A screenshot of a computer

  AI-generated content may be incorrect.
* 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 black screen with white text

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

STEP 1:🡪

A screenshot of a black screen

AI-generated content may be incorrect.

TOKEN CREATION WIYHOUT LIBRARY

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

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.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect. A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

Step 2: Client Sends JWT

**Step 3: .NET Middleware Validates JWT**

A computer screen shot of a program code

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. 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.
* A computer screen with white and red text

  AI-generated content may be incorrect.

What is Claims , Claim Identity and ClaimsPrincipal ?

* Claims: A piece of info about the user.
* Claim Principle : It’s a **.NET identity 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.

A computer screen shot of a program

AI-generated content may be incorrect.

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

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

A screenshot of a computer program

AI-generated content may be incorrect.

JWT VAlidation

A screen shot of a computer program

AI-generated content may be incorrect.

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)**

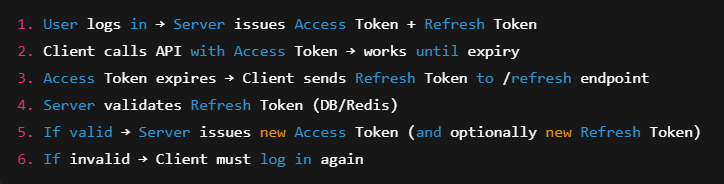
* Use a **refresh token mechanism** or maintain a **token blacklist** in DB/Redis.

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.



## **JWT TOKEN**

A screenshot of a computer

AI-generated content may be incorrect.

In Controller

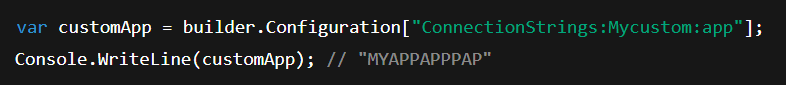
A computer screen with text on it

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

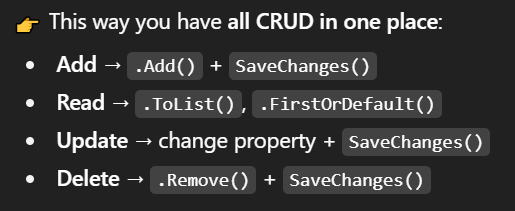
In Program.CS



A screenshot of a computer

AI-generated content may be incorrect.

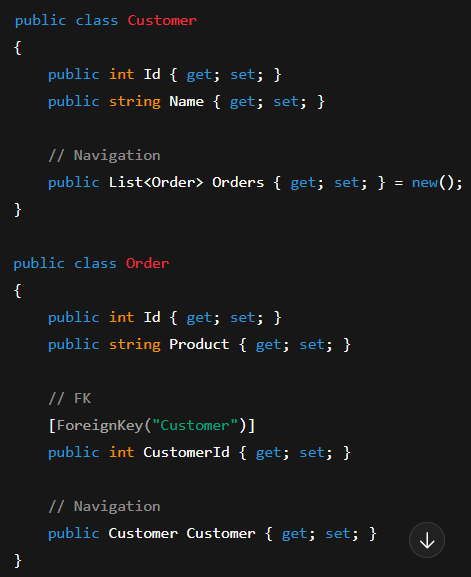


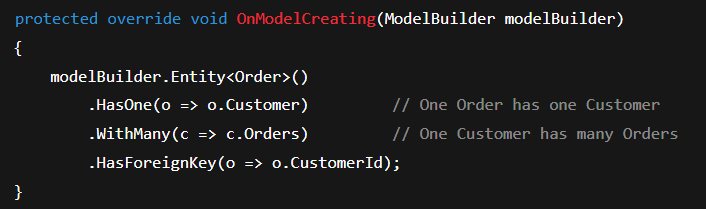


What are Relationships(**Navigation Property**)?

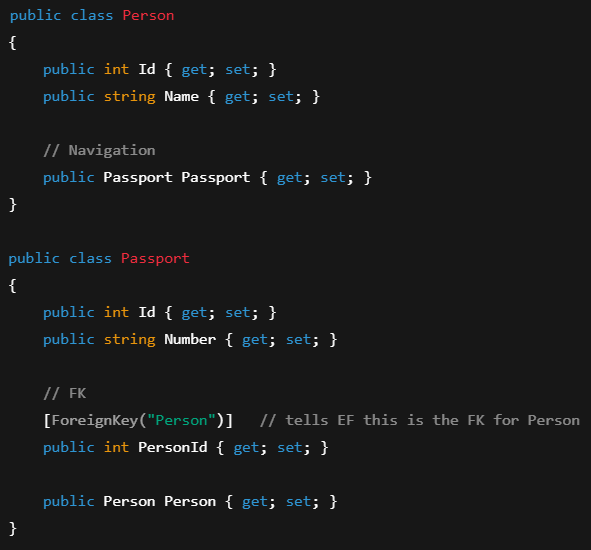
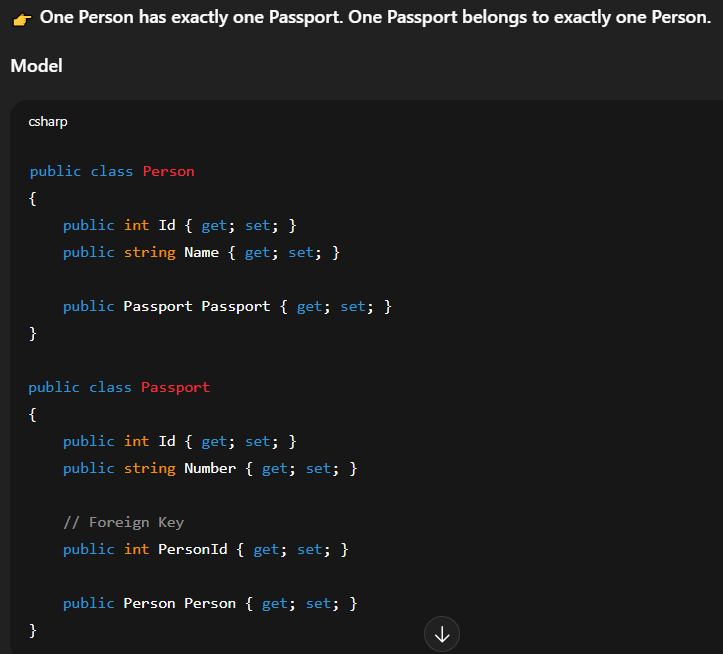
* A **relationship** means how two tables are connected in the database.
* A **Navigation Property** is how EF Core lets you **navigate** from one entity to another in C# objects.

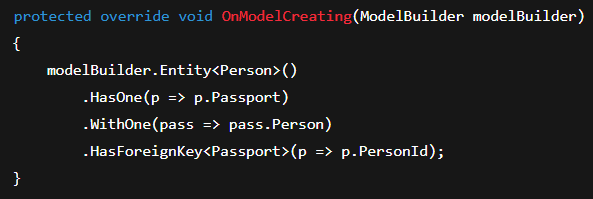
1. One-to-Many- Data annotation and fluent api both





1. One-to-One





1. Many-to-Many (Ignore applying data annotation on it otherwise create separate model with annotation)

