MVC TASK 6

Mohamed Haitham Afify

LinkedIn Article:

https://www.linkedin.com/pulse/understanding-middleware-aspnet-core-spotlight-kestrel-mohamed-afify-lstve/?published=t

Three use cases we need asynchronous programming?

1. Handling Web Requests Efficiently (e.g., in ASP.NET Core or Node.js)

When a web server receives thousands of HTTP requests per second, **asynchronous I/O** allows it to handle many requests *without blocking threads*.

Example:
Suppose your ASP.NET Core app fetches data from a database or calls an external API.
If the call is **synchronous**, the server thread must *wait* for the result — wasting resources.
But with **async/await**, that thread is released to handle another request until the data arrives.

Result:

- Higher throughput
- Better scalability
- More responsive applications

2. File and Network I/O Operations

File and network access are **slow** compared to CPU operations. If these are handled synchronously, the program freezes while waiting for data.

Example:

- Downloading or uploading files
- Reading/writing large files to disk
- Communicating with cloud storage (Azure Blob, AWS S3, etc.)

By using async operations (await File.ReadAllTextAsync(), for example), your app continues executing other work while waiting for I/O completion.

Result:

- Non-blocking user interface
- Smooth user experience
- Better utilization of system resources

3. Keeping Desktop or Mobile Apps Responsive

In UI-based applications (like WPF, Windows Forms, or mobile apps), running long tasks (database query, API call, etc.) synchronously **freezes the UI**.

Example:
A “Load Data” button fetches information from an online service.
If it’s synchronous, the window hangs until the request completes.
If it’s **asynchronous**, the UI stays interactive, and you can show a loading spinner or cancel button.

Result:

- Smooth, responsive interfaces
- Better user experience
- No “Not Responding” issues

What is the difference between thread and task ?

1. What is a Thread?

A **Thread** is a *low-level* concept — it represents an actual path of execution handled by the **operating system**.
It’s what the CPU runs your code on.

- Each thread has its own **stack**, **registers**, and **context**.
- Threads are **expensive** to create and manage.
- The OS schedules threads on available **CPU cores**

2. What is a Task?

A **Task** in C# is a **higher-level abstraction** built on top of threads.
It represents a **unit of work** — something that *may* run on a thread, or may just represent an asynchronous operation (like an I/O request).

- Task lives in the **Task Parallel Library (TPL)**.
- It can use **thread pool threads** automatically (you don’t manage threads yourself).
- It supports **continuations** and integrates with the async/await model.

What is delegate built_in types in details?

Built-in Delegate Types in .NET

Instead of creating your own delegate type manually every time,
C# provides three built-in generic delegate types:

- Action
- Func
- Predicate

`Action` — For Methods That Return `void`

- An Action delegate represents a method that returns nothing (`void`).
  - `Action action = () => Console.WriteLine("Hello from Action!");`
  - `action();`
- Output:
  - `Hello from Action!`
- It can also take parameters (up to 16).

Func — For Methods That Return a Value

A Func delegate represents a method that *returns* a value.

Example:

```
Func<int, int, int> add = (a, b) => a + b;
int result = add(5, 3)
Console.WriteLine(result);
```

Output: 8

Here:

- The last generic type is the return type.
- The others (if any) are input parameters.

`Predicate` — For Methods That Return `bool`

A **Predicate** is a *special case* of `Func<T, bool>` — used when you want to **test a condition**.

Example:

```
Predicate<int> isEven = num => num % 2 == 0;
Console.WriteLine(isEven(4)); // True
Console.WriteLine(isEven(7)); // False
```