# Overview of the asynchronous model

* The core of async programming is the Task and Task<T>
* **For I/O-bound code (db operation , Network operation File operation)**
  + you await an operation that returns a Task or Task<T> inside of an async method.
* **For CPU-bound code,** 
  + you await an operation that is started on a background thread with the [Task.Run](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run) method.
* The await keyword control to the caller of the method that performed await, and it ultimately allows a UI to be responsive or a service to be elastic/scale.
* compiler transforms your code into a **state machine** that keeps track of things like **yielding execution** when an await is reached and resuming execution when a background job has finished.
* When the await keyword is applied, it suspends the calling method and yields control back to its caller until the awaited task is complete.

# Async & Await

* An async method runs synchronously until it reaches its first await operator, at which point the method is suspended while the awaited task is completed. In the meantime, the control returns to the caller of the method.
* If the async method does not contain an await operator, the method executes synchronously.
* In C#, a Task represents an asynchronous operation
  + It is analogous to a promise or observables, used in other languages.
* Async method contains at least one await keyword
* Consequently, an async method with no await keyword is executed synchronously. A warning is emitted in this situation.
* In the async method MethodAAsync(), once the keyword await is meet for the first time the remaining of the task is actually executed by some random threads obtained from the runtime thread pool.- Asked by interviewer
* <https://blog.ndepend.com/c-async-await-explained/> - programe
* Async & await use thread pool effectively - <https://www.youtube.com/watch?v=_T3kvAxAPpQ>
* Task.Run(Async Method()) – fire and forget

# difference between Task.WaitAll and Task.WhenAll



## Task.WhenAll

* + It returns a Task that completes **when all the tasks in its argument list have completed**
  + running all method /endpoint in parallel
    - await Task.WhenAll(sumTask, wordTask);
  + Does not block thread
  + If you want to just be able to know when all tasks complete but not actually block your code then use WhenAll.
  + It is used to create a task that will complete if and only if all the other tasks have completed.

## Task.WaitAll

* + WaitAll blocks the current thread until all other tasks have completed execution
  + So, if you want to block until all tasks complete then use WaitAll.
  + Task.WaitAll blocks the current thread. It will remain blocked until all other tasks have completed execution. It has a void return value.