**C# 5: Async / Await**

Muy buenas! Como dije en el post anterior estoy trasteando un poco con la Developers Preview de Windows 8 y la nueva API WinRT para crear aplicaciones Metro. El tema está en que esta nueva API está diseñada de forma muy asíncrona. Por suerte en C# 5 el uso de métodos asíncronos se ha simplificado mucho gracias a dos nuevas palabras clave: async y await. Y dado que, creedme, vais a tener que usarlas en cuanto os pongáis con WinRT me he decidido escribir este post para comentarlas un poco 🙂

**async**

Esta palabra clave se aplica a la declaración de un método pero, contra lo que se suele pensar por primera vez **no declara que un método se ejecuta asíncronamente.** La palabra clave async lo que indica es que *este método se quiere sincronizar con métodos que se ejecutarán de forma asíncrona*. Si no usáis async podréis seguir llamando métodos de forma asíncrona (de hecho hasta ahora lo veníamos haciendo), lo que no podréis hacer (de forma trivial) es *sincronizaros con este método asíncrono*. ¿Que significa sincronizarse con un método asíncrono? Fácil: esperar a que termine. Ni más, ni menos.

Es como si *yo* me voy a un bar y allí me encuentro a un colega. Me siento con él, y pido una cerveza, de la misma que está tomando él. Mientras me la traen hablamos de temas de la vida hasta que mi colega propone un brindis. Pero todavía no ha llegado el camarero con mi cerveza, así que yo y mi amigo nos quedamos esperando *sin hacer nada* a que el camarero llegue (sí, los dos somos un poco nerds). Cuando el camarero llega, hacemos el brindis y seguimos hablando.

Declarar un método como async es requisito indispensable para poder usar await.

**await**

Esa palabre clave es la que permite que un método que ha llamado a otro método asíncrono se espere a que dicho método asíncrono termine. Usando de nuevo el ejemplo del bar, cuando mi amigo dice de hacer el brindis *debemos esperarnos a que llegue el camarero con mi cerveza*.

La clave de todo es entender que *desde el momento en que yo encargo mi cerveza al camarero* (llamada al método asíncrono) *hasta el momento en que decidimos que nos debemos esperar* yo (con mi amigo) hemos estado haciendo otras cosas (hablando de la vida). Eso, de nuevo trasladado a código fuente, significa que entre la llamada al método asíncrono y el uso de await habrá más líneas de código fuente. Por lo tanto no usamos *await* cuando llamamos al método asíncrono, lo hacemos más tarde cuando queremos esperarnos a que dicho método termine (y recoger el resultado, es decir mi cerveza).

Así pues… *sobre que aplicamos await?* Pues todo método que quiera ser ejecutado asíncronamente **debe** devolver un objeto especial, que sea (ojo con la originalidad de los ingleses) *awaitable*. Sobre este objeto, es sobre el que llamaremos a await para esperarnos a que el método asíncrono finalice y a la vez obtener el resultado. ¿Y que es un objeto *awaitable*? Pues un conocido de la TPL que viene con .NET 4: Un objeto Task o su equivalente genérico Task<T>.

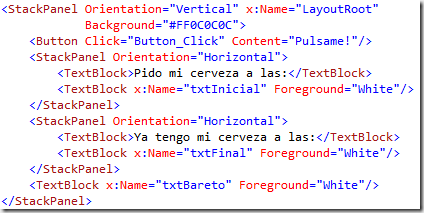
**Métodos asíncronos**

Para declarar un método que pueda ser llamado de forma asíncrona, lo único que debemos hacer es devolver un Task o Task<T> desde este método. Así se sencillo. Dejemos las cosas claras (al contrario que el chocolate): Devolver un Task NO convierte el método en asíncrono. Es la propia Task que es asíncrona. Podemos ver una Task como un delegate (o sea un método) que puede ser ejecutado de forma asíncrona. Trasladando eso de nuevo al ejemplo del bar, cuando yo pido la cerveza al camarero, le he encargado esta *tarea* y la he puesto en marcha. En términos de C# cuando llamo al método ServirCerveza de la clase camarero, este método me devuelve una Task<Cerveza>, que representa la tarea que he encargado al camarero. Luego **yo debo poner en marcha** esa tarea (con lo cual el camarero irá efectivamente a buscarla) y cuando toque esperarnos llamaremos a *await* sobre el objeto Task<Cerveza>. El resultado de llamar a *await* sobre una Task<Cerveza> es precisamente… un objeto de la clase Cerveza (mi cerveza para ser más exactos).

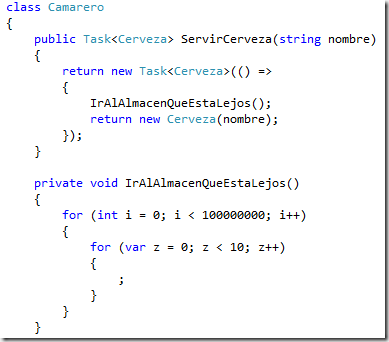
**Código, código, código**

Vamos a ver el ejemplo de la cerveza implementado en C# para que nos queden los conceptos más claros 😉

Para ello, dado que en la versión de VS2011 que viene con la Developers Preview de Win8 no podemos crear aplicaciones de consola, vamos a crear una aplicación Metro. En la vista principal pondremos 3 texblocks que nos permitirán ver cuando pedimos la cerveza al camarero, cuando nos la trae y como”hablamos” entre medias. El código XAML es muy simple:

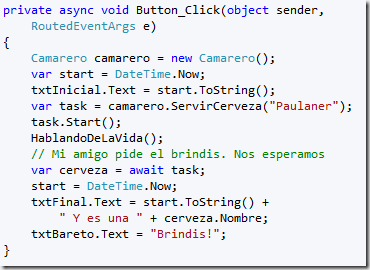
[](http://geeks.ms/cfs-file.ashx/__key/CommunityServer.Blogs.Components.WeblogFiles/etomas/image_5F00_40F7B18D.png)

Lo siguiente que necesito es una clase que represente a mi Camarero y un método que me permita pedirle una cerveza de forma asíncrona. Recordad que entonces debo declarar el método que me devuelva, no un objeto de Cerveza sino un objeto de Task<Cerveza>:

[](http://geeks.ms/cfs-file.ashx/__key/CommunityServer.Blogs.Components.WeblogFiles/etomas/image_5F00_27CC3188.png)

El método ServirCerveza de la clase Camarero espera un parámetro (el tipo de cerveza se quiere) y lo que hace es devolver una Task<Cerveza>. Como comenté una Task *es parecido a un delegate* sólo que es asíncrona, y en este caso una Task<T> se inicializa a partir de una Func<T> que le indica precisamente que se tendrá que hacer cuando se inicie la tarea. En nuestro ejemplo el camarero debe ir al almacén (que está lejos y es una operación que tarda un poco) y devolver la cerveza que se ha pedido.

Vayamos ahora a lo que ocurre cuando se pulse el botón:

[](http://geeks.ms/cfs-file.ashx/__key/CommunityServer.Blogs.Components.WeblogFiles/etomas/image_5F00_2542FFCA.png)

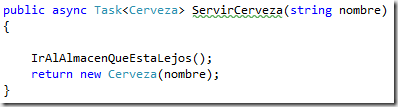
Ponemos en txtInicial la fecha y hora en que pedimos la cerveza. Y llamamos al método ServirCerveza del camarero. Este método retorna en el acto y nos devuelve una Task<Cerveza>. En este momento la ejecución de la tarea del camarero todavía no ha empezado. Cuando llamamos a task.Start() empieza la ejecución de dicha tarea de forma *asíncrona*. Y a la vez, yo y mi amigo seguimos hablando de cosas de la vida. El código de HablandoDeLaVida() se ejecuta concurrentemente con el de la tarea definida en ServirCerveza. Al final mi amigo propone el brindis y como no tengo cerveza nos esperamos, usando await sobre el objeto Task<Cerveza> que había recibido. Con esto nos esperamos a que finalice dicha tarea y obtenemos el resultado (que dado que era una Task<Cerveza> el resultado es una Cerveza). Y listos.

Observad como la función Button\_Click ha sido declarada como async para indicar que quiere llamar a métodos asíncronos y usar await para sincronizarse con ellos (esperar a que terminen).

El uso de async y await, junto con la clase Task de la TPL hace que en C#5 el crear y consumir métodos asíncronos sea tan fácil como ir al bar y pedir una cerveza! 🙂

Un saludo!

PD: Un comentario final, que quiero poner por completitud del post. Si declaráis un método async (porque quiereis hacer *await* sobre algún método asíncrono) *pero a la vez este método async puede ser llamado de forma asíncrona* y por lo tanto devuelve una Task<T>, entonces en la implementación del método async no es necesario que creeis la Task<T>, sino que podeis devolver directamente un objeto de tipo T. Es decir, en nuestro ejemplo el siguiente código:

[](http://geeks.ms/cfs-file.ashx/__key/CommunityServer.Blogs.Components.WeblogFiles/etomas/image_5F00_1071074A.png)

Compila correctamente. El método devuelve una Task<Cerveza> pero a diferencia de antes no tengo que crearla explicitamente. Y eso es debido al uso de async en la declaración. Eso supongo que es porque en muchos casos se van a ir encadenando métodos asíncronos y así nos ahorramos el tener que definir las Task<T> de forma explícita. Pero insisto, no os confundáis: es Task<T> lo que hace que el método *pueda* ser llamado de forma asíncrona, no async. De hecho si os fijáis en la imagen el nombre del método está subrayado en verde y eso es porque el compilador me está avisando que he declarado un método async… que no usa await en ningún momento, cosa que no tiene sentido (porque la única funcionalidad de async es permitir que el método use await).