Bringing async/await to the Contacts service

This article explains how to convert event-based asynchronous APIs to take advantage of the new C# 5.0 *async/await* keywords for asynchronous programming.

Introduction

With the introduction of asynchronous programming capabilities in C# 5.0 it becomes easier to write asynchronous code in a sequential manner.

But the new async capabilities rely on the task-based asynchronous programming model and not all APIs use it. In fact, for Windows Phone programming, at the current time, there are no APIs using the task-based asynchronous programming model.

For APIs using .NET's asynchronous programming model (APM) (introduced in the first version of the .NET Framework) a conversion is available through the TaskFactory.FromAsync extension method.

But there is no conversion available for APIs using the event-based asynchronous programming model (like the Contacts service in Windows Phone). The reason that such a conversion does not exist is that there is no clear pattern for event-based asynchronous APIs. Some allow cancellation (like the WebClient class) and others don't (like the Contacts class).

In this article we will learn several ways to convert an **event-based asynchronous API** into a **task-based asynchronous API** - using the Contacts service as an example.

Architectural Overview

The sections below all use the same basic mechanism for converting between event-based and task-based patterns (albeit with different mechanisms for extending the API).

The task-based asynchronous pattern allows that we return a “future value” to the caller immediately (represented by the Task class). The "real value" will be available as the `Result` property of this `Task` when the asynchronous operation completes.

We get this `Task` that will give us the future value by creating an instance of the TaskCompletionSource class of the intended type. The value of `Task` property of the `TaskCompletionSource` will be returned to the caller to await on it. When the caller awaits on the returned `Task`, it will be semantically blocked until the result of the task is set by calling `SetResult()` on the handler of the Completion event of the event-based API.

"Semantically blocked" means that the flow of execution of the calling method will be interrupted and resumed when the asynchronous operation is completed. This normally means that the executing thread will be blocked. However, on UI threads (such as the Windows Phone UI) that need its message pump working in order to keep the UI responsive, the flow of execution is returned to the message pump and when the asynchronous operation ends a message for the continuation is posted on the message pump and the execution resumes on the next instruction.

For more information see How to: Use Components That Support the Event-based Asynchronous Pattern and Implement the Task-based Asynchronous Pattern.

Extension Methods

In this case we use an extension method to implement the algorithm explained in the preceding section.

```
public static class ContactsAsyncExtenstions
{
    public static Task<IEnumerable<Contact>> SearchAsync(
        this Contacts contacts,
        string filter,
        FilterKind filterKind)
    {
        var taskCompletionSource =
            new TaskCompletionSource<IEnumerable<Contact>>();

        EventHandler<ContactsSearchEventArgs> handler = null;
        handler = (s, e) =>
```


(C) Copyright Nokia 2014. All rights reserved.

```
        {
            contacts.SearchCompleted -= handler;

            taskCompletionSource.TrySetResult(e.Results);
        };
        contacts.SearchCompleted += handler;

        contacts.SearchAsync(filter, filterKind, null);

        return taskCompletionSource.Task;
    }
}
```

Note that we do not have control over the instance of `Contacts` that we are operating on, so the first thing we do when handling the event is to unsubscribe from the event. As a general practice, we should also unsubscribe from the event if any error might occur.

Tip: If an exception could be thrown by the Contacts API we would also have to call `TaskCompletionSource.SetException` with the exception in the event handler.
This code doesn't check for exceptions because none are listed in the documentation for `Contacts.SearchAsync` and `ContactsSearchEventArgs.Result`. If `Contacts.SearchAsync` does throw an exception this will be propagated to the caller as it is still in the synchronous path.

Tip: Although `Contacts.SearchAsync` doesn't allow cancellation, a `CancellationToken` could be used to cancel the asynchronous operation. But for APIs that do not support cancellation, a more general approach like this is a better option

Using the `SearchAsync()` extension method is as shown below:

```
async Task<IEnumerable<Contact>> GetContactsAsync()
{
    var contactsService = new Contacts();
    var contacts = await contactsService.SearchAsync(filter: "filter", filterKind:
FilterKind.None);
    return contacts;
}
```

Extension method using async/await in the implementation of SearchAsync

Using async/await in the implementation of the `SearchAsync()` method will make it easier to write and easier to understand:

```
public async static Task<IEnumerable<Contact>> SearchAsync(this Contacts contacts,
string filter, FilterKind filterKind)
{
    var tcs = new TaskCompletionSource<IEnumerable<Contact>>();

    EventHandler<ContactsSearchEventArgs> handler = (s, e) => {
tcs.TrySetResult(e.Results); };

    try
    {
        contacts.SearchCompleted += handler;

        contacts.SearchAsync(filter, filterKind, null);

        return await tcs.Task;
```


(C) Copyright Nokia 2014. All rights reserved.

```
    }
    finally
    {
        contacts.SearchCompleted -= handler;
    }
}
```

Extending the Contacts class

Although using extension methods is a perfectly valid way of extending types we cannot (or do not want to) derive from, many times there are advantages in extending the type by delegation. This approach creates a new type that wraps the extended type. By extending the `Contacts` class we can also control its lifetime and the access to its members.

In our implementation instead of relying in an instance of the `Contacts` class provided by the caller, we will use an internal instance created when the instance of our class is created:

```
public class ContactsAsync
{
    private Contacts contacts = new Contacts();
}
```

Because we have full control of the instance of `Contacts`, we can have a global handler for the `SearchCompleted` event and create it immediately after its creation:

```
public ContactsAsync()
{
        this.contacts.SearchCompleted += (s, e) =>
            {
                // ...
            };
}
```

In the previous implementations we had an event subscription for every call and relied on the C# compiler to capture de instance of `TaskCompletionSource` and provide it to the event handler. Now that we have a global event handler, how are we going to supply it the exact instance of `TaskCompletionSource` corresponding to each call?

If you notice, event-based asynchronous APIs usually have an overload that allows supplying a state and using it in the event handler. In the case of the `Contacts` class this is done through the `state` argument of the `SearchAsync` method and the `State` of the `ContactsSearchEventArgs` class.

So, the implementation of our `SearchAsync()` becomes:

```
public Task<IEnumerable<Contact>> SearchAsync(
    string filter,
    FilterKind filterKind)
{
    var tcs = new TaskCompletionSource<IEnumerable<Contact>>();

    this.contacts.SearchAsync(filter: filter, filterKind: filterKind, state: tcs);

    return tcs.Task;
}
```

and the event handler becomes:

(C) Copyright Nokia 2014. All rights reserved.

```
(e.State as TaskCompletionSource<IEnumerable<Contact>>).TrySetResult(e.Results);
```

To complete the implementation of our `ContactsAsync` class we just need to add access to the `Accounts` property of the `Contacts` class through our own `Accounts` property:

```
public IEnumerable<Account> Accounts
{
        get { return this.contacts.Accounts; }
}
```

Here it is the complete implementation of our `ContactsAsync` class:

```
public class ContactsAsync
{
    private Contacts contacts = new Contacts();

    public ContactsAsync()
    {
        this.contacts.SearchCompleted += (s, e) =>
            {
                (e.State as
TaskCompletionSource<IEnumerable<Contact>>).TrySetResult(e.Results);
            };
    }

    public IEnumerable<Account> Accounts
    {
        get { return this.contacts.Accounts; }
    }

    public Task<IEnumerable<Contact>> SearchAsync(
        string filter,
        FilterKind filterKind)
    {
        var tcs = new TaskCompletionSource<IEnumerable<Contact>>();

        this.contacts.SearchAsync(filter: filter, filterKind: filterKind, state: tcs);

        return tcs.Task;
    }
}
```

Resources

- Asynchronous Programming with Async and Await (C# and Visual Basic)
- Task-based Asynchronous Pattern (TAP)
- Event-based Asynchronous Pattern (EAP)
- Asynchronous Programming Model (APM)
- Async for .NET Framework 4, Silverlight 4 and 5, and Windows Phone 7.5 and 8 (needed for using *async/await* in Windows Phone 7.5 and 8)
- Three Essential Tips For Async - Introduction video by Lucian Wischik
- async by Eric Lippert
- Eduasync by Jon Skeet
- An Async Primer

(C) Copyright Nokia 2014. All rights reserved.

- Best Practices in Asynchronous Programming (MSDN Magazine > March 2013 Issue >)

(C) Copyright Nokia 2014. All rights reserved.