Make me async

Почти месяц назад я написал статью «Эволюция сервисных методов». В ней я пришел к реализации библиотеки MakeMeAsync, которая позволяет декорировать вызовы методов, делая их асинхронными. При этом код остаётся максимально читабельным и чистым, на мой взгляд.

Вот уже месяц как библиотека используется в реальном боевом проекте и показывает себя с лучшей стороны. Сначала удалось сократить размер самих сервисов, так как большая часть кода была инфраструктурной, затем, когда код стал ясным, я легко увидел, что сервисы не в полной мере выполняют свои задачи, а скорее делегируют их на слой ниже. Это меня огорчило, так как реализация является неправильным подходом; но порадовало, так как я смог это увидеть, и мне было психологически легче перенести код в нужное место, так как сервис выглядел компактным и понятным.

Кроме появления более четкой организации логики кода, значительно облегчилась задача по отображению ошибок, которые могли появиться во время выполнения задач в асинхронном режиме. Мне не надо было для каждой задачи прописывать/указывать метод обработки ошибки, можно было задать такой обработчик один раз в начале работы программы и все.

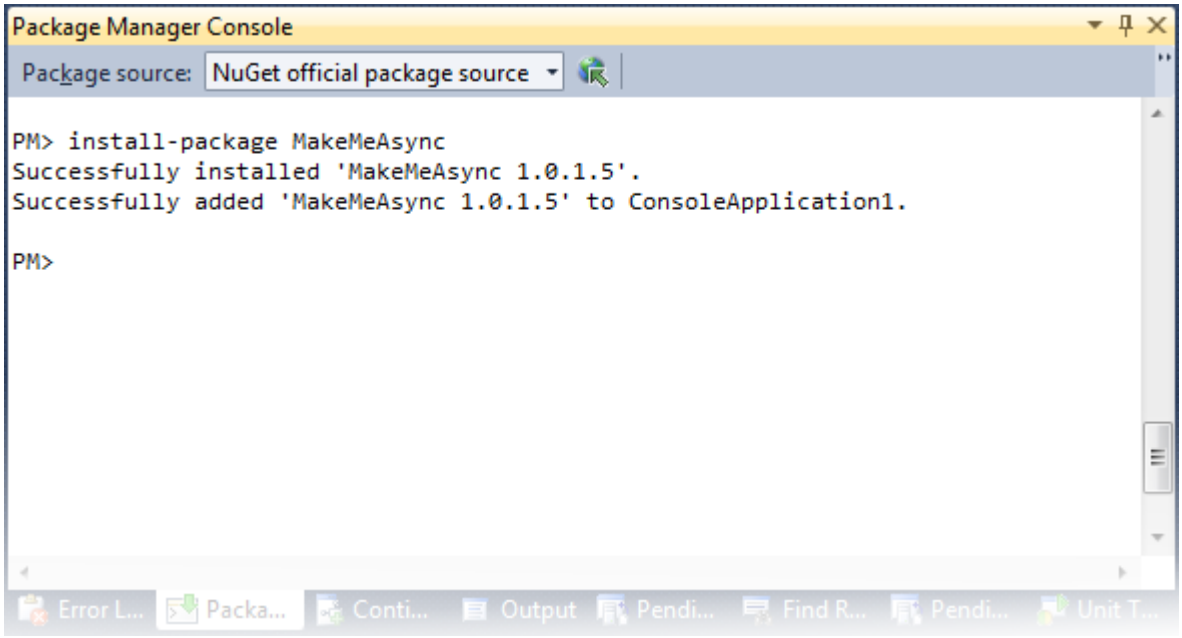
Итак, я думаю, что для начала стоит обновить информацию о том, как используется библиотека MakeMeAsync.

Установка

Установка библиотеки в проект происходит уже самым привычным для меня способом – с помощью пакетного менеджера NuGet. Если вы еще не прониклись его удобством, то советую обратить более пристальное внимание на его использование, особенно в свете последних дополнений.

Для установки, в консоли NuGet требуется набрать

```
Install-package MakeMeAsync
```



Всё, можно приступать к работе.

Возможности

В идеале я хочу писать простые сервисы, я не хочу думать об обвязках с многопоточностью, логированием, разграничением доступа, анонимных классах, динамических переменных и всем прочем. Хочется написать метод:

```
public class Service {
     public void Method() {
         // do some complicated actions
     }
}
```

И в клиенте сделать что-то в духе:

```
public void x() {
    service = new Service();
    service.Method().Async();
}
```

Общий вызов

Хех, однако ясно, что такой способ не пройдет ни при каких обстоятельствах, и **Method()** придется передавать в какой-либо метод и из этого выросло следующее (вдохновение мне принесла библиотека Rx):

```
public void x() {
    var service = new Service();

    Async.For(service.Method)
        .Subscribe(OnComplete)
        .Run();
}

private void OnComplete(Task task) {}
```

Ну не красота ли? Компактный исходный сервис, компактный вызов.

Обработка ошибок

Для обработки событий можно использовать метод **OnFault()**.

```
public void x() {
    var service = new Service();

    Async.OnFault(OnFault);

    Async.For(service.Method)
        .Subscribe(OnComplete)
        .Run();
}

private void OnFault(Task obj) {
}
```

При этом он будет использован для всех методов, которые будут вызваны асинхронно. Можете думать об этом как о работе с AppDomain.CurrentDomain.UnhandledException. Для каждого метода есть возможность переопределить поведение в случае возникновения ошибки.

```
   Async.For(service.Method)
        .Subscribe(OnComplete, OnFault)
        .Run();
```

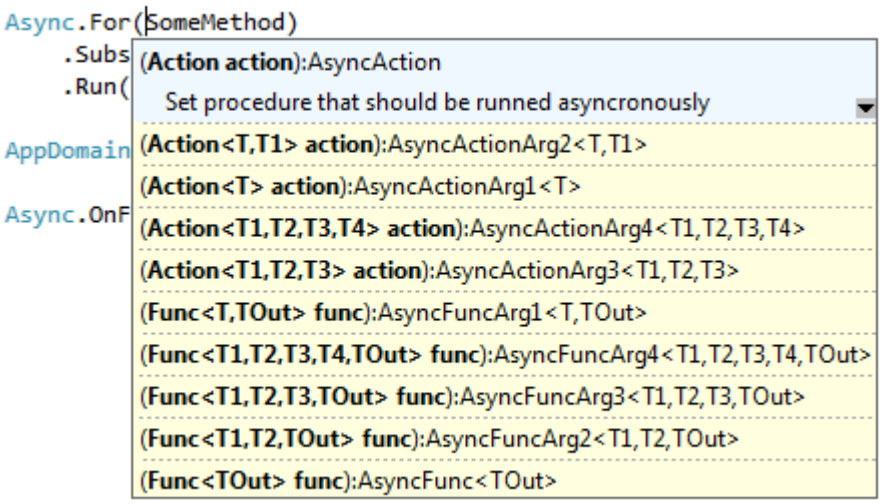
Обновление UI в заключительном событии

Так же периодически все попадаются на том, что хотят обновить содержимое UI в событии OnComplete. И теперь стало гораздо проще контролировать такие ситуации:

```
   Async.For(service.Method)
        .SubscribeOnDispatcher(OnComplete, OnFault)
        .Run();
```

Параметризованные вызовы

Закономерно наверно возникает вопрос с передачей параметров и с возвращением значений. Это не составляет большого труда, а метод **Run()** контролирует типизацию. Максимальное количество аргументов для функции на данный момент ограничено **четырьмя**. Это разумный максимум аргументов на мой взгляд.




```
var service = new Service();

var info = Async.For<int, string>(service.ProcedureIntStr)
                .Run(4, "s");
```

Метод Run() возвращает специальный класс, который содержит Task, подготовленные результаты, если есть, а так же готовую коллекцию ошибок, если таковые возникли.

Отложенная подписка

Если какой-то метод работает очень долго и запустить его надо как можно раньше, а результаты получить по мере необходимости, но можно указать это при запуске метода. Это не обходимо делать явно, так как все подписки обнуляются после завершения работы и передачи управления клиенту.

```
var info = Async.For(service.LongAction1Sec)
    .WithDelayedSubscription()
    .Run();

Async.Subscribe(info, OnCompleted);
```

Итого

Итак, общие плюсы такого решения:

- Меньше кода – меньше банальных ошибок
- На надо вводить анонимные классы для сервисов
- Легче тестировать сервисы
- Возможность указать единую реакцию на исключения
- Легче читать код
- В целом само отписывается от себя
- Callback возможно выполнить в потоке UI

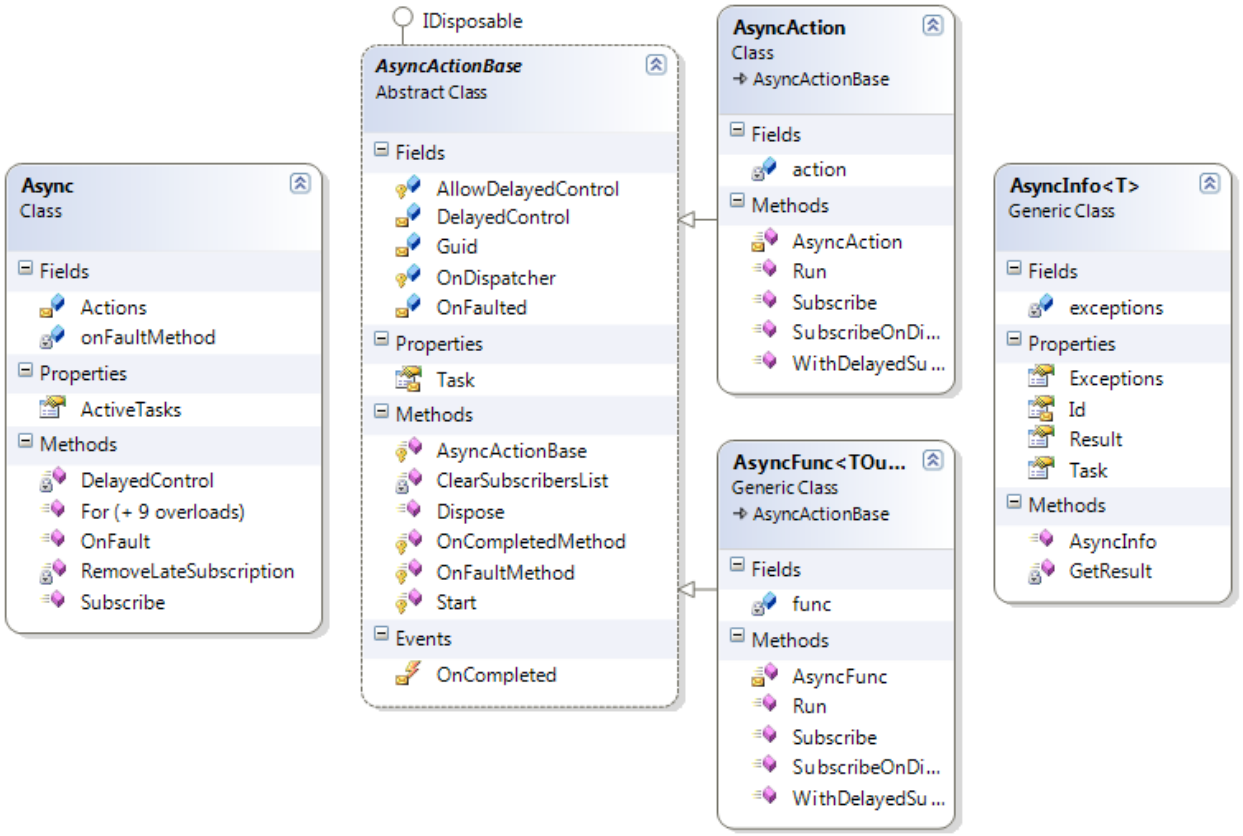
До этого я писал, что в минусах может значиться отсутствие CancellationToken, но подумав здраво пришел к выводу что для общего случая такая поддержка не нужна в силу самой природы/механизма отмена операции. Для того чтобы отмена операции работала, надо явно

прописать ее работу в теле метода. Однозначно определить в методе когда опрашивать CancellationToken для того чтобы увидеть сигнал отмены. Если этого не делать, то асинхронный метод не завершится волшебным образом по выдаче сигнала. Подача сигнала отмены это не операция kill для процесса. И так как поддержка CancellationToken должна быть явной, то потребуется передавать его либо параметром, и тогда все есть для приема любых аргументов, либо CancellationToken определен полем класса, и тогда от библиотеки MakeMeAsync ничего не зависит в плане отмены операции.

Можно конечно пофантазировать на тему распознавания параметров и специального указания на нужные CancellationToken, чтобы можно было бы вызвать метод в духе Async.CancelAll(), но положа руку на сердце признаемся, что такое маловероятно. Разве только спортивный интерес может сподвигнуть к реализации данного функционала.

Реализация

Для краткости сразу приведу диаграмму классов



Точкой входа является класс Async, с помощью него составляется все выражение для последующего выполнения. Результатом будет AsyncInfo. Неявными для пользователя являются классы формируемые для выполнения асинхронно, это наследники AsyncActionBase, коих на данный момент 9 штук.

Общий механизм выглядит следующим образом:

- Пользователь, с помощью класса Async и метода For() указывает, какой метод необходимо выполнить асинхронно.

- Класс Async создает новый класс-обертку AsyncActionArgX или AsyncFuncArgX, где X количество аргументов в переданном методе.
- Класс Async так же инициализирует все дополнительные подписки для класса-обертки.
- Класс Async сохраняет созданный объект во внутренней коллекции для дальнейшего управления и возвращает его пользователю.
- Пользователь, получив объект, указывает, что вызвать по завершении работы асинхронного метода с помощью методов Subscribe() и SubscribeOnDispatcher().
- Пользователь говорит о запуске асинхронного метода с помощью Run() передавая аргументы для метода, который будет выполняться асинхронно.

Основной сценарий

По большому счету класс Async написан не сложно, но в нем должны быть перечислены все перегрузки метода **For()** для разного количества параметров, как для процедур, так и для функций. Покажу сразу, как идет реализация для метода с параметрами. Увеличение или уменьшение параметров в дальнейшем дело механическое:

```
public static AsyncActionArg2<T, T1> For<T, T1>(Action<T, T1> action) {
    var asyncAction = new AsyncActionArg2<T, T1>(action) {
                                                    OnFaulted = onFaultMethod,
                                                    DelayedControl = DelayedControl
                                                          };
    Actions.Add(asyncAction.Guid, asyncAction);
    return asyncAction;
}
```

Здесь мы видим, как создается класс-обертка AsyncActionArg2, которому в конструктор передается метод пользователя. Так же указывается метод для реакции на ошибки onFaultMethod, и метод для поздней/отложенной подписки. Из описания по использованию можно догадаться, что onFaultMethod инициализируется статическим методом OnFault().

```
public static void OnFault(Action<Task> onFault) {
    onFaultMethod = onFault;
}
```

Не стал придумывать никаких проверок для входящего параметра, так что можно наверно стать самому себе злобным буратино.

Далее стоит рассмотреть класс AsyncActionArg2 и родительский класс AsyncActionBase. Строго говоря все самое интересное происходит в базовом классе, в то время как дочерние классы являются просто настройкой базового класса с учетом количества аргументов, так как суть работы не меняется от аргументов.

Для примера привожу полный листинг класса AsyncActionArg2. Остальное классы являются копией с точностью до количества аргументов.

```
public class AsyncActionArg2<T, T1> : AsyncActionBase {
    private readonly Action<T, T1> action;

    internal AsyncActionArg2(Action<T, T1> action) {
        this.action = action;
    }

    public AsyncActionArg2<T, T1> WithDelayedSubscription() {
        AllowDelayedControl = true;
        return this;
    }
```

```
    public AsyncActionArg2<T, T1> Subscribe(Action<Task> onCompleted, Action<Task>
onFaulted = null) {
        OnFaulted = onFaulted;
        OnCompleted += onCompleted;
        return this;
    }

    public AsyncActionArg2<T, T1> SubscribeOnDispatcher(Action<Task> onCompleted,
Action<Task> onFaulted = null) {
        OnDispatcher = true;
        return Subscribe(onCompleted, onFaulted);
    }

    public AsyncInfo<object> Run(T arg1, T1 arg2) {
        v var task = new Task(() => action(arg1, arg2));
        Start(task);
        return new AsyncInfo<object>(task, Guid);
    }
}
```

Как видно из реализации, никакой реальной работы в данном классе фактически не производится, идет простая настройка. Мне кажется даже пояснять ничего не надо. Единственное что создается непосредственно экземпляр класса Task, так как только в дочерних классах известны количество и тип параметров для лучшей работы IntelliSense.

Так и дошли до самого интересного, до класса AsyncActionBase. Данный класс осуществляет нужные подписки, получает контексты работы откликов, генерирует уникальный Guid для возможности отложенной подписки, осуществляет отписки от событий.

Самым интересным методом в данном случае является метод Start().

```
protected void Start(Task task) {
    if(!AllowDelayedControl && DelayedControl != null)
        DelayedControl(Guid);

    Task = task;
    var ui = OnDispatcher
               ? TaskScheduler.FromCurrentSynchronizationContext()
               : TaskScheduler.Default;
    task.ContinueWith(OnCompletedMethod, CancellationToken.None,
TaskContinuationOptions.NotOnFaulted, ui);
    task.ContinueWith(OnFaultMethod, CancellationToken.None,
TaskContinuationOptions.OnlyOnFaulted, ui);
    task.Start();
}
```

Собственно здесь обрабатываются выставленные программистом опции работы и соответствующим образом строится дальнейшая работа экземпляра класса Task. В первом условии удаляется информация о запускаемом действии из класса Async.

Методы OnCompletedMethod() и OnFaultMethod() сделаны по одному принципу и здесь специально не передаются события, которые указал пользователь, так как следующая реализация оказывается проще:

```
protected void OnCompletedMethod(Task task) {
    if (OnCompleted != null) {
```

```
            OnCompleted(task);
            ClearSubscribersList();
        }
    }

    protected void OnFaultMethod(Task task) {
        if (OnFaulted != null) {
            OnFaulted(task);
            ClearSubscribersList();
        }
    }
```

А проще она из-за того, позволяет единообразно построить код вне зависимости от того, будет ли отложенная подписка на события или нет. Если бы я сразу передал события пользователя в ContinueWith(), то мне пришлось бы в дочерних методах дополнительно анализировать опцию отложенной подписки в момент создания экземпляра класса Task и задавать явно опцию TaskCreationOptions.LongRunning. Меньше кода - меньше ошибок.

Так же можно увидеть, что происходит самостоятельная отписка от всех событий после того, как все подписчики были уведомлены об окончании работы метода.

Отложенная подписка

При отложенной подписке в целом выполняются все те же самые действия. Отличиями будут только более поздняя отписка и более долгое хранение дочерних классов.

Отложенная подписка на события осуществляется по классу информации, который получает пользователь после вызова метода Run(). Полученный объект будет являться идентификатором по которому можно будет осуществить дополнительную подписку.

```
public static void Subscribe(AsyncInfo<object> info, Action<Task> onCompleted) {
    if (info == null || info.Task == null) {
        throw new NullReferenceException("passed info or task should be not null");
    }

    if (!Actions.ContainsKey(info.Id)) {
        throw new InvalidOperationException("Unable subscribe, check that you uses
WithDelayedSubscription(). For more information see docs");
    };

    if (info.Task.Status != TaskStatus.RanToCompletion) {
        var asyncActionBase = Actions[info.Id];
        asyncActionBase.OnCompleted += onCompleted;
        asyncActionBase.OnCompleted += RemoveLateSubscription;
    }
    else {
        onCompleted(info.Task);
        Actions.Remove(info.Id);
    }
}

private static void RemoveLateSubscription(Task task) {
    var asyncActionBase = Actions.Values.SingleOrDefault(i => i.Task == task);
    if (asyncActionBase == null) return;

    asyncActionBase.OnCompleted -= RemoveLateSubscription;
    Actions.Remove(asyncActionBase.Guid);
}
```

Если метод все еще работает, то добавляем подписку, если же метод отработал, то сразу возвращаем результат работы метода и отписываем всех интересующихся.

Честно сказать, сложно рассказать последовательно и линейно совсем не линейную и не одноуровневую структуру организации классов. Да и рассказ этот скорее о том, что все не так сложно как может показаться на первый взгляд и что стоит дерзать и реализовывать свои задумки в любом виде.

Как я уже написал в начале, практика использования библиотеки показывает, что она жизнеспособна и приносит вполне определенную и конкретную пользую в боевом, реальном проекте. Код стал более лаконичным и позволил увидеть некоторые другие проблемы организации кода, которые раньше скрывались за обилием инфраструктурного кода.

Hard’n’heavy!

Violet Tape