Task-based Asynchronous Pattern with async and await

Mariana Goranova, Elena Kalcheva-Yovkova, Stanimir Penkov
Technical University of Sofia, Sofia, Bulgaria,
mgor@tu-sofia.bg, elena@tu-sofia.bg, spenkov@tu-sofia.bg

***Abstract**: Asynchronous programming enhances the overall responsiveness of applications and helps to avoid bottlenecks. .NET Framework supports a simplified approach, async programming that gets all the benefits of traditional asynchronous programming but with less efforts from developer. The aim of this paper is to study the power of the Task-based Asynchronous programming model and to demonstrate the C# language support for asynchronous programming. The paper illustrates the main advantages of asynchronous programming with an example of long-running calculation over sequence of values.*

***Keywords:** asynchronous programing model, event-based asynchronous pattern, task-based asynchronous pattern.*

1. INTRODUCTION

Let a program consists of conceptually distinct tasks. The single-threaded synchronous model is the simplest style of programming where each task is performed one at a time. When one task is finishing completely, another task is starting.

In the multi-threaded synchronous model each task is performed in a separate thread of control. The operating system manages the threads.

In the asynchronous model, the tasks are interleaved with one another, but in a single thread of control. When one task is executing, another task is not.

The asynchronous model has benefits when:

- There are a large number of tasks and there is always at last one task that can make progress.
- The tasks perform lots of I/O where a synchronous program will waste lots of time blocking when other tasks could be running.
- The tasks are largely independent from one to another so there is little need for inter-task communication.

Asynchronous programming is a powerful technique that enables us to more easily write scalable and responsive applications. When we write everything asynchronously we can achieve better system utilization and consume recourses only when they are actually needed for execution. The .NET framework provides asynchronous method implementation well optimized and good or better performance using existing patterns.

In this work, we examine the Task-based Asynchronous Programming model introduced with .NET 4 that makes asynchronous programming simpler to develop, understand, and maintain using **async** and **await** keywords.

2. RELATED WORK

The .NET Framework provides three design patterns for asynchronous operations:

- Asynchronous operations that use **IAsyncResult** objects and require **Begin** and **End** methods (Asynchronous Programming Model – APM) [1].

- Asynchronous operations that require a method with **Async** suffix and one or more events, event handler delegate types, and **EventArgs**-derived types (Event-based Asynchronous Pattern – EAP).
- Asynchronous operations that use tasks and require a single method to initiate and complete the operation (Task-based Asynchronous Pattern – TAP, using **async**/**await** statements – C#, version 5) [2, 3, 4, 5, 6, 7, 8, 9, 10].

APM and EAP are no longer recommended for new development. TAP is the recommended approach to asynchronous programming in the .NET Framework.

2.1. Class Task

TAP is based on the classes **Task<TResult>** and **Task** that are part of the Task Parallel Library (TPL). **Task<TResult>** represents the concept of some work that will produce a result of type **TResult** in the future. The non-generic **Task** class represents the concept of work that will complete in the future but returns no result. These types realize the basic concept of the TAP [5] – to represent asynchronous operations in a single method combining both the status of the operation and the interaction with these operations into a single object. Their method **Run(Action action)** queues the specified work to run on the thread pool and returns a task handle for that work, where **action** is a delegate encapsulating a method that has no parameters and does not return a value, i.e. the work to execute asynchronously.

2.2. Async/await Statements

In C# 5, the **async** and **await** modifiers were added to work with the **Task** class to make it significantly easier to write asynchronous code in our applications and so make the TAP method even more powerful [3].

The **async** keyword is a modifier to a method or anonymous method holding an asynchronous operation. It only enables the **await** keyword in that method and manages the method results. It does not run this method on a thread pool.

The **await** keyword invokes an asynchronous operation. It is like a unary operator that examines the status of the asynchronous operation. Await evaluates the *expression* to obtain an object representing work that will produce a *result* in the future and immediately returns the control to the caller. If the operation has already completed, the remaining *statements* are executed and the method just continues running synchronously like a regular method. If the operation has not completed, it acts asynchronously – the current method can't continue past that point until the asynchronous operation has completed.

```
var result = await expression;
statement/s;
```

The compiler replaces **await** with some code that sets up a special object – called the *awaiter* – to wait for the completion of the awaited operation. The generated code returns the control to the caller of the awaited method and the execution proceeds while we wait for the completion of the operation. When the operation completes, the generated code uses the track of location where execution was suspended and runs the remainder of the method. The method pauses until the operation is complete, but the actual thread is not blocked.

2.3. Asynchronous Method

The TAP model is now the recommended approach for asynchronous programming and .NET propose many classes with **async** methods that return **Task** or **Task<TResult>**. Fig. 1 shows how to convert a synchronous method to an asynchronous method using the TAP model. The asynchronous method has **async** as a modifier, includes a suffix **Async** in the method name and does not permit **out** and **ref** parameters. The return type is:

- **Task<TResult>**, if the corresponding synchronous method has **return** *expression*, where *expression* is of type **TResult**.
- **Task**, if the corresponding synchronous method does not have **return** *expression*.
- **void**, if the method is asynchronous event handler.

The asynchronous method includes at least one **await** *expression*.

```
void Method()
{
    //…
}

<TResult> Method()
{
    //…
    return <expression>;
}
```

Fig. 1.a: Sync method

```
async Task MethodAsync()
{
    var task = Task.Run( () => Method() );
    await task;
}

async Task<TResult> MethodAsync()
{
    var task = Task.Run( () => Method() );
    TResult result = await task;
}
```

Fig. 1.b: Converting a sync method to an async method

At the beginning the asynchronous method is executed just like any other method – it runs synchronously until it encounters **await** or throws an exception. The **await** keyword marks a point where the method can't continue until the awaited asynchronous operation is complete and the control returns to the method's caller. When the asynchronous operation completes the control returns to the marked point.

3. LONG-RUNNING OPERATION EXAMPLE

3.1. Calculation over a Sequence of Values

The extension generic method **Accumulate** [11] presents a general algorithm for accumulation of collection elements (Fig. 2). The **source** collection has to implement the **IEnumerable<T>** interface that provides the iteration of the collection. The specified **seed** value from the type **TAccumulate** is used as the initial accumulator value. The collection elements are from type **T** which does not have constrains for assignment and add operations, because C# does not provide generic operators. The accumulation operation **op** is represented with the **BinaryOperation<TAccumulate, T, TAccumulate>** delegate where the binary operation has two operands of type **T1** and **T2** correspondingly and returns a result of type **TResult**. The synchronous method **Accumulate** returns a result of type **TAccumulate**.

Depending on the source, this method can take a long time and needs to be converted as asynchronous method. One way to make it asynchronously is to create a method **AccumulateAsync** with **async** modifier and simply change the return type to **Task<TAccumulate>**. The method **PrintAccumulateAsync** is decorated by the keyword **async** and it calls **AccumulateAsync** asynchronously. When it calls, the execution immediately returns to the caller and once the long asynchronous operation is completed, it will get the control back (write the calculated sum over a sequence of values).

```
namespace AsyncAwaitExample
{
  public delegate TResult BinaryOperation<T1,T2,TResult>(T1 oper1,T2 oper2);

  public static class Accumulator
  {
    public static TAccumulate Accumulate<T, TAccumulate> (this IEnumerable<T> source,
                  TAccumulate seed, BinaryOperation<TAccumulate, T, TAccumulate> op)
    {
        TAccumulate acc = seed;            // initial accumulator value
        foreach (T item in source)         // for each element of collection
          acc = op(acc, item);             //      executes the operation and saves the accumulator value
        return acc;                           // return the accumulator value
    }

    public static async Task<TAccumulate> AccumulateAsync<T, TAccumulate>
                            (this IEnumerable<T> source, TAccumulate seed,
                             BinaryOperation<TAccumulate, T, TAccumulate> op)
    {
      var task = Task.Run( () => source.Accumulate<T, TAccumulate>(seed, op) );
      TAccumulate result = await task;
      return result;
    }

    public static async Task PrintAccumulateAsync<T, TAccumulate>
                           (this IEnumerable<T> source, TAccumulate seed,
                            BinaryOperation<TAccumulate, T, TAccumulate> op, string format)
    {
       var sumTask = source.AccumulateAsync<T, TAccumulate>(seed, op);
       TAccumulate sum = await sumTask;
       Console.WriteLine(format,sum);
    }
  }

  class Program
  {  public static void DoSomething()
    {  for (int i = 0; i < 50; i++)
          Console.Write(".");
      Console.WriteLine();
    }

    static void Main(string[] args)
    {
      int[] a       = { 1, 2, 3, 4, 5 };
      double[] b = { 1.0, 2.0, 3.0};
      var result1 = a.PrintAccumulateAsync<int, int>(0, (seed, element) => seed + element,
                                                         "1+2+3+4+5 = {0}");
      DoSomething();
      var result2 = b.PrintAccumulateAsync<double, double>(1.0, (seed, element) => seed * element,
                                                                  "1.0*2.0*3.0 = {0:F3}");
      DoSomething();
    }
  }
}
```

Fig. 2: Code example

3.2. Example Results

The following steps are executed:

- The **Main** method calls the asynchronous method **PrintAccumulateAsync** in the main thread.
- **PrintAccumulateAsync** calls the asynchronous method **AccumulateAsync** to calculate the sum of elements of the array **a**.
- **AccumulateAsync** starts the task using

  ```
  var task = Task.Run( () => source.Accumulate<T, TAccumulate>(seed, op) );
  ```

  This task represents a long operation **Accumulate** and will run in the working thread in the thread pool. A thread pool is a collection of threads that can be used to perform several tasks in the background and the primary thread is free to perform other tasks asynchronously. **AccumulateAsync** returns the control to the **PrintAccumulateAsync** to avoid blocking the recourses. **AccumulateAsync** returns a **Task<TAccumulate>** where **TAccumulate** is an integer, and **PrintAccumulateAsync** assigns the task to the **sumTask** variable. The task will produce an actual integer when the work is complete.
- **PrintAccumulateAsync** can't continue until the awaited **AccumulateAsync** is complete and with **await** returns the control to **Main**. **Main** calls the synchronous method **DoSomething**. The control remains in **PrintAccumulateAsync**, if **AccumulateAsync** completes before awaiting.
- When the long operation finishes, the working thread in the pool completes its task and it is returned to a queue of waiting threads, where it can be reused. The result of the long-running operation is saved in the task **sumTask** of **PrintAccumulateAsync** and the result is received with **await** in **sum**.
- **PrintAccumulateAsync** prints the calculated **sum**.
- **PrintAccumulateAsync** ends and returns the control to **Main**.
- All steps are repeated when the **Main** method calls the asynchronous method **PrintAccumulateAsync** to calculate the product of elements of the array **b**.

Fig. 3 shows the program output:

```
.1+2+3+4+5 = 15
 ................................................
 .....................................1.0*2.0*3.0 = 6.000
 ............
```

Fig. 3: Sample of results

In this example, the program handles long-running operations without blocking the recourses using TAP. The calculation over sequence of values needs long time, that's why the asynchronous method **AccumulateAync** returns the control to the **PrintAccumulateAsync**, **PrintAccumulateAsync** can't continue and returns the control to **Main**. **Main** proceeds the execution calling **DoSomething** after the asynchronous task has started. When the awaited long operation completes, the remainder code of **PrintAccumulateAsync** prints the calculated sum of elements of the array **a**. The behavior of the program is similarly when **Main** calls **PrintAccumulateAsync** to print the product of elements of the array **b**.

The task representing long-running operation doesn't create additional thread, because an async method doesn't run on its own thread. It only uses the thread pool to complete the task. The system manages all tasks and the thread pool. That's why TAP is being more efficient and effective for asynchronous program.

4. CONCLUSIONS

Performing asynchronous operations is the key to building scalable, responsive and maintainability applications. When coupled with the thread pool, asynchronous operations allow us to take advantage of all the CPUs in the machine [12]. The Task-based Asynchronous Pattern designed by Microsoft makes it easy for developers to take advantage of these capabilities. This pattern uses the **async** and **await** keywords that simplify the programming in .NET. They allow writing an asynchronous program almost as if it is a usual synchronous program.

In this paper we discussed the TAP and **async**/**await** features for creating asynchronous functionality that is almost as simple as the synchronous code. The new trend is to bring the simplicity of synchronous programming models to this asynchronous programming paradigm. Our future work will study the pattern defined to transition from Event-based Asynchronous Pattern to TAP.

Acknowledgements

This work is supported by the Scientific and Research Sector of the Technical University of Sofia, Bulgaria research project 152PD0021-09/2015.

5. REFERENCES

[1] Goranova, M. (2004) Asynchronous Programming with Callbacks in .NET Framework, Proceedings of the International Conference of Computer Science, Sofia, Bulgaria, 210-215.
[1] Task-based Asynchronous Pattern (TAP), https://msdn.microsoft.com/en-us/library/hh873175.aspx.
[2] Asynchronous Programming with Async and Await (C# and Visual Basic), https://msdn.microsoft.com/en-us/library/hh191443.aspx.
[3] Toub, S. (2012), Task-based Asynchronous Pattern, Microsoft, http://www.microsoft.com/en-us/download/details.aspx?id=19957.
[4] Marini, D. (2014) Improving Your Asynchronous Code Using Tasks, Async and Await, http://www.infoq.com/articles/Tasks-Async-Await.
[5] Cleary, S. (2013), Async and Await, http://blog.stephencleary.com/2012/02/async-and-await.html.
[6] Clearly, S. (2013) Best Practices in Asynchronous Programming, *MSDN Magazine*, No 3, https://msdn.microsoft.com/en-us/magazine/jj991977.aspx.
[7] Clearly, S. (2014) Concurrency in C# Cookbook, O'Reily.
[8] Assil (2014) .NET Asynchronous Patterns, http://www.codeproject.com/Articles/646239/NET-Asynchronous-Patterns.
[9] Rappl, F. (2013) Asynchronous models and patterns, http://www.codeproject.com/Articles/562021/Asynchronous-models-and-patterns.
[10] Goranova, M. (2009) Generic Programming in C#. *Elektrotechnica & Elektronica* (ISSN 0861-4717) No 11-12, 35-42.
[11] Richter, J. (2010) CLR via C#, Microsoft Press.