Task-based Asynchronous Pattern (TAP)

.NET Framework 4.5

The Task-based Asynchronous Pattern (TAP) is based on the System.Threading.Tasks.Task and System.Threading.Tasks.Task<TResult> types in the System.Threading.Tasks namespace, which are used to represent arbitrary asynchronous operations. TAP is the recommended asynchronous design pattern for new development.

Naming, Parameters, and Return Types

TAP uses a single method to represent the initiation and completion of an asynchronous operation. This is in contrast to the Asynchronous Programming Model (APM or **IAsyncResult**) pattern, which requires `Begin` and `End` methods, and in contrast to the Event-based Asynchronous Pattern (EAP), which requires a method that has the `Async` suffix and also requires one or more events, event handler delegate types, and **EventArg**-derived types. Asynchronous methods in TAP include the `Async` suffix after the operation name; for example, `GetAsync` for a get operation. If you're adding a TAP method to a class that already contains that method name with the `Async` suffix, use the suffix `TaskAsync` instead. For example, if the class already has a `GetAsync` method, use the name `GetTaskAsync`.

The TAP method returns either a System.Threading.Tasks.Task or a System.Threading.Tasks.Task<TResult>, based on whether the corresponding synchronous method returns void or a type **TResult**.

The parameters of a TAP method should match the parameters of its synchronous counterpart, and should be provided in the same order. However, *out* and *ref* parameters are exempt from this rule and should be avoided entirely. Any data that would have been returned through an *out* or *ref* parameter should instead be returned as part of the **TResult** returned by Task<TResult>, and should use a tuple or a custom data structure to accommodate multiple values. Methods that are devoted exclusively to the creation, manipulation, or combination of tasks (where the asynchronous intent of the method is clear in the method name or in the name of the type to which the method belongs) need not follow this naming pattern; such methods are often referred to as *combinators*. Examples of combinators include WhenAll and WhenAny, and are discussed in the Using the Built-in Task-based Combinators section of the article Consuming the Task-based Asynchronous Pattern.

For examples of how the TAP syntax differs from the syntax used in legacy asynchronous programming patterns such as the Asynchronous Programming Model (APM) and the Event-based Asynchronous Pattern (EAP), see Asynchronous Programming Patterns.

Initiating an Asynchronous Operation

An asynchronous method that is based on TAP can do a small amount of work synchronously, such as validating arguments and initiating the asynchronous operation, before it returns the resulting task. Synchronous work should be kept to the minimum so the asynchronous method can return quickly. Reasons for a quick return include the following:

- Asynchronous methods may be invoked from user interface (UI) threads, and any long-running synchronous work could harm the responsiveness of the application.
- Multiple asynchronous methods may be launched concurrently. Therefore, any long-running work in the synchronous portion of an asynchronous method could delay the initiation of other asynchronous operations, thereby decreasing the benefits of concurrency.

In some cases, the amount of work required to complete the operation is less than the amount of work required to launch the operation asynchronously. Reading from a stream where the read operation can be satisfied by data that is already buffered in memory is an example of such a scenario. In such cases, the operation may complete synchronously, and may return a task that has already been completed.

Exceptions

An asynchronous method should raise an exception to be thrown out of the asynchronous method call only in response to a usage error. Usage errors should never occur in production code. For example, if passing a null reference (**Nothing** in Visual Basic) as one of the method's arguments causes an error state (usually represented by an ArgumentNullException exception), you can modify the calling code to ensure that a null reference is never passed. For all other errors, exceptions that occur when an asynchronous method is running should be assigned to the returned task, even if the asynchronous method happens to complete synchronously before the task is returned. Typically, a task contains at most one exception. However, if the task represents multiple operations (for example, WhenAll), multiple exceptions may be associated with a single task.

Target Environment

When you implement a TAP method, you can determine where asynchronous execution occurs. You may choose to execute the workload on the thread pool, implement it by using asynchronous I/O (without being bound to a thread for the majority of the operation's execution), run it on a specific thread (such as the UI thread), or use any number of potential contexts. A TAP method may even have nothing to execute, and may just return a Task that represents the occurrence of a condition elsewhere in the system (for example, a task that represents data arriving at a queued data structure).The caller of the TAP method may block waiting for the TAP method to complete by synchronously waiting on the resulting task, or may run additional (continuation) code when the asynchronous operation completes. The creator of the continuation code has control over where that code executes. You may create the continuation code either explicitly, through methods on the Task class (for example, ContinueWith) or implicitly, by using language support built on top of continuations (for example, **await** in C#, **Await** in Visual Basic, **AwaitValue** in F#).

Task Status

The Task class provides a life cycle for asynchronous operations, and that cycle is represented by the TaskStatus enumeration. To support corner cases of types that derive from Task and Task<TResult>, and to support the separation of construction from scheduling, the Task class exposes a Start method. Tasks that are created by the public Task constructors are referred to as *cold tasks*, because they begin their life cycle in the non-scheduled Created state and are scheduled only when Start is called on these instances. All other tasks begin their life cycle in a hot state, which means that the asynchronous operations they represent have already been initiated and their task status is an enumeration value other than TaskStatus.Created. All tasks that are returned from TAP methods must be activated. If a TAP method internally uses a task's constructor to instantiate the task to be returned, the TAP method must call Start on the Task object before returning it. Consumers of a TAP method may safely assume that the returned task is active and should not try to call Start on any Task that is returned from a TAP method. Calling Start on an active task results in an InvalidOperationException exception.

Cancellation (Optional)

In TAP, cancellation is optional for both asynchronous method implementers and asynchronous method consumers. If an operation allows cancellation, it exposes an overload of the asynchronous method that accepts a cancellation token (CancellationToken instance). By convention, the parameter is named *cancellationToken*.

C#

```
public Task ReadAsync(byte [] buffer, int offset, int count,
                      CancellationToken cancellationToken)
```

The asynchronous operation monitors this token for cancellation requests. If it receives a cancellation request, it may choose to honor that request and cancel the operation. If the cancellation request results in work being ended prematurely, the TAP method returns a task that ends in the Canceled state; there is no available result and no exception is thrown. The Canceled state is considered to be a final (completed) state for a task, along with the Faulted and RanToCompletion states. Therefore, if a task is in the Canceled state, its IsCompleted property returns **true**. When a task completes in the Canceled state, any continuations registered with the task are scheduled or executed, unless a continuation option such as NotOnCanceled was specified to opt out of continuation. Any code that is asynchronously waiting for a canceled task through use of language features continues to run but receives an OperationCanceledException or an exception derived from it. Code that is blocked synchronously waiting on the task through methods such as Wait and WaitAll also continue to run with an exception.

If a cancellation token has requested cancellation before the TAP method that accepts that token is called, the TAP method should return a Canceled task. However, if cancellation is requested while the asynchronous operation is running, the asynchronous operation need not accept the cancellation request. The returned task should end in the Canceled state only if the operation ends as a result of the cancellation request. If cancellation is requested but a result or an exception is still produced, the task should end in the RanToCompletion or Faulted state. For asynchronous methods used by a developer who wants cancellation first and foremost, you don't have to provide an overload that doesn't accept a cancellation token. For methods that cannot be canceled, do not provide overloads that accept a cancellation token; this helps indicate to the caller whether the target method is actually cancelable. Consumer code that does not desire cancellation may call a method that accepts a CancellationToken and provide None as the argument value. None is functionally equivalent to the default CancellationToken.

Progress Reporting (Optional)

Some asynchronous operations benefit from providing progress notifications; these are typically used to update a user interface with information about the progress of the asynchronous operation. In TAP, progress is handled through an IProgress<T> interface, which is passed to the asynchronous method as a parameter that is usually named *progress*. Providing the progress interface when the asynchronous method is called helps eliminate race conditions that result from incorrect usage (that is, when event handlers that are incorrectly registered after the operation starts may miss updates). More importantly, the progress interface supports varying implementations of progress, as determined by the consuming code. For example, the consuming code may only care about the latest progress update, or may want to buffer all updates, or may want to invoke an action for each update, or may want to control whether the invocation is marshaled to a particular thread. All these options may be achieved by using a different implementation of the interface, customized to the particular consumer's needs. As with cancellation, TAP implementations should provide an IProgress<T> parameter only if the API supports progress notifications. For example, if the `ReadAsync` method discussed earlier in this article is able to report intermediate progress in the form of the number of bytes read thus far, the progress callback could be an IProgress<T> interface:

C#

```
public Task ReadAsync(byte[] buffer, int offset, int count,
                      IProgress<long> progress)
```

If a `FindFilesAsync` method returns a list of all files that meet a particular search pattern, the progress callback could provide an estimate of the percentage of work completed as well as the current set of partial results. It could do this either with a tuple:

C#

```
public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(
            string pattern,
            IProgress<Tuple<double,
            ReadOnlyCollection<List<FileInfo>>>> progress)
```

or with a data type that is specific to the API:

C#

```
public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(
    string pattern,
    IProgress<FindFilesProgressInfo> progress)
```

In the latter case, the special data type is usually suffixed with `ProgressInfo`.

If TAP implementations provide overloads that accept a *progress* parameter, they must allow the argument to be **null**, in which case no progress will be reported. TAP implementations should report the progress to the Progress<T> object synchronously, which enables the asynchronous method to quickly provide progress, and allow the consumer of the progress to determine how and where best to handle the information. For example, the progress instance could choose to marshal callbacks and raise events on a captured synchronization context.

IProgress<T> Implementations

The .NET Framework 4.5 provides a single IProgress<T> implementation: Progress<T>. The Progress<T> class is declared as follows:

C#

```
public class Progress<T> : IProgress<T>
{
    public Progress();
    public Progress(Action<T> handler);
    protected virtual void OnReport(T value);
    public event EventHandler<T> ProgressChanged;
}
```

An instance of Progress<T> exposes a ProgressChanged event, which is raised every time the asynchronous operation reports a progress update. The ProgressChanged event is raised on the SynchronizationContext object that was captured when the Progress<T> instance was instantiated. If no synchronization context was available, a default context that targets the thread pool is used. Handlers may be registered with this event. A single handler may also be provided to the Progress<T> constructor for convenience, and behaves just like an event handler for the ProgressChanged event. Progress updates are raised asynchronously to avoid delaying the asynchronous operation while event handlers are executing. Another IProgress<T> implementation could choose to apply different semantics.

Choosing the Overloads to Provide

If a TAP implementation uses both the optional CancellationToken and optional IProgress<T> parameters, it could potentially require up to four overloads:

C#

```
public Task MethodNameAsync(…);
public Task MethodNameAsync(…, CancellationToken cancellationToken);
public Task MethodNameAsync(…, IProgress<T> progress);
public Task MethodNameAsync(…,
    CancellationToken cancellationToken, IProgress<T> progress);
```

However, many TAP implementations provide neither cancellation or progress capabilities, so they require a single method:

C#

```
public Task MethodNameAsync(…);
```

If a TAP implementation supports either cancellation or progress but not both, it may provide two overloads:

C#

```
public Task MethodNameAsync(…);
public Task MethodNameAsync(…, CancellationToken cancellationToken);

// … or …

public Task MethodNameAsync(…);
public Task MethodNameAsync(…, IProgress<T> progress);
```

If a TAP implementation supports both cancellation and progress, it may expose all four overloads. However, it may provide only the following two:

C#

```
public Task MethodNameAsync(…);
public Task MethodNameAsync(…,
    CancellationToken cancellationToken, IProgress<T> progress);
```

To compensate for the two missing intermediate combinations, developers may pass None or a default CancellationToken for the *cancellationToken* parameter and **null** for the *progress* parameter.

If you expect every usage of the TAP method to support cancellation or progress, you may omit the overloads that don't accept the relevant parameter.

If you decide to expose multiple overloads to make cancellation or progress optional, the overloads that don't support cancellation or progress should behave as if they passed None for cancellation or **null** for progress to the overload that does support these.

Related Topics

Title	**Description**
Asynchronous Programming Patterns	Introduces the three patterns for performing asynchronous operations: the Task-based Asynchronous Pattern (TAP), the Asynchronous Programming Model (APM), and the Event-based Asynchronous Pattern (EAP).
Implementing the Task-based Asynchronous Pattern	Describes how to implement the Task-based Asynchronous Pattern (TAP) in three ways: by using the C# and Visual Basic compilers in Visual Studio, manually, or through a combination of the compiler and manual methods.
Consuming the Task-based Asynchronous Pattern	Describes how you can use tasks and callbacks to achieve waiting without blocking.
Interop with Other Asynchronous Patterns and Types	Describes how to use the Task-based Asynchronous Pattern (TAP) to implement the Asynchronous Programming Model (APM) and Event-based Asynchronous Pattern (EAP).