Understanding Asynchronous Programming in C#

Jeremy Clark
www.jeremybytes.com
@jeremybytes

Materials

https://github.com/jeremybytes/async-workshop-2021

Schedule

• Class Hours	1:30 p.m. – 4:30 p.m.
• Break	2:30 p.m. – 2:40 p.m.
• Break	3:30 p.m. – 3:40 p.m.

Q&A after the breaks

All Times are Eastern Daylight Time

Agenda 1

- Calling async methods with Task
- "await"ing async methods
- Getting Results
- Continuing after async is complete
- Dealing with Exceptions
- Cancellation

Agenda 2

- Writing async methods
  - Task.Run()
  - "await" inside async methods
  - Return values
  - Cancellation
- Running code in Parallel
  - Using Tasks directly

Topics (in no particular order)

- ConfigureAwait()
- Action
- Lambda expressions
- TaskContinuationOptions
- async/await
- CancellationToken.None
- ThrowIfCancellationRequested
- async void
- Task.Run()
- Task.Result
- GetAwaiter().GetResult()
- IsFaulted, IsCanceled, IsCompleted
- AggregateException
- OperationCanceledException
- CancellationTokenSource
- async MVC Controllers
- TaskFactory.FromAsync()

Running Asynchronous Code

Asynchronous Patterns

- Asynchronous Programming Model (APM)
- Event Asynchronous Pattern (EAP)
- Task Asynchronous Pattern (TAP)

Asynchronous Programming Model (APM)

- Method-Based
- Methods
  - IAsyncResult BeginGetData()
  - EndGetData(IAsyncResult …)
- IAsyncResult

Event Asynchronous Pattern (EAP)

- Method/Event-Based
- Method
  - GetDataAsync()
- Event
  - GetDataCompleted
  - Results in EventArgs

Task Asynchronous Pattern (TAP)

- Task-Based
- Method Returns a Task
  - Task<T> GetDataAsync()
  - "T" = result type for the Task
- Task
  - Represents a concurrent operation
  - May or may not operate on a separate thread
  - Can be chained and combined

Desktop App Sample: Using Task

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
peopleTask.ContinueWith(
    task =>
    {
        List<Person> people = task.Result;
        foreach (var person in people)
            PersonListBox.Items.Add(person);
    }
```

Web App Sample: Using Task

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
Task<ViewResult> resultTask = peopleTask.ContinueWith(
    task =>
    {
        List<Person> people = task.Result;
        ViewData["RequestEnd"] = DateTime.Now;
        return View("Index", people);
    });
```

async & await

- Syntactic Wrapper Around Task
  - “await” pauses the current method until Task is complete
  - Looks like a blocking operation
  - Does not block current thread
- “async” is just a Hint
  - Does not make a method run asynchronously
  - Tells the compiler to treat “await” as noted above

Desktop App Sample: Using await

```
List<Person> people = await reader.GetPeopleAsync();
foreach (var person in people)
    PersonListBox.Items.Add(person);
```

Web App Sample: Using await

```
try
{
    List<Person> people = await reader.GetPeopleAsync();
    return View("Index", people);
}
finally
{
    ViewData["RequestEnd"] = DateTime.Now;
}
```

Task.Result 1

.Result

- Should only be used inside a continuation.
- If ".Result" is used outside of a continuation, then the operation will block (and possibly deadlock).
- If ".Result" is accessed on a faulted task, it will raise an AggregateException.

Task.Result 2

.GetAwaiter().GetResult()

- Was designed for internal use.
- It is sometimes used because it returns an Exception (not an AggregateException).
- Blocking effects are the same as with .Result.

Task.Result 3

Advice

Avoid using .Result or
.GetAwaiter().GetResult()
to break asynchrony.

Where Continuations Run

Default Task Behavior

- By default, a Task continuation does *not* run on the current context (thread).
- This means if you need to access resources from the current context (thread), you cannot do it by default.

Note: "Context" and "thread" are not technically equivalent. There are some async operations that do not use thread resources. But for most situations, we can think of these as interchangeable.

Default Task Behavior

Runs on Main Thread | **Runs somewhere else**

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
peopleTask.ContinueWith(
    task =>
    {
        List<Person> people = task.Result;
        foreach (var person in people)
            PersonListBox.Items.Add(person);
    }
);
```

Runs somewhere else

Task Scheduler

- TaskScheduler.FromCurrentSynchronizationContext will return to the prior context.
- For web applications, this means going back to the request thread.
- This may be needed for WebForms or applications that require Session or similar information.

Task Continuation in Main Context

Runs on Main Thread — **Runs somewhere else**

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
peopleTask.ContinueWith(
    task =>
```

Runs on Main Thread

```
    {
        List<Person> people = task.Result;
        foreach (var person in people)
            PersonListBox.Items.Add(person);
    },
    TaskScheduler.FromCurrentSynchronizationContext());
```

Default await Behavior

- By default, code after "await" *does* run on the current context (thread).

- This means that you can safely access resources from that context (thread) – such as UI elements (desktop/mobile) or Session information (web).

Default await Behavior

Runs on Main Thread

Runs somewhere else

```
ClearListBox();
List<Person> people = await reader.GetPeopleAsync();
foreach (var person in people)
    PersonListBox.Items.Add(person);
```

Runs on Main Thread

ConfigureAwait 1

By default, code running after "await" returns to the current context.

This is fine for many situations (and won't break anything), but using ConfigureAwait(false) can optimize performance.

ConfigureAwait 2

ConfigureAwait determines whether processing needs to go back to the current context (thread) after "await"ing an operation.

ConfigureAwait 3

- ConfigureAwait(true) returns to the current context.
  - This is the default
- ConfigureAwait(false) uses whatever context is readily available.

ConfigureAwait(false) is preferred for optimization purposes.

await with ConfigureAwait(false)

Runs on Main Thread

```
ClearListBox();
```

Runs somewhere else

```
List<Person> people = await reader.GetPeopleAsync()
                                  .ConfigureAwait(false);
```

Runs somewhere else

```
foreach (var person in people)
    PersonListBox.Items.Add(person);
```

Runs somewhere else

Importance of Asynchronous Code

Reminder:

Web servers have a limited number of threads to handle incoming requests.

Getting off of these threads (with async code) frees them up to take additional requests.

ConfigureAwait 4

General Guideline:

- ConfigureAwait(false) for library code
- ConfigureAwait(true) for UI code

Exception:

- .NET Core ASP.NET applications do *not* have a current context, so this setting will be ignored.
- .NET Framework ASP.NET applications *do* have a context. You may need to go back to the prior context if you need Session or similar information.

Useful Properties and Methods

.ContinueWith() Parameters 1

- Action<Task>
  - A delegate to run when the task is complete.

- TaskScheduler
  - TaskSchedule.FromCurrentSynchronizationContext will return to the prior thread (e.g. to run the continuation on the UI thread).

.ContinueWith() Parameters 2

- CancellationToken
  - A canceled token prevents the continuation running.
  - CancellationToken.None can be used as a placeholder.

- TaskContinuationOptions
  - OnlyOn… and NotOn… values set conditions on whether the continuation will run.

Task Properties (.NET Core / .NET 5)

- Task Properties
  - IsFaulted
  - IsCanceled
  - IsCompleted*
  - IsCompletedSuccessfully

**Note: Means "no longer running" not "completed successfully"*

IsCompletedSuccessfully

- .NET Core (all versions)
- .NET 5
- .NET Standard 2.1

- NOT .NET Standard 2.0
- NOT .NET Framework

Task Properties (.NET Framework)

- Task Properties
  - IsFaulted
  - IsCanceled
  - IsCompleted*
  - **Status**

**Note: Means "no longer running" not "completed successfully"*

- TaskStatus
  - ***Canceled***
  - Created
  - ***Faulted***
  - ***RanToCompletion***
  - Running
  - WaitingForActivation
  - WaitingForChildrenToComplete
  - WaitingToRun

async void

- async void
- Only for true "fire and forget"
- Disadvantages
  - Cannot tell when (or if) the operation completes
  - Cannot tell whether the operation was successful
  - Cannot see exceptions that occur

- Reminder: Exceptions stay on their own thread unless we go looking for them. Using "await" with a Task is one way to show them.

Exception Handling

- AggregateException
  - Tree structure of exceptions
- Flatten()
  - Flattens the tree structure to a single level of InnerExceptions

Cancellation 1

- CancellationToken is ReadOnly
  - new CancellationToken(true)
  - new CancellationToken(false)
- CancellationTokenSource
  - IDisposable → "using" or call "Dispose"
  - cts.Token → CancellationToken
  - cts.Cancel() → Sets "IsCancellationRequested" to true

Cancellation 2

- ThrowIfCancellationRequested
  - Sets Task Status property
  - Sets IsCompleted, IsCanceled, etc. properties
  - Throws OperationCanceledException (needed for "await")

Cancellation and Continuations

- ContinueWith CancellationToken parameter
  - When "IsCancellationRequested" is true, the continuation **will not run**.
  - An option is to use "CancellationToken.None" as a dummy token.

Writing Asynchronous Methods

Writing Asynchronous Methods 1

- Directly return a Task
- Ex:

```
public Task<Person> GetPersonAsync(int id)
{
    Task<Person> personTask = Task.Run(() => GetPerson(id));
    return personTask;
}
```

Writing Asynchronous Methods 2

- If you "await" something in your method, then the return value is automatically wrapped in a Task.
- Ex:

```
public async Task<Person> GetPersonAsync(int id)
{
    Person person = await Task.Run(() => GetPerson(id));
    return person;
}
```

ConfigureAwait

General Guideline:

- ConfigureAwait(false) for library code
- ConfigureAwait(true) for UI code

Exception:

- .NET Core ASP.NET applications do *not* have a current context, so this setting will be ignored.
- .NET Framework ASP.NET applications *do* have a context. You may need to go back to the prior context if you need Session or similar information.

Parallel Programming

Parallel Programming 1

- Multiple "await"s run in sequence (one at a time)
- Ex: multiple service calls

```
await CallService1Async()
await CallService2Async()
await CallService3Async()
```

CallService2Async will not run until after CallService1Async is complete.
CallService3Async will not run until after CallService2Async is complete.

Parallel Programming 2

- Multiple Tasks can run in parallel (at the same time)
- Ex: multiple service calls

```
Task.Run( () => CallService1 ).ContinueWith(…)
Task.Run( () => CallService2 ).ContinueWith(…)
Task.Run( () => CallService3 ).ContinueWith(…)
```

CallService1, CallService2, and CallService3 all run at the same time.

Parallel Programming 3

- await Task.WhenAll() can be used to determine when all tasks are complete
- Ex:

```
var taskList = new List<Task>();
taskList.Add(task1);
taskList.Add(task2);
taskList.Add(task3);
await Task.WhenAll(taskList);
```

Wrap Up

Task Asynchronous Pattern (TAP)

- Task-Based
- Method Returns a Task
  - Task<T> GetDataAsync()
- Task
  - Represents a concurrent operation
  - May or may not operate on a separate thread
  - Can be chained and combined

async & await

- Syntactic Wrapper Around Task
  - “await” pauses the current method until Task is complete
  - Looks like a blocking operation
  - Does not block current thread
- “async” is just a Hint
  - Does not make a method run asynchronously
  - Tells the compiler to treat “await” as noted above

Thank You!

Jeremy Clark

- http://www.jeremybytes.com
- jeremy@jeremybytes.com
- @jeremybytes

https://github.com/jeremybytes/async-workshop-2021