Asynchronous Programming in C#

Jeremy Clark
www.jeremybytes.com
@jeremybytes

Materials

https://github.com/jeremybytes/async-workshop-nov2020

Schedule

- Hours 1:00 p.m. – 4:00 p.m.
- Break 2:00 p.m. – 2:10 p.m.
- Break 3:00 p.m. – 3:10 p.m.

All Times are Eastern Standard Time

Agenda Part 1

- Getting Results from Async Methods
  - Awaiting Tasks
  - Task Continuations
  - Proper use of Task.Result
  - Avoiding Task.GetAwaiter().GetResult()
  - Avoiding Task.Wait()

Agenda Part 2

- Where Continuations Run
  - Default behavior for post-await code and task continuations
  - The important of Task.ConfigureAwait()
  - Differences for web applications between .NET Core and .NET Framework

Agenda Part 3

- Unit Testing
  - Testing async methods with MSTest
  - Easily creating asynchronous fake objects
  - Testing for exceptions

Agenda Part 4

- Status and Exceptions
  - The dangers of unobserved exceptions
  - Using Task.IsCompleted, Task.IsFaulted
  - Catching full exceptions with Task (AggregateException)
  - Catching partial exceptions with await
  - Dangers of async void methods
  - Returning null vs. Task.CompletedTask or Task.FromResult()

Additional Topics (if time permits)

- Letting asynchrony propagate
- Parallel programming and exceptions

Part 1

Getting Results from Asynchronous Methods

Task Asynchronous Pattern

- 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

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

- Task.Result is a blocking operation. If the Task has not completed, the current thread will be blocked until the Task completes.

- This breaks the asynchronous nature of the code.

Importance of Asynchronous Code

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.

Task.Result 2

.Result

- ".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 3

.GetAwaiter().GetResult()

- For internal use only (meaning, for the C# language / compiler teams)
- It is sometimes used because it returns an Exception (rather than an AggregateException).
- Blocking effects are the same as with .Result.

Task.Result 4

Advice

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

Task.Wait

Task.Wait()

- ".Wait()" blocks the current thread until the task is complete.
- Like using Task.Result, this breaks asynchrony and removes the benefit of async code.


Part 2
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

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

Runs somewhere else

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 desirable because the current context does not need to be captured. This is preferred for optimization purposes.

await with ConfigureAwait(false)

Runs on Main Thread

```
ClearListBox();
List<Person> people =
```

Runs somewhere else

```
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.

Part 3

Unit Testing

Unit Testing Async Methods

The good news:

Most unit testing frameworks (including MSTest and xUnit) support async unit tests.

Unit Testing Async Methods

1. Make the test method "public async Task" instead of "public void".

2. "await" the called method inside the test.

3. Check results with assertions as usual.

Async Unit Test Example

```
[TestMethod]
public async Task GetPeople_WithGoodRecords_Returns...()
{
    var reader = new CSVReader(unusedPath);
    reader.FileLoader = new FakeFileLoader(FakeDataType.Good);
    var result = await reader.GetPeopleAsync();
    Assert.AreEqual(2, result.Count());
}
```

Fake Object with Async Methods

1. Create a class that implements the interface.

2. On a method that returns "Task", use `return Task.CompletedTask.`

3. On a method that returns "Task<T>", use `return Task.FromResult<T>(testDataResult).`

Fake Object with Async Methods

```
public Task<IReadOnlyCollection<Person>> GetPeopleAsync()
{
    var people = new List<Person>()
    {
        new Person() {Id = 1, ...},
        new Person() {Id = 2, ...},
    };
    return Task.FromResult<IReadOnlyCollection<Person>>(people);
}
```

Testing for Exceptions

1. "await" the async method.

2. Use a try/catch block
   - Inside the catch block is the "pass" state
   - If the code gets past the point that should throw an exception, this is a "fail" state.

3. Or use test framework-specific exception checking

Testing for Exceptions

```
[TestMethod]
public async Task WithTask_OnFailure_ReturnsErrorView()
{
    var controller = new PeopleController(GetFaultedReader());
    try
    {
        var view = await controller.WithTask();
        Assert.Fail("Expected Exception not thrown");
    }
    catch (NotImplementedException) { // This is the passing state }
}
```

This line should throw an exception

Part 4

Status and Exceptions

Task Properties (.NET Core)

- 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

Exception Handling with Task

- Task throws an AggregateException
  - Tree structure of exceptions

- This contains all exceptions thrown in the process. For parallel code and parent/child tasks, this can result in multiple exceptions.

Exception Handling with Task

- AggregateException.Flatten()
  - Flattens the tree structure to a single level of InnerExceptions.
  - The inner exceptions can be iterated and logged.

Exception Handling with Task

```
peopleTask.ContinueWith(task =>
  {
    if (task.IsFaulted)
    {
      foreach (var ex in task.Exception.Flatten().InnerExceptions)
          logger.LogException(ex, $"ERROR\n{ex.GetType()}\n{ex.Message}");
    }
    if (task.IsCompletedSuccessfully) { ... }
  });
```

Exception Handling with await

- If an exception is thrown in an awaited method, the AggregateException is unwrapped.
- "await" throws the first inner exception in the AggregateException.
- This is often sufficient (as many AggregateExceptions contain a single inner exception), but be aware that you may be losing information (particularly with parallel code).

Exception Handling with await

```
try
{
    ClearListBox();
    List<Person> people = await reader.GetPeopleAsync();
    foreach (var person in people)
        PersonListBox.Items.Add(person);
}
catch (Exception ex)
{
    logger.LogException(ex, $"ERROR\n{ex.GetType()}\n{ex.Message}");
}
```

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.

Avoid Returning "null" Tasks 1

- Returning a "null" instead of a Task is similar to "async void". There is no way to check for completion, exceptions, etc.

Avoid Returning "null" Tasks 2

- Task.CompletedTask
  Can be used to return a Task that is in the "completed" state.

- Task.FromResult<T>(result)
  Can be use to return a Task with a specific payload.

- Both of these can be used without creating a Task using TaskFactory or Task.Run.

Extras

Additional Topics

Letting Asynchrony Propagate

- It's tempting to use "Task.Result" to break the asynchronous chain.
- Instead, let the asynchronous method flow up through your code.

Advice

Async is like plumbing.
You don't want to pipes to stop partway in the house. You want them to go all the way through.

-Kathleen Dollard (Microsoft)

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);
```

Catching Partial AggregateExceptions

```
var taskList = new List<Task>();
try
{
    foreach (int id in ids) { [run tasks and add to taskList] }
    await Task.WhenAll(taskList);
}
catch (Exception ex) {
    // "ex" is first exception in the AggregateException
}
```

Catching the Entire Aggregate Exception

```
await Task.WhenAll(taskList).ContinueWith(
    task =>
     {
         // "task.Exception" is full AggregateException
         // that can be flattened / iterated for logging
         logger.LogException(task.Exception);
     }, TaskContinuationOptions.OnlyOnFaulted);
```

Thank You!

Jeremy Clark

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

https://github.com/jeremybytes/async-workshop-nov2020