Using Async - Await in C# as Designed

Keith Voels

Please take the two paper handouts up front while they last or online at https://bit.ly/2JjQnHR or https://github.com/keithdv/AsyncAsDesigned/blob/master/Handout.pdf
Keith Voels

- Application Architect – Ecolab
  - .NET Developer – 15 years
- Email: keithvoels@gmail.com
- GitHub
  - https://github.com/keithdv/AsyncAsDesigned
  - All code presented today is available
- LinkedIn
  - www.linkedin.com/in/keith-voels-b6569586
Async - Await as designed - Goals

• Significant performance benefit
• Part of a bigger picture – Task-Based Asynchronization Programming
• What are ExecutionContext and SynchronizationContext and why should I care?
• Code Examples
Agenda

• Async - Await Server Performance
• System.Threading.Tasks - Task-Based Asynchronous Programming
• Understanding the role of ExecutionContext, SynchronizationContext and ConfigureAwait
• Code Examples
Async - Await – Why use it?

• So why learn it and use it?
  • Provides significant performance gains running server-side code by reducing thread contention.

• Demo: Async - Await Application Server Throughput
Async - Await – Why use it?
Agenda

• Async - Await Server Performance
• System.Threading.Tasks - Task-Based Asynchronous Programming
• Understanding the role of ExecutionContext, SynchronizationContext and ConfigureAwait
• Code Examples
Understanding Async - Await means understanding Task-Based Asynchronous Programming.

- Tasks is a `namespace`
  - Provides the behaviors for TAP design pattern
- Task is a class
  - Instantiated and garbage collected like any other object
- Async - Await are `keywords`
  - Makes TAP code shorter, easier to read.
    - Asynchronous code reads like synchronous code
- Analogous to `?` and Nullable
  - `?` is the keyword and Nullable is the namespace and behavior.
System.Threading.Tasks - Task-Based Asynchronous Programming (TAP)

• A task represents the initiation and completion of an asynchronous operation
• Delegate + Execution = Task

Delegates
• Delegate
• ContinueWith

Execution
• Status
• Exception
• Task<T>.Result
• Execution Context (Hidden)
System.Threading.Tasks – TAP as Designed

• The design pattern is Task-Based Asynchronous Programming (TAP)
• It is natural, even expected, for TAP to spread throughout your code
• An instance of Task is like an instance of any other class
  • Always handle the returned Task from an awaitable method or it may not be executed and exceptions will be lost
• Task.Run() and await keyword don’t always cause additional threads to be created. They are scheduled on the Task Scheduler.
• In fact, TAP reduces the number of threads created by allowing the thread pool to decide when to continue and on which thread.
Agenda

• Async - Await Server Performance
• System.Threading.Tasks - Task-Based Asynchronous Programming
• Understanding the role of ExecutionContext, SynchronizationContext and ConfigureAwait
Understanding ExecutionContext, SynchronizationContext and ConfigureAwait – TAP Challenges

• A Task that can be executed by any thread brings challenges.

• **Thread Local Storage**
  • No longer a valid location to source flow information like identity and culture.
  • Solution: **ExecutionContext**
    • Store flow critical information in a container and link it to the Task.

• **UI Thread**
  • UI Controls are not thread safe and can only be interacted with while on the UI thread.
  • Solution: **SynchronizationContext**
    • Execute on the Task’s continuation on the captured environment (i.e. thread).
    • Abstraction so that platforms that don’t have a UI thread can provide their own behavior.

• Put Simply: Move from Thread Local Storage to the Call Stack
Understanding ExecutionContext, SynchronizationContext and ConfigureAwait – Top Level Description

• Execution Context: Critical Objects like Culture, Identity, Permissions
  • Critical - Leave it alone!

• Synchronization Context: UI apps must continue on the same thread
  • Optional – UI Applications need it. Others do not.
  • UI: .ConfigureAwait(true) => Continue on the UI thread. (Default)
  • Non – UI : .ConfigureAwait(false) => Any Thread
    • Note: Also SynchronizationContext.Current == null
Understanding ExecutionContext, SynchronizationContext and ConfigureAwait – Code

WPF / Forms

// Use ONLY for UI events with signatures that do not allow 'async Task'
public async void AsyncAwaitExercise1_Click(object sender, RoutedEventArgs e)
{
    await AsyncMethod().ConfigureAwait(false);
}

// Not used outside of WPF / Forms
public async Task ITouchUIComponents_AsyncMethod()
{
    await AsyncMethod().ConfigureAwait(false);
}

ASP.NET

// Do not use async void outside of WPF / Forms
public async void AsyncAwaitExercise1_Click(object sender, RoutedEventArgs e)
{
    public async Task AsyncMethod()
    {
        await AsyncMethod().ConfigureAwait(false);
    } // Performance improvement, optional for .NET Core
}

Library

// Do not use async void outside of WPF / Forms
public async void AsyncAwaitExercise1_Click(object sender, RoutedEventArgs e)
{
    public async Task AsyncMethod()
    {
        await AsyncMethod().ConfigureAwait(false); // Required to make use in WPF/Forms guaranteed to not block
    }
}
Understanding ExecutionContext, SynchronizationContext and ConfigureAwait

<table>
<thead>
<tr>
<th><strong>Execution Context</strong></th>
<th><strong>Synchronization Context</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Required; Down Call Stack</td>
<td>• Optional; Up Call Stack</td>
</tr>
<tr>
<td>• Provides a <em>single container</em> for all information relevant to the logical thread of execution.</td>
<td>• Captured Location; Environment</td>
</tr>
<tr>
<td>• Framework captures the EC at each asynchronous fork.</td>
<td>• Abstraction to queue work on a particular location (i.e. UI Thread)</td>
</tr>
<tr>
<td>• <strong>Cannot be suppressed.</strong></td>
<td>• Framework calls SC.Post at each continuation.</td>
</tr>
<tr>
<td>• Accessed using Thread.ExecutionContext and Task.ExecutionContext</td>
<td>• <strong>May be suppressed</strong> with .ConfigureAwait(false) or SC.Current == null</td>
</tr>
</tbody>
</table>
Understanding ExecutionContext, SynchronizationContext and ConfigureAwait – Code Demo Setup

![Diagram showing the logical execution flow of async operations with different context scenarios.]

### Code Demo Setup

**AsyncAwait_A()**
```
Thread 1
```

**AsyncAwait_B()**
```
Task A.EC.Run()
Task B.EC = EC.Capture()
Task B.EC.Run()
Task B.EC.SC.Post()
```

**AsyncAwait_C()**
```
Task C.EC.Run()
Task C.EC = EC.Capture()
Task C.EC.SC.Post()
```

**Logical Execution Flow**

1. Task A.EC.Run()
2. Task B.EC = EC.Capture()
3. Task A.EC.Run()
4. Task B.EC.Run()
5. Task B.EC.SC.Post()
6. Task C.EC.Run()
7. Task C.EC.SC.Post()
8. Task B.EC.SC.Post()
9. Task A.EC.SC.Post()

**DispatcherSync...Context**

- **Or**
- **ConfigureAwait(false) Or Sync...Context.Current = Null**

### Further Reading
- [ExecutionContext](https://docs.microsoft.com/en-us/dotnet/api/system.threading.executioncontext)
- [SynchronizationContext](https://docs.microsoft.com/en-us/dotnet/api/system.threading.synchronizationcontext)
- [ConfigureAwait(false)](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskconfigureawait)
Conclusion

• Email: KeithVoels@gmail.com
• GitHub: keithdv

• Links
  • [Video] The zen of async: Best practices for best performance – Microsoft Tech Ed
    • https://www.youtube.com/watch?v=vu2kEstdfuc8
    • Highly Recommended – Commentary from the Microsoft Team on Async - Await design
  • Stephen Toub and Stephen Cleary
    • Async and Await - Stephen Cleary
      • https://blog.stephencleary.com/2012/02/async-and-await.html
    • Async/Await - Best Practices in Asynchronous Programming - Stephen Cleary
    • ExecutionContext vs SynchronizationContext - Stephen Toub
      • https://blogs.msdn.microsoft.com/pfxteam/2012/06/15/executioncontext-vs-synchronizationcontext/