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

System.Threading.Tasks – Tasks vs Async - Await

- **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 <u>it may not be executed</u> and <u>exceptions will be lost</u>
- Task.Run() and await keyword ***don't always*** cause additional threads to be created. They are *scheduled* on the Task Scheduler.
- In fact, TAP <u>***reduces*** the number of threads</u> created by allowing the thread pool to decide when to continue and on which thread.

- https://msdn.microsoft.com/en-us/library/mt674882.aspx
- https://msdn.microsoft.com/en-us/magazine/jj991977.aspx - Stephen Cleary

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.
    - System.Threading.Thread.CurrentThread.CurrentCulture =>
    - System.Threading.Thread.CurrentThread.**ExecutionContext**.CurrentCulture
- **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).
    - <u>Abstraction</u> 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)~~;

}

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

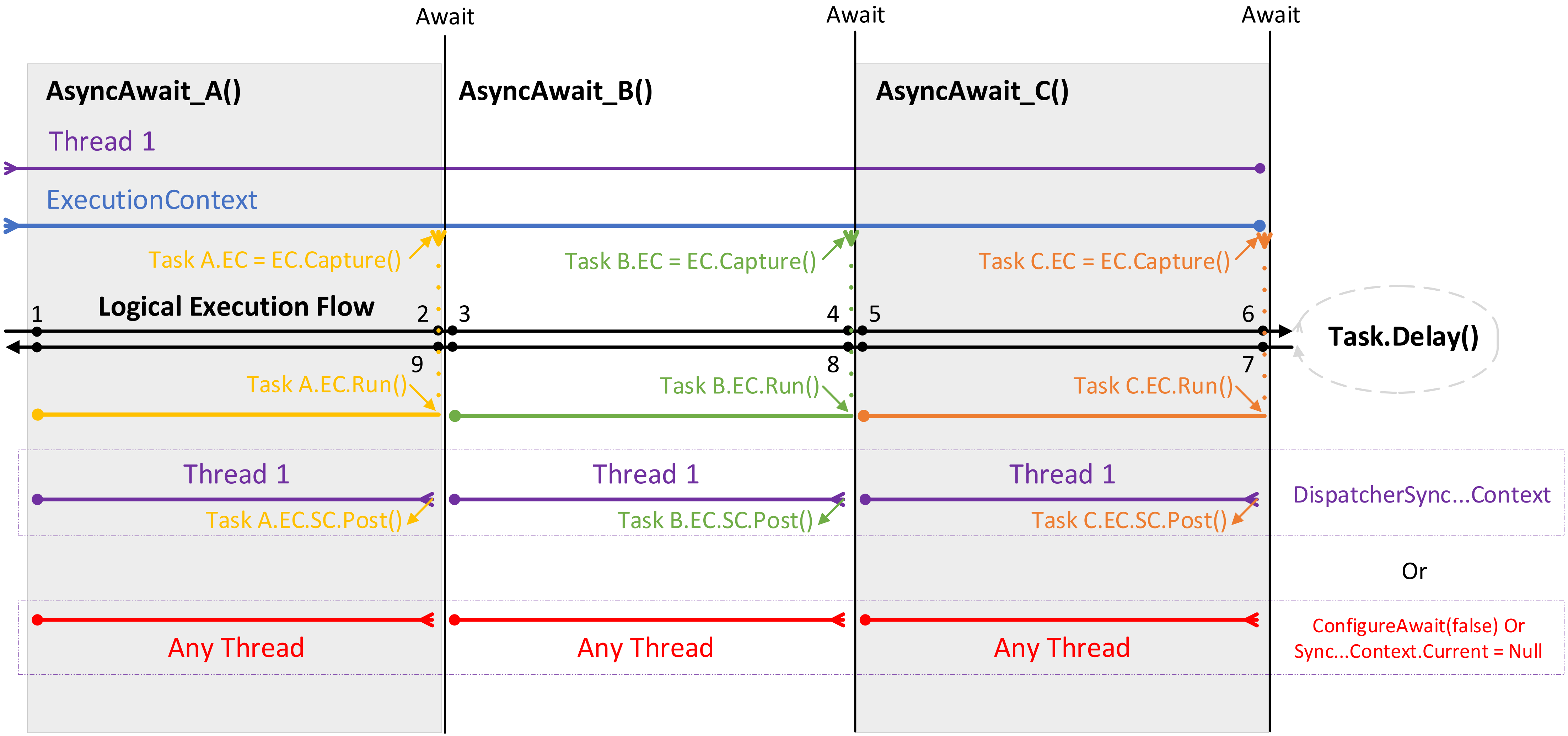
Execution Context

- Required; Down Call Stack
- Provides a **single container** for all information relevant to the logical thread of execution.
- Framework captures the EC at each <u>asynchronous fork</u>.
- **Cannot be suppressed**.*
- Accessed using Thread.ExecutionContext and Task.ExecutionContext

Synchronization Context

- Optional; Up Call Stack
- Captured Location; Environment
- Abstraction to queue work on a particular location (i.e. UI Thread)
- Framework calls SC.Post at <u>each continuation</u>.
- **May be suppressed** with .ConfigureAwait(false) or SC.Current == null

Understanding ExecutionContext, SynchronizationContext and ConfigureAwait – Code Demo Setup



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=vu2kEstfuc8
    - **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
      - https://msdn.microsoft.com/en-us/magazine/jj991977.aspx
    - ExecutionContext vs SynchronizationContext - Stephen Toub
      - https://blogs.msdn.microsoft.com/pfxteam/2012/06/15/executioncontext-vs-synchronizationcontext/