Additional Return Types and Avoiding Common Pitfalls

Kevin Dockx
Architect

@KevinDockx | www.kevindockx.com

Coming Up

Additional async return types

Common pitfalls to avoid when writing asynchronous code

Additional Async Return Types

Any type that has an accessible `GetAwaiter()` method

- Returned object must implement `System.Runtime.CompilerServices.ICriticalNotifyCompletion`

Additional Async Return Types

`Task` **and** `Task<T>` **are reference types**

- Unwanted memory allocation in performance-critical paths

Thanks to `GetAwaiter()`**, a value type can be created instead**

- `ValueTask<T>`

`ValueTask<T>`

A struct that wraps a `Task<T>` and a result of type `T`

Additional Async Return Types

If `ValueTask<T>` is completed

- … the underlying result would be used

Otherwise

- … the task is allocated

Additional Async Return Types

Good? Bad?

- When the async method runs asynchronously, it's **bad** for performance
- When the async method runs synchronously it's **good** for performance

Additional Async Return Types

Other types intended for Windows workloads

- `DispatcherOperation`
- `IAsyncAction`
- `IAsyncActionWithProgress<TProgress>`
- `IAsyncOperation<TResult>`
- `IAsyncOperationWithProgress<TResult, TProgress>`

Offloading Legacy Code to a Background Thread

Legacy code, like a long-running algorithm, is computational-bound code

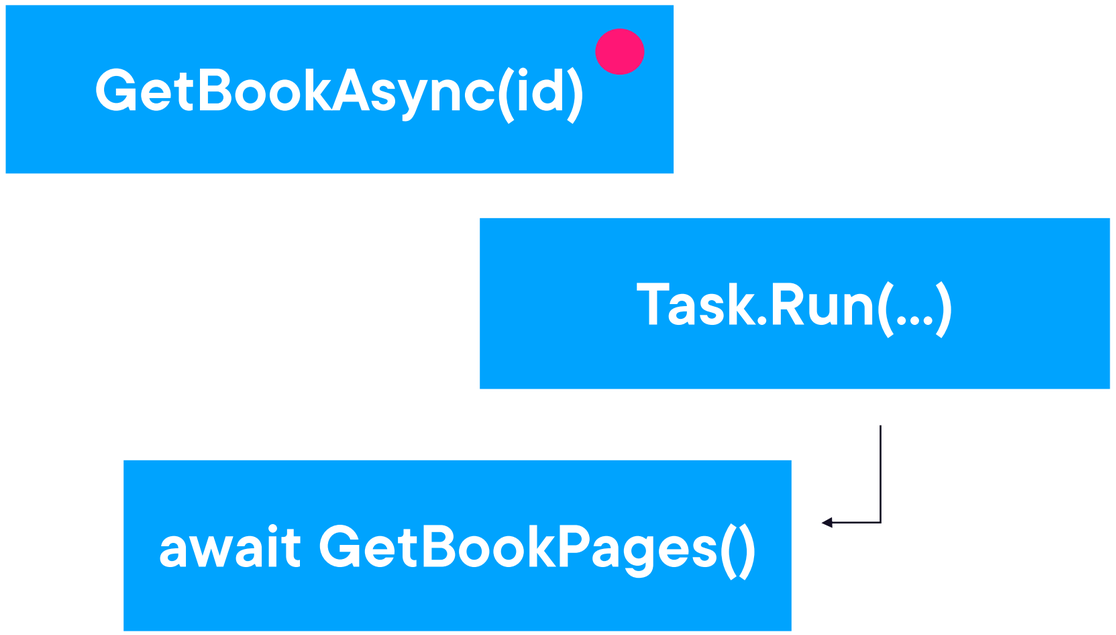
This can be offloaded to a background thread using `async await`

- Can run concurrently

Demo

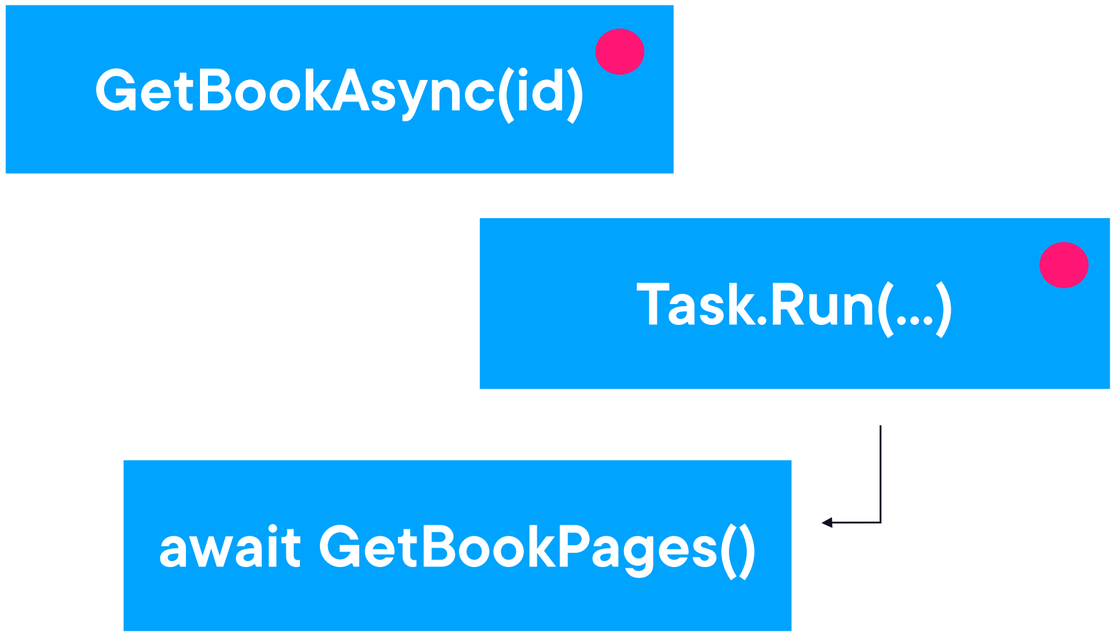
Wrapping synchronous code with
`Task.Run()`

Pitfall #1: Using Task.Run() on the Server



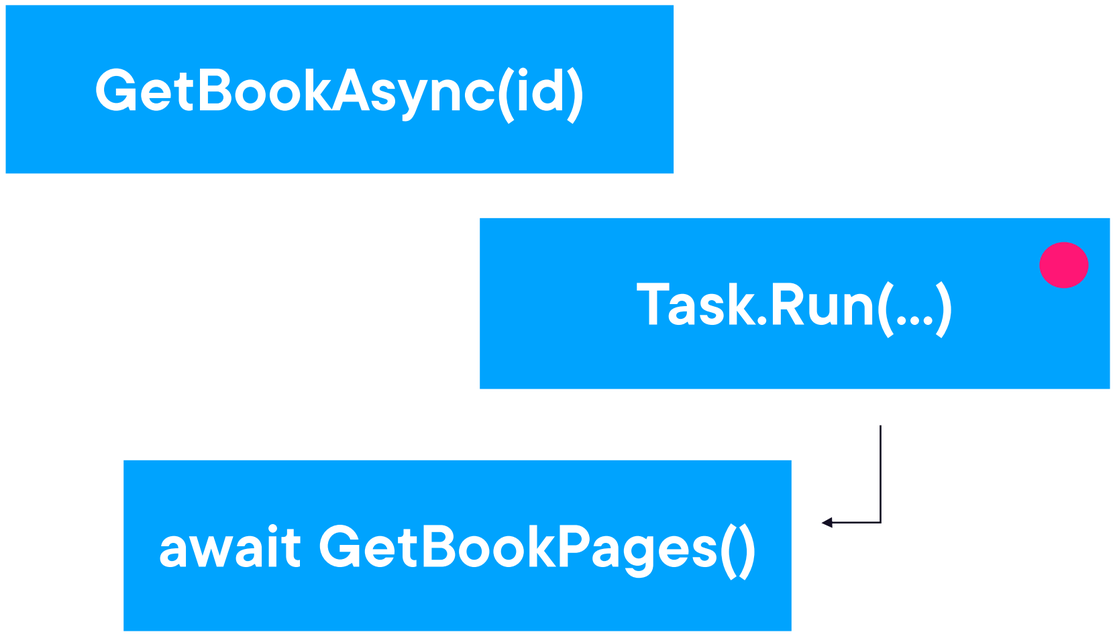
Pseudocode sample using `Task.Run()`

Pitfall #1: Using Task.Run() on the Server



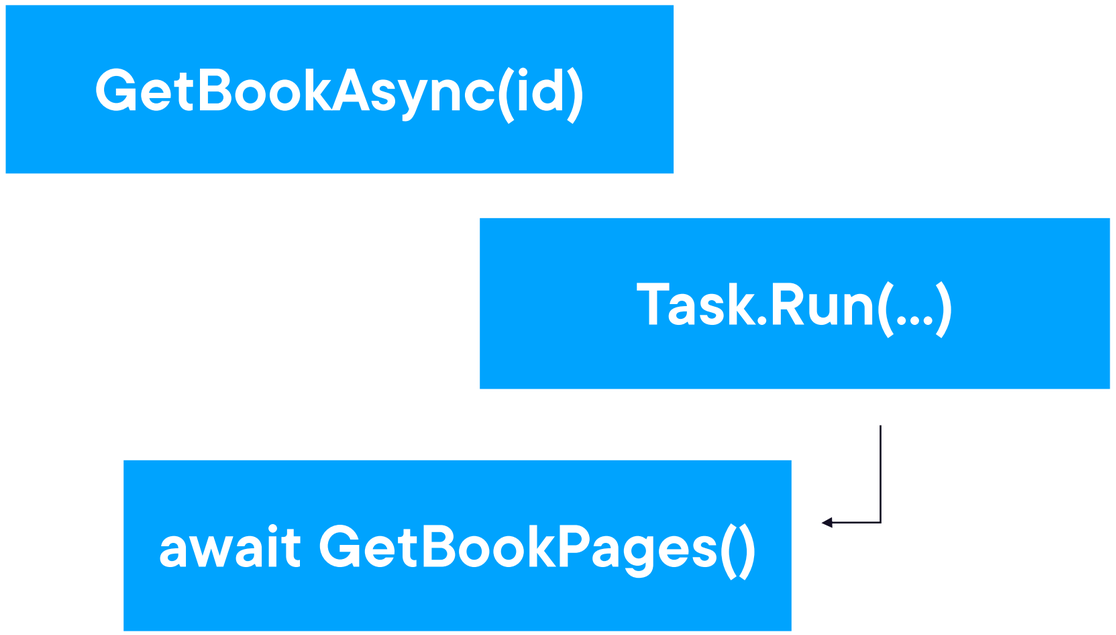
Pseudocode sample using `Task.Run()`

Pitfall #1: Using Task.Run() on the Server



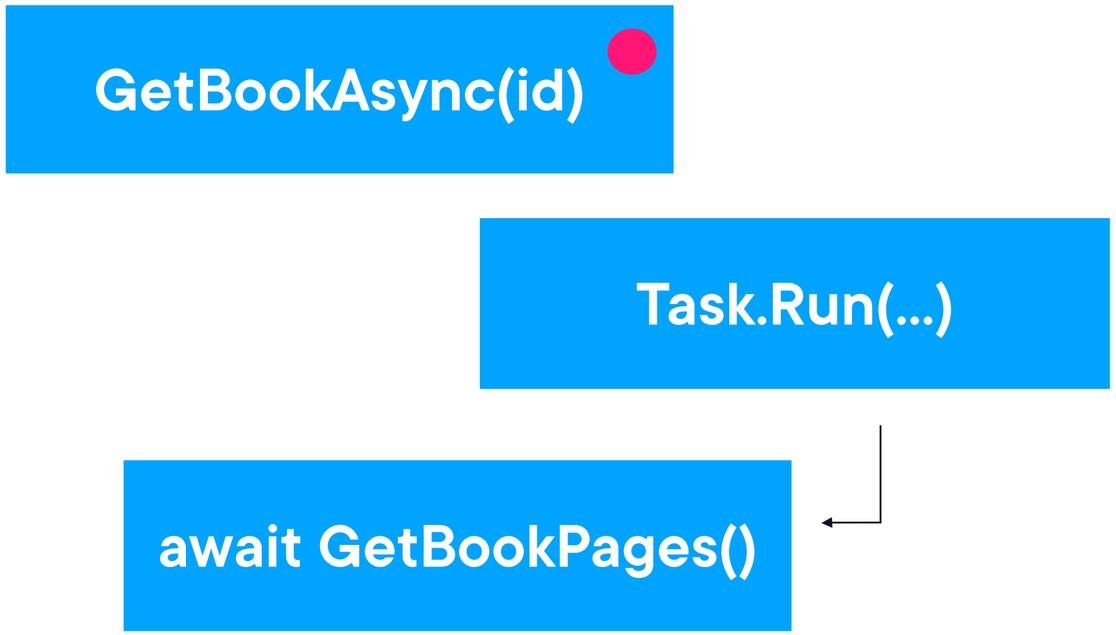
Pseudocode sample using `Task.Run()`

Pitfall #1: Using Task.Run() on the Server



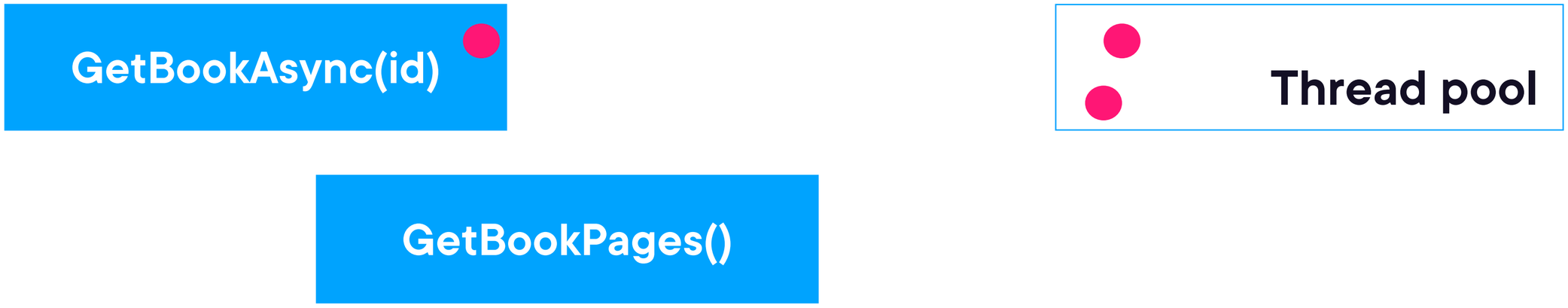
Pseudocode sample using `Task.Run()`

Pitfall #1: Using Task.Run() on the Server



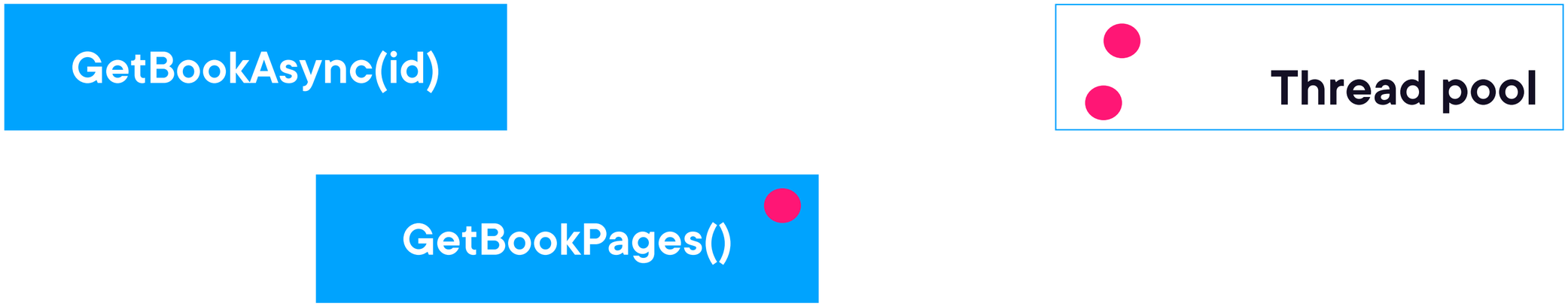
Pseudocode sample using `Task.Run()`

Pitfall #1: Using Task.Run() on the Server



Pseudocode sample without `Task.Run()`

Pitfall #1: Using Task.Run() on the Server



Pseudocode sample without `Task.Run()`

Pitfall #1: Using Task.Run() on the Server

ASP.NET Core is not optimized for `Task.Run()`

- Creates an unoptimized thread
- Causes overhead

Pitfall #1: Using Task.Run() on the Server

`Task.Run()` on the server decreases scalability

It's intended for use on the client
(e.g.: to keep the UI responsive)

Demo

Blocking async code

Pitfall #2: Blocking Async Code

`Task.Wait()` **and** `Task.Result()` **block the calling thread**

- Thread isn't returned to the thread pool

Blocking async code hurts scalability

Pitfall #2: Blocking Async Code

ASP.NET Core doesn't have a synchronization context (the old ASP.NET does)

- Improves performance
- Makes it easier to write async code

```
await MyMethodAsync().ConfigureAwait(false);

await httpClient.GetAsync("UriToResource").ConfigureAwait(false);
```

ConfigureAwait(false)

Avoids deadlocks in the old, full .NET framework
Not necessary anymore in ASP.NET Core

Demo

Modifying shared state

Pitfall #3: Modifying Shared State

Different threads might manipulate the same state at the same time

- Correctness cannot be guaranteed

Summary

Notable additional return types

- `GetAwaiter()`
- `IAsyncEnumerable<T>`

Summary

Don’t use `Task.Run()` on the server

- Hurts scalability

Don’t block async code

- Hurts scalability

Don’t modify shared state

- State can’t be guaranteed

The End is Nigh...

Questions?
Use the discussions tab on the course page

Please consider rating this course! :)


You're ready
to be
AWESOME!