Starting at the Bottom with Your Data Access Layer

Kevin Dockx
ARCHITECT

@KevinDockx https://www.kevindockx.com

Coming Up

Keywords async & await

The Purpose of Task and Task<T>

DAL and Repository

Naming Guidelines, Conventions and Best Practices

The async / await Keywords

Marking a method with the async modifier

- Ensures that the await keyword can be used inside that method
- Transforms the method into a state machine (generated by the compiler)

The async / await Keywords

Using the await operator

- Tells the compiler that the async method can't continue until the awaited asynchronous process is complete
- Returns control to the caller of the async method (potentially right back up to the thread being freed)

The async / await Keywords

A method that is not marked with the async modifier cannot be awaited,

When an async method doesn't contain an await operator, the method simply executes as a synchronous method does

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

The async / await Keywords

```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}

public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();
     return books;
}

public IEnumerable<Guid> CalculateBookIdsForUser()
{    …
     return bookIdsForUser;
}
```

Async Return Types

void

Task

Task<T>

Types with accessible GetAwaiter methods (new in C#7)

Async Return Types

void

- Only advised for event handlers
- Hard to handle exceptions
- Difficult to test
- No easy way to notify the calling code of their status

Async Return Types

Task and Task<T>

- Represents a single operation that returns nothing (Task) or a value of type T (Task<T>) and usually executes asynchronously.
- Represents the execution of the async method

Async Return Types

Task and Task<T>

- Status, IsCanceled, IsCompleted, and IsFaulted properties allow determining the state of a Task
- Gets status complete when the method completes (and optionally returns the method value as the Task's result)

Through Task and Task<T> we can know the state of an async operation

Tasks are managed by the state machine generated by the compiler when a method is marked with the async modifier

Async Return Types

Types with ...

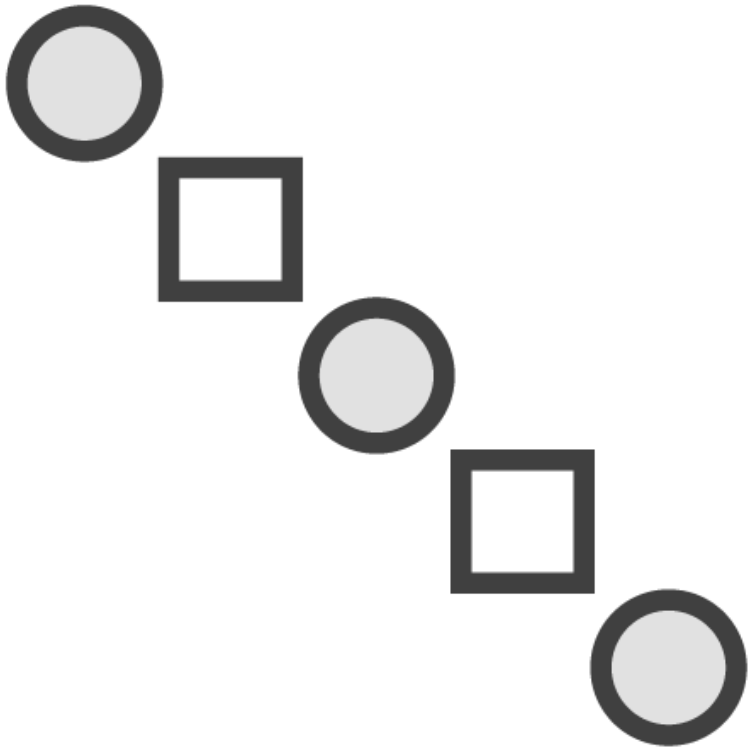
- ... an accessible GetAwaiter method
- ... of which the returned object implements the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface

Async Return Types

Using reference types can induce memory allocation in performance-critical paths, and that can adversely affect performance

Supporting generalized return types allows returning a lightweight value type

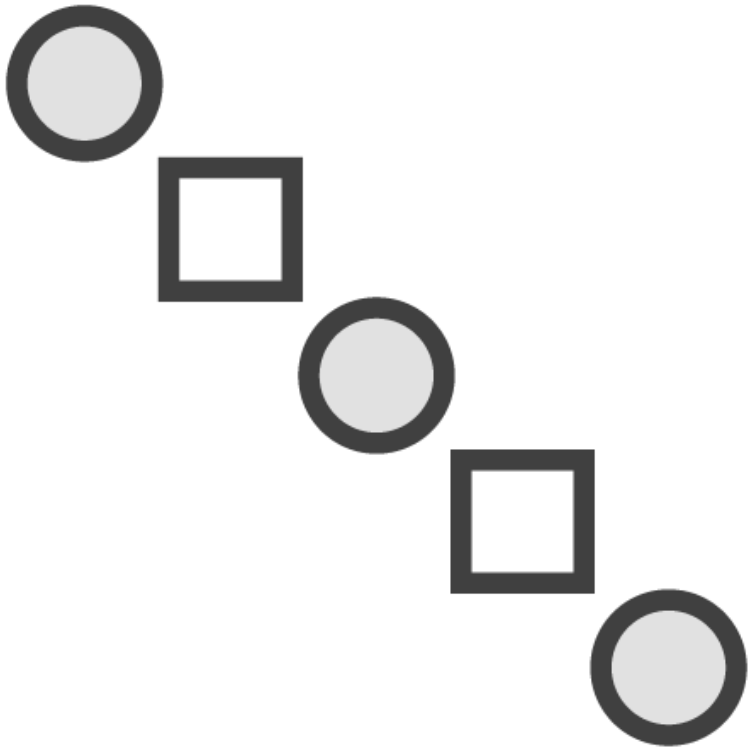
Async Patterns: TAP, EAP, and APM



Task-based Asynchronous Pattern (TAP)

- Best practice today
- Based on Task, Task<T>, GetAwaiter()-implementing types

Async Patterns: TAP, EAP, and APM

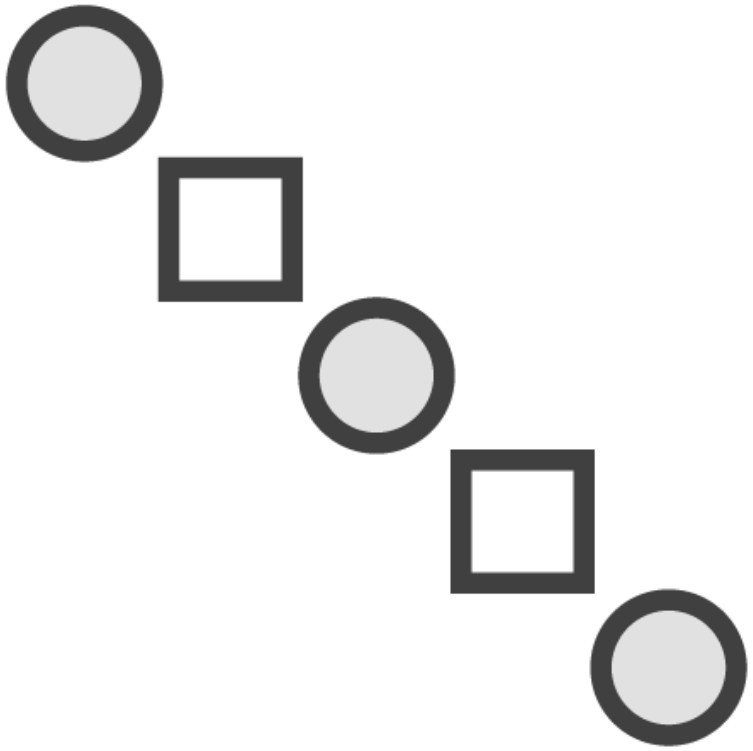


Event-based Asynchronous Pattern (EAP)

- Multithreading without the complexity
- MethodName**Async** (method)
- MethodName**Completed** (event)
- MethodName**AsyncCancel** (method)

Mainly used before .NET 4

Async Patterns: TAP, EAP, and APM

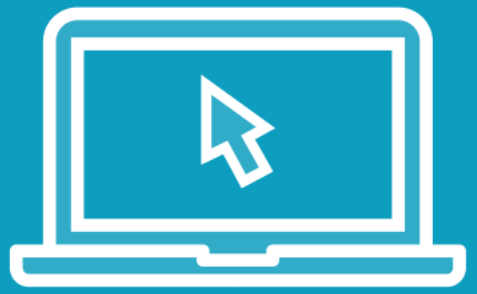


Asynchronous Programming Model (APM)

- Async operations are implemented as two methods named Begin*OperationName* and End*OperationName*

FileStream used to default to this model, which has since been replaced by TAP

Demo



Starting From Scratch with a DAL

The Repository Pattern

Without the repository pattern, we're likely to ...

- ... run into code duplication
- ... create error-prone code
- ... make it harder to test the consuming class

The Repository Pattern

An abstraction that reduces complexity and aims to make the code, safe for the repository implementation, persistence ignorant

The Repository Pattern

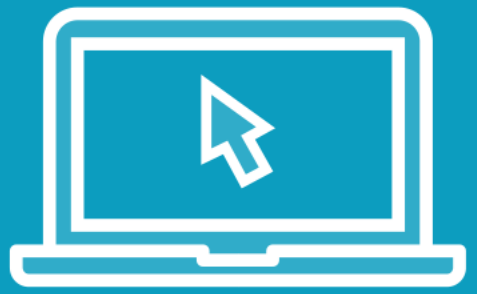
When using the repository pattern, we can achieve ...

- ... less code duplication
- ... less error-prone code
- ... better testability of the consuming class

Persistence Ignorant

Switching out the persistence technology is not the main purpose of the repository pattern, choosing the best one for each repository method is

Demo



Designing a Repository Contract

Contracts and Async Modifiers

An interface is a contract, which makes the GetBooksAsync() definition a *contract detail*

Using the async/await keywords tell us how the method is implemented, which makes it an *implementation detail*

Demo

Implementing the Repository Contract

Summary

Marking a method with the async modifier

- Ensures that the await keyword can be used inside that method
- Transforms the method into a state machine

Summary

Using the await operator

- Tells the compiler that the async method can't continue until the awaited asynchronous process is complete
- Returns control to the caller of the async method

Summary

A Task

- Represents a single operation that returns nothing (Task) or a value of type T (Task<T>)
- Represents the execution of the async method

Summary

Through Task and Task<T> we can know the state of an async operation

Tasks are managed by the state machine generated by the compiler when a method is marked with the async modifier

Summary

Don’t return void from an async operation (unless it’s an event handler)

Tasks are managed by the state machine generated by the compiler when a method is marked with the async modifier

Summary

Generalized return types

- Have an accessible GetAwaiter() method
- Returned object implements System.Runtime.CompilerServices.ICriticalNotifyCompletion