Starting at the Bottom with Your Data Access Layer

Kevin Dockx
Architect

@KevinDockx | www.kevindockx.com

Coming Up

Keywords `async` & `await`

The purpose of `Task` and `Task<T>`

The data access layer and repository pattern

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

The async / await Keywords

A method that is not marked with the `async` modifier should not be awaited

When an asynchronous 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

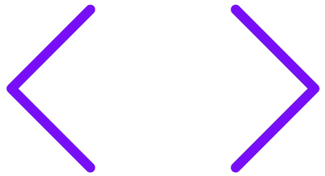
```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}
public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();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

The async / await Keywords

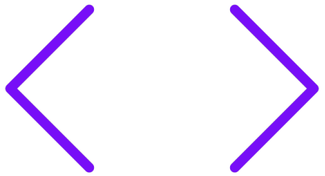
```
public async IActionResult MyActionThatCallsGetBooksAsync()
{    …
     var books = await GetBooksAsync();
     …
}
public async Task<IEnumerable<Book>> GetBooksAsync()
{
     var bookIds = CalculateBookIdsForUser();
     var books = await _context.Books.Where(b =>
                 bookIds.Contains(b.Id)).ToListAsync();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

Async Return Types

`void`



`Task`



`Task<T>`

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 asynchronous 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 asynchronous operation

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

```
public class StateMachineExample : IAsyncStateMachine
{

    public void MoveNext()
    {
        // move to the next state
    }

    public void SetStateMachine(IAsyncStateMachine stateMachine)
    {
        // set the state machine
    }
}
```

State Machine Example

An implementation of `IAsyncStateMachine`

Async Patterns: TAP, EAP, and APM

Task-based Asynchronous Pattern (TAP)

- Best practice today
- Based on `Task` and `Task<T>` (and other async return types)

Async Patterns: TAP, EAP, and APM

Event-based Asynchronous Pattern (EAP)

- Multithreading without the complexity
- `MethodNameAsync` (method)
- `MethodNameCompleted` (event)
- `MethodNameAsyncCancel` (method)

Mainly used before .NET 4 (full framework)

Async Patterns: TAP, EAP, and APM

Asynchronous Programming Model (APM)

- Async operations are implemented as two methods named `BeginOperationName` and `EndOperationName`

`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 the 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 a state machine generated by the compiler when a method is marked with the `async` modifier

Asynchronously Reading Resources