Synchronous vs Asynchronous Programming in ASP.NET Core Web API  
(https://www.c-sharpcorner.com/article/synchronous-vs-asynchronous-programming-in-asp-net-core-web-api/)  
  
Synchronous Programming

In synchronous programming, tasks are executed one after the other, sequentially. When a request is made to a synchronous API, the server processes the request and waits for it to be completed before moving on to the next task. This means that if one operation takes a long time to finish, it can block the execution of subsequent operations, potentially leading to slower response times for clients.

Asynchronous Programming

On the other hand, asynchronous programming allows tasks to be executed concurrently. When a request is made to an asynchronous API, the server can initiate tasks and continue processing other requests without waiting for the previous tasks to be completed. This can lead to better scalability and responsiveness, especially in scenarios where certain operations, such as I/O operations, may take some time.

ASP.NET Core Web API

In ASP.NET Core Web API, you have the flexibility to choose between synchronous and asynchronous programming models. The framework supports both approaches. Asynchronous programming is particularly useful when dealing with I/O-bound operations, such as accessing a database or making external API calls, where the application can continue processing other tasks while waiting for the I/O operation to complete.

To implement asynchronous programming in ASP.NET Core Web API, you can use the `async` and `await` keywords in your controller methods, allowing you to write non-blocking code. This helps improve the overall performance and responsiveness of your API, especially in scenarios with high concurrency.

while synchronous programming is straightforward, asynchronous programming in ASP.NET Core Web API can enhance scalability and responsiveness by allowing the server to handle multiple tasks concurrently. The choice between synchronous and asynchronous programming depends on the specific requirements and characteristics of your application.

Let's Start the complete Implementation.

Create Model

namespace SyncVsAsyncProgrammingIn.NETCoreAPI.Model

{

public class User

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Address { get; set; }

public string Country { get; set; }

public string ZipCode { get; set; }

}

}

Application Db Context

using Microsoft.EntityFrameworkCore;

using SyncVsAsyncProgrammingIn.NETCoreAPI.Model;

using System.Collections.Generic;

namespace SyncVsAsyncProgrammingIn.NETCoreAPI.ApplicationDbContext

{

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options)

: base(options)

{

}

public DbSet<User> Users { get; set; }

}

}

IUserRepository Interface For User Repository

using SyncVsAsyncProgrammingIn.NETCoreAPI.Model;

namespace SyncVsAsyncProgrammingIn.NETCoreAPI.IRepository

{

public interface IUserRepository

{

Task<User> GetUserByIdAsync(int id);

User GetUserByIdSync(int id);

}

}

User Repository

using Microsoft.EntityFrameworkCore;

using SyncVsAsyncProgrammingIn.NETCoreAPI.ApplicationDbContext;

using SyncVsAsyncProgrammingIn.NETCoreAPI.IRepository;

using SyncVsAsyncProgrammingIn.NETCoreAPI.Model;

namespace SyncVsAsyncProgrammingIn.NETCoreAPI.Repository

{

public class UserRepository : IUserRepository

{

private readonly AppDbContext \_dbContext;

private readonly ILogger<UserRepository> \_logger;

public UserRepository(AppDbContext dbContext, ILogger<UserRepository> logger)

{

\_dbContext = dbContext;

\_logger = logger;

}

public User GetUserByIdSync(int id)

{

try

{

// Synchronous database query

return \_dbContext.Users.FirstOrDefault(u => u.Id == id);

}

catch(Exception ex)

{

\_logger.LogError($"Error in UserRepository.GetUserById: {ex.Message}");

throw new ApplicationException("An error occurred while fetching user details. Please try again later.");

}

}

public async Task<User> GetUserByIdAsync(int id)

{

// Asynchronous database query

try

{

return await \_dbContext.Users.FirstOrDefaultAsync(u => u.Id == id);

}

catch(Exception ex)

{

\_logger.LogError($"Error in UserRepository.GetUserById: {ex.Message}");

throw new ApplicationException("An error occurred while fetching user details. Please try again later.");

}

}

}

}

Register the Service in the Program.cs Class

using Microsoft.EntityFrameworkCore;

using Microsoft.OpenApi.Models;

using SyncVsAsyncProgrammingIn.NETCoreAPI.ApplicationDbContext;

using SyncVsAsyncProgrammingIn.NETCoreAPI.IRepository;

using SyncVsAsyncProgrammingIn.NETCoreAPI.Repository;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

var configuration = builder.Configuration;

// Add services to the container.

builder.Services.AddDbContext<AppDbContext>(options =>

{

options.UseSqlServer(configuration.GetConnectionString("DefaultConnection")); // Replace with your database provider and connection string

});

// Add services to the container.

builder.Services.AddTransient<IUserRepository , UserRepository>();

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at <https://aka.ms/aspnetcore/swashbuckle>

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "Sync Vs Async Programming In Asp.Net Core Web API", Version = "v1" });

});

var app = builder.Build();

// Configure the HTTP request pipeline.

if(app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

ASyncUserController

using Microsoft.AspNetCore.Mvc;

using SyncVsAsyncProgrammingIn.NETCoreAPI.IRepository;

namespace SyncVsAsyncProgrammingIn.NETCoreAPI.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ASyncUserController : ControllerBase

{

private readonly IUserRepository \_userRepository;

public ASyncUserController(IUserRepository userRepository)

{

\_userRepository = userRepository;

}

[HttpGet(nameof(GetUserByIdAsync))]

public async Task<IActionResult> GetUserByIdAsync(int id)

{

try

{

// Asynchronous database call

var user = await \_userRepository.GetUserByIdAsync(id);

if(user == null)

{

return NotFound(); // 404 Not Found

}

return Ok(user); // 200 OK with user details

}

catch(Exception ex)

{

// Handle exceptions

return StatusCode(500, $"Internal Server Error: {ex.Message}");

}

}

}

}

SyncUserController

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using SyncVsAsyncProgrammingIn.NETCoreAPI.IRepository;

namespace SyncVsAsyncProgrammingIn.NETCoreAPI.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class SyncUserController : ControllerBase

{

private readonly IUserRepository \_userRepository;

public SyncUserController(IUserRepository userRepository)

{

\_userRepository = userRepository;

}

[HttpGet(nameof(GetUserById))]

public IActionResult GetUserById(int id)

{

try

{

// Synchronous database call

var user = \_userRepository.GetUserByIdSync(id);

if(user == null)

{

return NotFound(); // 404 Not Found

}

return Ok(user); // 200 OK with user details

}

catch(Exception ex)

{

// Handle exceptions

return StatusCode(500, $"Internal Server Error: {ex.Message}");

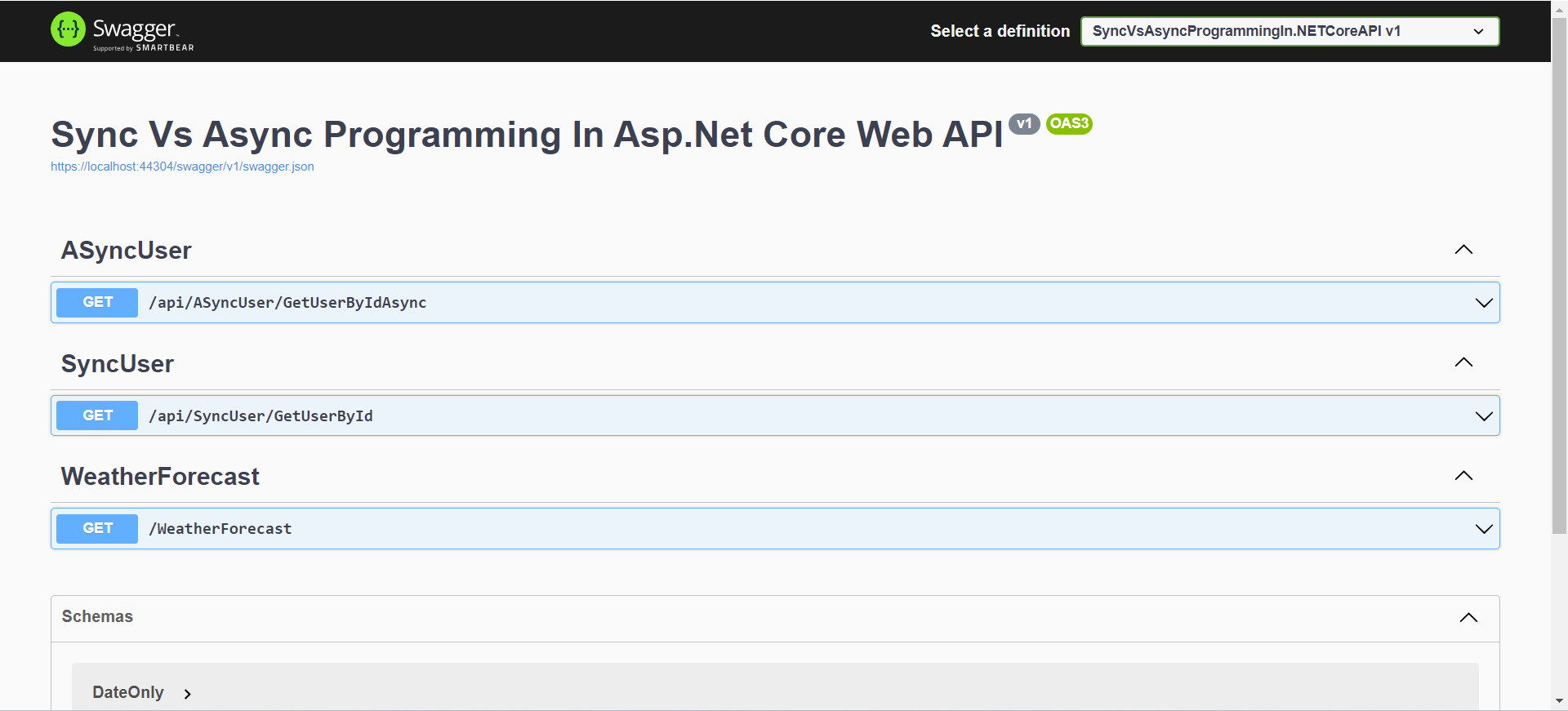
}

}

}

}

**Output**



**GitHub Project URL:**<https://github.com/SardarMudassarAliKhan/SyncVsAsyncProgrammingIn.NETCoreAPI>

Conclusion

Mastering the balance between synchronous and asynchronous programming is crucial for optimizing the performance and scalability of ASP.NET Core Web API applications. While synchronous operations are suitable for simple and quick tasks, leveraging asynchronous programming becomes essential when dealing with time-consuming operations, such as database queries or external API calls. By understanding the strengths and use cases of each approach, developers can make informed decisions to create responsive and efficient web APIs that meet the demands of modern, high-performance applications.

**Async vs Sync in C#: Understanding the Key Differences**

(<https://www.bytehide.com/blog/async-vs-sync-csharp>)

Synchronous Programming

Synchronous programming is a sequential execution model in which a function or method must complete before the next one can start. This often leads to blocking, where the program halts until a long-running operation finishes.

Example

Imagine you are in a line at the grocery store. Each person in the line has to wait for the person in front of them to finish paying before they can start paying for their groceries.

This is like synchronous programming, where each task has to wait for the previous task to finish before it can start. While waiting, nothing else happens, and everything comes to a standstill.

Asynchronous Programming

[Asynchronous programming](https://content.bytehide.com/asynchronous-programming-csharp/), on the other hand, allows multiple tasks to run concurrently without blocking the main thread. This approach significantly improves responsiveness and scalability in modern applications.

Example

Now, imagine you are at a playground with multiple slides. Many kids can play on different slides at the same time without waiting for one another.

This is like asynchronous programming, where multiple tasks can run at the same time without stopping others from running. This way, the playground remains lively and active, just like a responsive and scalable application.

The Importance of Async and Sync in C#

Mastering async and sync patterns is essential for C# developers, as they help build responsive, efficient, and maintainable applications. By understanding when to use each approach, you can optimize your code and avoid common pitfalls.

The Core Concepts of Async and Sync in C#

Before diving into the differences, let’s first understand the building blocks of async and sync methods in C#.

The async Keyword

The [async keyword](https://content.bytehide.com/async-csharp/) marks a method as asynchronous, indicating that it can run without blocking the main thread. This allows the method to perform time-consuming operations while other tasks continue to execute.

// Example of async method

public async Task<string> FetchDataAsync()

{

// Time-consuming operation here

}

The await Keyword

The [await keyword](https://content.bytehide.com/await-csharp/) is used in conjunction with async methods, allowing the execution to yield control to the calling method until the awaited task is complete. This prevents blocking while maintaining a logical flow in your code.

// Example of using await

public async Task<string> ProcessDataAsync()

{

string data = await FetchDataAsync();

// Continue processing data

}

Tasks and Task-Based Asynchronous Pattern (TAP)

Tasks are fundamental building blocks of async programming in C#. The Task class represents an asynchronous operation, which can be awaited using the await keyword. The Task-Based Asynchronous Pattern (TAP) is a standard pattern that promotes writing asynchronous code using Task and Task<TResult> objects.

**Features and Benefits of Async and Sync in C#**

1. Improved Responsiveness

Asynchronous methods improve the responsiveness of applications, particularly in UI-driven scenarios. By not blocking the main thread, async methods allow the application to stay responsive even during time-consuming operations.

2 .Scalability

Async programming is crucial for building scalable applications, as it allows for better resource utilization. By not blocking threads, it frees up resources to handle other tasks in parallel.

1. Code Maintainability

Async methods allow for a more readable and maintainable codebase. By using the await keyword, developers can write asynchronous code that appears similar to synchronous code, making it easier to understand and maintain.

Comparing Async and Sync Methods in C#

Syntax Differences

Async methods use the async and await keywords, while sync methods do not. The return type of an async method is typically Task or Task<TResult>.

// Synchronous method

public int CalculateSum(int a, int b)

{

return a + b;

}

// Asynchronous method

public async Task<int> CalculateSumAsync(int a, int b)

{

return await Task.Run(() => a + b);

}

Execution Flow

In synchronous programming, the execution flow is sequential, and methods must complete before the next one can start. In asynchronous programming, the await keyword allows the execution to yield control, enabling the program to continue without waiting for the async method to complete.

Error Handling

Error handling in async methods is quite similar to synchronous methods; you can use try-catch blocks to catch exceptions. However, exceptions in async methods are encapsulated in the returned task, making it essential to await the task to handle exceptions properly.

// Example of error handling in async methods

public async Task HandleErrorAsync()

{

try

{

await SomeAsyncMethod();

}

catch (Exception ex)

{

// Handle exception here

}

}

Use Cases for Async and Sync in C#

1. When to Use Synchronous Methods

Synchronous methods are suitable for simple, computationally-bound operations that do not involve IO-bound tasks or long-running operations. They are also ideal when a specific execution order must be maintained. For instance:

// Simple synchronous method to calculate the factorial of a number

public int CalculateFactorial(int n)

{

if (n <= 1)

return 1;

else

return n \* CalculateFactorial(n - 1);

}

In this example, the method calculates the factorial of a number synchronously. The calculations are purely computational and do not involve any IO-bound tasks, making synchronous execution appropriate.

1. When to Use Asynchronous Methods

Asynchronous methods should be used for IO-bound tasks, long-running operations, and situations where responsiveness or scalability is crucial. Common examples include file operations, network requests, and UI updates. Let’s take a look at some examples:

// Asynchronous method to download a file from a URL

public async Task DownloadFileAsync(string url, string destinationPath)

{

using (HttpClient client = new HttpClient())

{

using (HttpResponseMessage response = await client.GetAsync(url))

{

using (Stream streamToReadFrom = await response.Content.ReadAsStreamAsync())

{

using (Stream streamToWriteTo = File.Open(destinationPath, FileMode.Create))

{

await streamToReadFrom.CopyToAsync(streamToWriteTo);

}

}

}

}

}

In this example, the method downloads a file from a URL asynchronously. Downloading a file is an IO-bound task that can take a significant amount of time, making asynchronous execution suitable to avoid blocking the main thread.

// Asynchronous method to update UI elements in a responsive manner

private async void Button\_Click(object sender, RoutedEventArgs e)

{

// Disable the button to prevent multiple clicks

Button.IsEnabled = false;

// Perform a time-consuming operation asynchronously

await Task.Run(() => PerformLongRunningOperation());

// Update the UI element with the result

ResultLabel.Content = "Operation Completed";

// Re-enable the button

Button.IsEnabled = true;

}

In this example, we have a button click event in a UI application. When the button is clicked, a long-running operation is performed asynchronously, ensuring that the UI remains responsive. Once the operation is completed, the UI is updated with the result.

This is a common use case for asynchronous methods in UI applications to prevent the UI from freezing during long-running operations.

Potential Drawbacks of Asynchronous Programming

While async programming offers numerous benefits, it also has some pitfalls. Let’s explore these challenges with some real-world examples and tips for mitigating them.

1. Complexity and Learning Curve

Asynchronous programming can be more complex than synchronous programming, with a steeper learning curve for developers new to the concept. For example, consider the following asynchronous method that uses Task.WhenAll to process multiple tasks concurrently:

public async Task ProcessMultipleFilesAsync(IEnumerable<string> filePaths)

{

var fileProcessingTasks = filePaths.Select(async filePath =>

{

var content = await File.ReadAllTextAsync(filePath);

// Process the content here

});

await Task.WhenAll(fileProcessingTasks);

}

This code might be harder to understand for developers new to asynchronous programming, but with practice and experience, they can become proficient in writing and understanding async code.

1. Debugging Challenges

Debugging async code can be more challenging due to the non-linear execution flow, making it harder to trace issues. For example, consider the following async method:

public async Task<string> FetchDataAsync()

{

using (HttpClient client = new HttpClient())

{

var response = await client.GetAsync("https://api.example.com/data");

return await response.Content.ReadAsStringAsync();

}

}

If an exception occurs while fetching data from the API, it might be challenging to identify the exact location of the issue in the asynchronous code. To mitigate this, use logging to record detailed information about the execution flow, and consider using tools like [Visual Studio’s async debugging features](https://docs.microsoft.com/en-us/visualstudio/debugger/using-the-async-feature?view=vs-2019) to help trace issues.

1. Potential Performance Issues

Improper use of async-await can lead to performance issues, such as thread pool starvation or deadlocks, if not used correctly. For example, using Task.Run to offload synchronous code to the thread pool can lead to thread pool starvation:

public async Task<int> CalculateFactorialAsync(int n)

{

// Avoid using Task.Run for computationally-bound operations like this

return await Task.Run(() => CalculateFactorial(n));

}

Instead, consider using parallelism techniques like [Parallel.ForEach](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach?view=net-5.0) or [PLINQ](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq) for computationally-bound operations.

Best Practices for Async and Sync Methods in C#

To make the most of async and sync methods, follow these best practices.

1. Naming Conventions

Async methods should have an “Async” suffix to indicate their asynchronous nature. For example:

public async Task<int> CalculateSumAsync(int a, int b)

{

return await Task.FromResult(a + b);

}

1. Using ConfigureAwait

Always use ConfigureAwait(false) when you don’t need to resume on the original context to avoid potential deadlocks. For example:

public async Task<string> FetchDataAsync()

{

using (HttpClient client = new HttpClient())

{

var response = await client.GetAsync("https://api.example.com/data")

.ConfigureAwait(false);

return await response.Content.ReadAsStringAsync()

.ConfigureAwait(false);

}

}

1. Handling Exceptions

Handle exceptions in async methods using try-catch blocks and ensure proper awaiting of tasks to catch encapsulated exceptions. For example:

public async Task ProcessDataAsync()

{

try

{

string data = await FetchDataAsync();

// Process data here

}

catch (HttpRequestException ex)

{

// Handle exception related to HTTP requests

}

catch (Exception ex)

{

// Handle other exceptions

}

}

Following these best practices and understanding the potential drawbacks, you can effectively use async and sync methods in your C# applications.

**Asynchronous and synchronous programming: What’s the difference?  
(**[**https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/**](https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/) **)**

Synchronous, sometimes called “sync,” and asynchronous, also known as “async,” are two different programming models.

Understanding how these two models differ is critical for:

* Building application programming interfaces (APIs)
* Creating event-based architectures
* Deciding how to handle long-running tasks

Example of a synchronous callback in C#:

public void SynchronousCallbackExample()

{

Action<string> callback = message => Console.WriteLine(message);

DoSomethingSynchronously(callback);

}

public void DoSomethingSynchronously(Action<string> callback)

{

// Simulate a time-consuming operation

for (int i = 0; i < 5; i++)

{

callback($"Iteration {i}");

Thread.Sleep(1000); // Simulate work

}

}

Example of an asynchronous callback in C#:

public async Task AsynchronousCallbackExample()

{

Func<string, Task> callbackAsync = async message =>

{

await Task.Delay(2000); // Simulate non-blocking work asynchronously

Console.WriteLine(message);

};

await DoSomethingAsynchronously(callbackAsync);

}

public async Task DoSomethingAsynchronously(Func<string, Task> callbackAsync)

{

var tasks = new Task[15];

for (int i = 0; i < 15; i++)

{

tasks[i] = callbackAsync($"Iteration {i}");

//await Task.Delay(1000);

}

await Task.WhenAll(tasks);

}

Asynchronous Programming With C#

(<https://www.c-sharpcorner.com/UploadFile/84c85b/asynchronous-programming-with-C-Sharp-5-0/> )

Interestingly enough, any method we normally create in C# is synchronous by default. For example, the following method fetches data from a database and binds it to a TextBox synchronously.  
  
private void LoadData() {

// Create connection

SqlConnection conn = new SqlConnection(@ "network address= .; integrated

security = true; database = EmployeeDb ");

// Create command

string sql = @ "select EmpId,Name

from dbo.EmployeeDetails where EmpID <= 500 ";

// Data binding code goes here

try {

// Open connection

conn.Open();

// Execute query via ExecuteReader

SqlDataReader rdr = cmd.ExecuteReader();

while (rdr.Read()) {

txtReader.AppendText("\nEmpID: ");

txtReader.AppendText(rdr.GetValue(1) + "\t\t" + rdr.GetValue(0));

txtReader.AppendText("\n");

}

} catch (SqlException ex) {

MessageBox.Show(ex.Message + ex.StackTrace, "Exception Details");

} finally {

conn.Close();

}

}

**How bad is synchronous programming?**

* It badly impacts the UI that has just one thread to run its entire user interface code.
* Synchronous behavior leaves end users with a bad user experience and a blocked UI whenever the user attempts to perform some lengthy (time-consuming) operation.

Business Scenario and Problem Statement

Consider a real-world business case in which a UI binds data to the data grid by fetching it from the database. While data is being fetched and bound to the grid the rest of the UI is blocked. Any attempt of interaction with other UI controls will not be evident until the data loading is over. This UI blockage gets over when data fetch-and-binding is completely done. Refer to "Figure 1-1 Synchronous Behavior" below

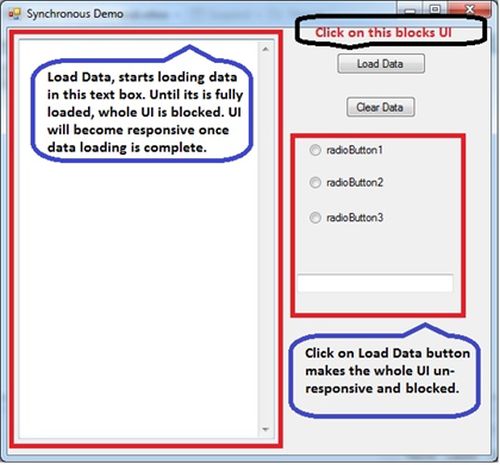


Figure 1-1 Synchronous Behavior

Solution to the Synchronous Problem

A synchronous method call can create a delay in program execution that causes a bad user experience. Hence, an asynchronous approach (threads) will be better. An asynchronous method call (cretion of a thread) will return immediately so that the program can perform other operations while the called method completes its work in certain situations.

The asynchronous method's behavior is different than synchronous ones because an asynchronous method is a separate thread. You create the thread; the thread starts executing, but control is immediately returned back to the thread that called them time; while the other thread continues to execute.

In general, asynchronous programming makes sense in two cases as,

* If you are creating a UI intensive application in which the user experience is the prime concern. In this case, an asynchronous call allows the user interface to remain responsive. Unlike as shown in Figure 1-1.
* If you have other complex or expensive computational work to do, you can continue; interacting with the application UI while wait for the response back from the long-running task.

Asynchronous Patterns in C#

There are various ways to use threads in applications. These recipes are known as Patterns.

**Asynchronous Programming Model Pattern**

* Relies on two corresponding methods to represent an asynchronous operation: BeginMethodName and EndMethodName
* Most often you must have seen this while using delegates or method invocation from a Web Service.

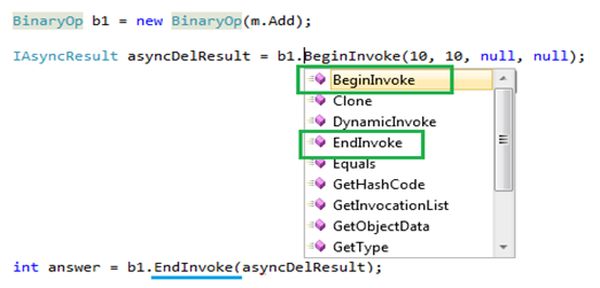


Figure 1-2 APM Pattern

**Event Based Asynchronous Pattern**

* The Event-based Asynchronous Pattern has a single MethodNameAsync method and a corresponding MethodNameCompleted event
* Basically, this pattern enforces a pair of methods and an event to collaborate and help the application execute a thread asynchronously

A screenshot of a computer

AI-generated content may be incorrect.

Figure 1-3 Event Based Pattern

**Task based Asynchronous Pattern**

* The Microsoft .NET Framework 4.0 introduces a new Task Parallel Library (TPL) for parallel computing and asynchronous programming. The namespace is "System.Threading.Tasks".
* A Task can represent an asynchronous operation and a Task provides an abstraction over creating and pooling threads.

A computer code with text

AI-generated content may be incorrect.

Figure 1-4 Task Based Pattern

C# 5.0 async and await based Asynchronous Pattern

* Two new keywords, async and await, were introduced in C# 5.0 and .NET 4.5. These are implemented at the compiler level and built on top of the "System.Threading.Tasks.Task" feature of .NET 4.0.
* To work with async and await, you must have Visual Studio 2012

async void LoadEmployee\_Click(object sender, RoutedEventArgs e) {

// ...

await viewer.LoadEmplployeeAsync();

// ...

}

**Problem with older Asynchrnous Patterns**

With earlier patterns, the programmer needed to do all the plumbing and collaboration between a pair of methods (BeginMethod and EndMethod) or a method and an event (MethodAsync and MethodCompleted) to make them functional; see Figure 1-2 APM Pattern. This approach was a tedious job not only in terms of syntax but also from sequence of statements inside the method body.

C# 5.0 async/await offers a completely different and easy way to do asynchronous programming. With this feature it's no longer the responsibility of the programmer to do the syntax related tedious work, rather this is now done by the keywords (C# 5.0 async / await) provided by the programming language.

As a result, asynchronous code is easy to implement and retain its logical structure. Hence now it is as easy as writing your normal method without concern of any extra plumbing and so on. As shown in other asynchronous patterns in which you need to deal with a pair of methods or a combination of methods and events and so on.

**Business Scenario**

Consider a real-world business case, a WPF UI binding data to the data grid by fetching a large number of rows from a database. While data is being fetched and bound to a grid, the rest of the UI should continue to be responsive. Any attempt at interaction with other UI controls must **not** be blocked and data loading and binding must continue in parallel.. Refer to "Figure 1-1 Synchronous Behavior" below.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 1-5 Asynchronous Behavior

**Let's Code**

If you look at the code below, it looks like normal code as shown at the very beginning of this article. The differences worth noting are highlighted in yellow in the code block below.

Private async void LoadCustomersAsync() {

using(EmployeeDbEntities ent = new EmployeeDbEntities()) {

IdbConnection conn = (ent.Connection as EntityConnection).StoreConnection;

conn.Open();

using(DbCommand cmd = (DbCommand) conn.CreateCommand()) {

var query = from p in ent.EmployeeDetails

where p.Name.Contains("FN") && p.SurName.Contains("SN") && (p.Name + p.SurName).Length > 3

select p;

//Convert linq query to SQL statement for CommandText

string str = ((ObjectQuery) query).ToTraceString();

cmd.CommandText = str;

// Invoke Async flavor of ExecuteReader

var task = await cmd.ExecuteReaderAsync();

//translate retieved data to entity customer

var cust1 = await Task.Run(

() => ent.Translate < EmployeeDetails > (task).ToList < EmployeeDetails > ());

employeesDataGrid.ItemsSource = cust1;

}

}

}

As you noticed, the flow looks very natural and no extra plumbing appears in the code. Except async/await, task and of course the asynchronous flavor of the main function that is retrieving data from the database; in our case, ExecuteReaderAsync() is the method.

This code will allow you to perform UI interaction; when data is being fetched and grid binding is taking place, refer to the Figure 1-6 async/await in action.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 1-6 async/await in action (as you can see in image 36K + rows pulled)

**Legacy Operations**

Microsoft suggests that with the release of .NET 4.5, the following commonly used methods should be considered as legacy operations. When possible and if you are usng .NET 4.5 then you must use async and await to do asynchronous programming in your application.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 1-7 Legacy Operations

**What if you don't have Visual Studio 2012**

Since Visual Studio 2012 is still not adopted by many development teams in various organizations and many developers still use Visual Studio 2010. So, can they use async and await syntax there?

Microsoft released an async CTP that is supposed to work well with Visual Studio 2010 (without SP1) and allow the developers to use the same syntax.

Search for "async CTP" in Bing or Google.

A screenshot of a computer

AI-generated content may be incorrect.

**Side-by-Side Comparison of various ways techniques**

A table with text on it

AI-generated content may be incorrect.