Anthony Gibah: Seamless Digital

Part 2: Theory Questions

1. What issues can arise from using async void methods in C#? When should you use async Task instead, and why?

   Ans: async void in C# is asynchronous and cannot be awaited, therefore, it is error-prone because loggers and try-catch statements are difficult to implement. Async Task can be awaited; therefore, they are a better option to be used in C#. an example f this is found in the ToDo project in Part 1.

```
// GET: api/ToDos
[HttpGet]
public async Task<ActionResult<IEnumerable<ToDoItem>>>
GetToDoItems()
{
    try
    {
        var todoResponses = await
_toDoService.FetchToDoList();
        return Ok(todoResponses);
    }
    catch (Exception e)
    {
        _logger.LogError(e, "An error occurred while
fetching ToDo items.");
        return StatusCode(500, "An error occurred while
processing your request.");
    }
}
```

2. Explain the potential problems with using static variables in a multi-threaded or web application context. How can this affect application behavior.

   Ans: Using static variables in a mult-threaded or web application context means that multiple processes have access and can write to the static variables. Since these variables are shared by all instances and threads, it causes irregularities in values. The solution is to ensure that only one thread controls writing to the static variable.

3. What is the difference between the == operator and the .Equals() method in C# when comparing objects.

   Ans: The == operator compares the reference of the objects. This means it checks if it is that specific memory location that is being compared. The .Equals() on the other hand just compares the state, which is the instance variables of the object. For exact comparison, make use of ==.

4. What happens if you do not implement the IDisposable interface for a class that uses unmanaged resources. What are the consequences, how can you prevent them?

   Ans: The IDisposable ensures that resources are released promptly to prevent leakage of resources. The consequences include resources leaking to other resources in the application causing errors. It can be used by implementing the Dispose method in the IDisposable interface.

5. Explain the key differences between Common Table Expressions (CTEs) and temprorary tables. In which scenarios whould you prefer one over the other?

   Ans: Common Table Expressions (CTEs) are defined within a single query and exist only for the duration of that query's execution. They are particularly suitable for recursive queries or for simplifying complex query logic without the need to create physical structures. On the other hand, temporary tables are stored in the temp database and persist for the duration of the session or batch. Temporary tables can be indexed and are ideal for scenarios where the data needs to be reused across multiple queries.

   I prefer to use CTEs for simpler, transient data operations and recursive queries, while I use temporary tables for performance optimization or when data is reused across multiple queries.

6. Do you see any issues with this LNQ query?

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
}
var books = dbContext.Books
    .Where(p => IsRecommended(p.Title))
    .ToList();
bool IsRecommended(string title)
{
    return title.StartsWith("A") && title.EndsWith("Z");
}
```

   Ans: Yes, there is an issue with this LINQ query. The IsRecommended method cannot be translated into SQL because LINQ to Entities requires all expressions to be translatable into SQL queries. Methods defined in C# code, like IsRecommended, are not supported because the database context cannot interpret them.

Part 3: Opinion Questions

What is your stance on exception handling in backend applications and how do you typically implement it in your C# projects?

Ans: Exception handling is vital to every program. It is particularly essential in backend applications to handle errors correctly for debugging purposes. In my backend applications, I typically create Custom Exception classes that extend the Exception Class that .Net provides. This helps me to better understand the Exceptions and the point of their occurrence.

Example of a simple custom Exception handler:

```
public class ExceptionHandler: Exception
{
    public string error;

    public ExceptionHandler(string error)
        : base(error)
    {
        this.error = error;
    }

    public ExceptionHandler(string error, Exception inner)
        : base(error, inner)
    {
        this.error = error;
    }
}
```

The code above shows where the Exception is being extended. This approach enables me to define more specific exception types, making it easier to understand the nature of the error and pinpoint its origin. Additionally, I make use of structured try-catch blocks and logging mechanisms to capture exceptions and provide meaningful feedback while ensuring sensitive information is not exposed.