Lab 02 - Adding Async to an Existing Application

Objectives

This lab takes an existing application and adds async to key methods (in order to avoid blocking). Asynchrony is bubbled up through various levels of the application.

Application Overview

The "Starter" folder contains the code files for this lab.

Visual Studio 2022: Open the "Mazes.sln" solution.
Visual Studio Code: Open the "Starter" folder in VS Code.

This is a web application that produces mazes. There are a number of algorithms that generate mazes with different speeds and biases (a bias could be a tendency to include diagonal paths, longer/shorter paths, or twistier/straighter paths).

The "MazeWeb" project contains a web application that creates and shows the mazes.

The "MazeGeneration" project contains the "MazeGenerator" that creates the mazes.

The "Algorithms" project contains a number of different maze algorithms used to produce the mazes.

The "MazeGrid" project contains the infrastructure for the grid, cells, text outputs, and graphical outputs. This code was originally taken from the book "Mazes for Programmers" and translated from Ruby to C#.

Visual Studio 2022:
Build and run the application (by using F5 or "Start Debugging" from the Debug menu). The browser window will pop up with parameters to select.

Visual Studio Code:
Build and run the application. One way to do this is from the command line.

- Open a command prompt (PowerShell, Terminal, or cmd.exe) and navigate to the project folder: *[working_directory]/Lab02/Starter/MazeWeb/*
- Build the application using "dotnet build".

```
PS C:..\Lab02\Starter\MazeWeb> dotnet build
```

- Run the application using "dotnet run".

```
PS C:..\Lab02\Starter\MazeWeb> dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[14]
```

```
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:..\Lab02\Starter\MazeWeb
```

- Open a web browser and navigate to "https://localhost:5001".

> Note: You may get a runtime error if you do not have a developer certificate on your machine (for HTTPS). To create a developer certificate, type the following:
>
> ```
> dotnet dev-certs https
> ```
>
> Then to trust the certificate, type the following:
>
> ```
> dotnet dev-certs https --trust
> ```
>
> You will get a popup verifying that you want to trust the certificate on your machine. Click "Yes".
>
> *Note: the trust step works on Windows and macOS only. Check the Microsoft docs for "dotnet dev-certs" for more information on using this on Linux.*

With the website open:
Select some values from the drop-downs and click the button. This will show a maze created using the selected parameter. The different shades of color are a a "heat map" with the darker areas being furthest from the center of the maze and lighter areas being closer to the center of the maze.

Try different maze sizes and algorithms. Larger mazes will take longer to generate (compare "75" to "175"). Also different algorithms take different times. "Sidewinder" is a deterministic algorithm so will always run with a consistent speed. "Hunt and Kill" and "Recursive Backtracker" both have random elements, so will take different amounts of time to run.

Lab Goals

There are two "slow" areas of this application:

- Generating the maze
- Creating the output image

Because we do not want to block the incoming request threads in our web application, we will make both of these methods asynchronous.

The "MazeGenerator" class creates the grid and uses the maze algorithm to create the maze. The "Generate" method is one that we want to make asynchronous.

The "Grid" class creates the output image. The "ToPng" method is one that we want to make asynchronous.

With these methods updated, we will also change the upstream methods and use/add asynchrony as necessary.

Current Classes

MazeGeneration/MazeGenerator.cs:

```
public class MazeGenerator : IMazeGenerator
{
    public void GenerateMaze()
    {
        algorithm.CreateMaze(mazeGrid);
        isGenerated = true;
    }
    public string GetTextMaze(bool includePath = false) { ... }
    public Image GetGraphicalMaze(bool includeHeatMap = false) { ... }
    ...
}
```

MazeGrid/Grid.cs:

```
public abstract class Grid
{
    public Image ToPng(int cellSize = 10) { ... }
    ...
}
```

MazeWeb/Controllers/MazeController.cs:

```
public class MazeController : Controller
{
    public IActionResult Index(int size, string algo, MazeColor color) { ... }
    private int GetSize(int size) { ... }
    private IMazeAlgorithm GetAlgorithm(string algo) { ... }
    private Image GenerateMazeImage(int mazeSize, IMazeAlgorithm algorithm,
MazeColor color) { ... }
    private static byte[] ConvertToByteArray(Image img) { ... }
}
```

Hints

If an asynchronous version of a method does not exist, look at creating and running a new Task to wrap the existing method.

Maze Generation
In MazeGenerator.cs:

- Make the "Generate" method asynchronous. This can be done within the "Generate" method or by updating "IMazeAlgorithm" (specifically, the "CreateMaze" method).
- Update methods that call the "Generate" method.

In MazeController.cs:

- Modify methods that call the MazeGenerator by either passing the Task along or "awaiting" methods. (Note: there may be functional differences between these two options.)
- Remember that controller actions can be asynchronous.

Image Creation
In Grid.cs:

- Create an asynchronous version of the "ToPng" method.
- Update any calls to "ToPng" to use the asynchronous version.
- As with the Maze Generation task, keep working your way through the method callers until you get to the MazeController class.

BONUS CHALLENGE: Additional Methods

- Note: There is an asynchronous version of the "SaveAsPng" method. Locate this method call and use the asynchronous version.

If you want more assistance, step-by-step instructions are included below. Otherwise, if you'd like to challenge yourself, **STOP READING NOW**

Maze Generation: Step-By-Step

1. Open the "MazeGeneration/MazeGenerator.cs" file.
2. Make the "GenerateMaze" method asynchronous.

Here is the current method:

```
    public void GenerateMaze()
    {
        algorithm.CreateMaze(mazeGrid);
        isGenerated = true;
    }
```

There are 2 options. With the first option, we can wrap the "CreateMaze" call in a Task. This would look like the following:

```
public async Task GenerateMaze()
{
    await Task.Run(() => algorithm.CreateMaze(mazeGrid));
    isGenerated = true;
}
```

Although this works, I would prefer to make the "CreateMaze" method asynchronous. This could allow the different maze algorithms to implement their own asynchrony (for example, if an algorithm "awaits" a Task).

For this option, we will modify the "IMazeAlgorithm" interface.

3. Open the "Algorithms/IMazeAlgoritm.cs" file.

4. Add a new "CreateMazeAsync" method with the following code:

```
public interface IMazeAlgorithm
{
    void CreateMaze(Grid grid);
    Task CreateMazeAsync(Grid grid)
    {
        return Task.Run(() => CreateMaze(grid));
    }
}
```

This adds a method to the interface that has a default implementation. If the implementing "algorithm" class does not include its own CreateMazeAsync method, then the one from the interface is used. As mentioned above, this gives each algorithm the option of adding its own methods that have async built in.

5. Back in the "MazeGenerator" class, update "Generate" to call the new async method.

```
public async Task GenerateMaze()
{
    await algorithm.CreateMazeAsync(mazeGrid);
    isGenerated = true;
}
```

If we try to build the solution, we get an error that the "MazeGenerator" class does not implement all of the members of the "IMazeGenerator" interface. This is because we changed the signature of the "Generate" method.

6. Open the "MazeGeneration/IMazeGenerator.cs" file.

7. Update the method signature of the "Generate" method by changing the return type from "void" to "Task".

```
    public interface IMazeGenerator
    {
        Task GenerateMaze();
        Image GetGraphicalMaze(bool includeHeatMap = false);
        string GetTextMaze(bool includePath = false);
    }
```

At this point, the application will build. But if we run the application, we will get a runtime error when we try to generate a maze.

To see why, let's look at the "GetGraphicalMaze" method on the "MazeGenerator" class.

```
    public Image GetGraphicalMaze(bool includeHeatMap = false)
    {
        if (!isGenerated)
            GenerateMaze();

        if (includeHeatMap)
        {
            Cell start = mazeGrid.GetCell(mazeGrid.Rows / 2, mazeGrid.Columns /
2);
            mazeGrid.distances = start.GetDistances();
        }
        var result = mazeGrid.ToPng(30);
        return result;
    }
```

Notice that "GenerateMaze" is called in this method, but it is not awaited. This means that the maze generation gets kicked off, but the rest of the method runs before that generation is complete.

To fix this, we can "await" the "GenerateMaze" call.

8. Add "await" before "GenerateMaze" and update the return type to "Task<Image>". (Don't forget to add the "async" modifier as well.)

```
    public async Task<Image> GetGraphicalMaze(bool includeHeatMap = false)
    {
        if (!isGenerated)
            await GenerateMaze();

        ... // rest of the method not shown
    }
```

9. Make the same change to the "GetTextMaze" method:

```
    public async Task<string> GetTextMaze(bool includePath = false)
    {
        if (!isGenerated)
            await GenerateMaze();

        ... // rest of the method not shown
    }
```

Note: "GetTextMaze" is used for console applications to generate a text-based maze. Although it is not used in the web application, we should update this method while we are here.

10. Update the method signatures of "IMazeGenerator".

Since we changed the return types for "GetGraphicalMaze" and "GetTextMaze", we need to update those methods in the interface as well.

```
    public interface IMazeGenerator
    {
        Task GenerateMaze();
        Task<Image> GetGraphicalMaze(bool includeHeatMap = false);
        Task<string> GetTextMaze(bool includePath = false);
    }
```

If we try to build at this point, we get an error in the "MazeController" of the web application. Our next steps will be to continue making methods async inside the controller.

11. Open the "MazeWeb/Controllers/MazeController.cs" file.

12. In the "GenerateMazeImage" method, "await" the call to "GetGraphicalMaze" (and change the method signature as needed).

```
    private async Task<Image> GenerateMazeImage(int mazeSize, IMazeAlgorithm
algorithm, MazeColor color)
    {
        IMazeGenerator generator =
            new MazeGenerator(
                new ColorGrid(mazeSize, mazeSize, color),
                algorithm);

        Image maze = await generator.GetGraphicalMaze(true);
        return maze;
    }
```

If we try to build at this point, we get an error in the "MazeController". We need to continue adding async to the "Index" method.

13. In the "Index" method, "await" the "GenerateMazeImage" method (and update the method signature).

```
    public async Task<IActionResult> Index(int size, string algo, MazeColor color)
    {
        int mazeSize = GetSize(size);
        IMazeAlgorithm algorithm = GetAlgorithm(algo);
        Image mazeImage = await GenerateMazeImage(mazeSize, algorithm, color);
        byte[] byteArray = ConvertToByteArray(mazeImage);
        return File(byteArray, "image/png");
    }
```

> *Note: "Index" is a controller action. The good news is that ASP.NET Core can handle asynchronous controller actions. We do not need to add any additional code or configuration for this action to work.*

14. Build and run

At this point, the solution builds successfully. When we run the application, we get the same visual behavior as before. However, behind the scenes, the maze generation is happening on a separate thread. This means that we are not blocking a request thread during the maze generation.

Next, we will do the same thing with the image creation.

Image Creation: Step-By-Step

1. Open the "MazeGrid/Grid.cs" file.

2. Note the "ToPng" method.

This method generates a .png file for the maze. This process can take a while to complete, particularly for large grid sizes.

To avoid blocking the request thread, we will move this processing to a Task. But instead of modifying the existing "ToPng" method, we will add a new method wrapper.

3. Add a new method called "ToPngAsync" that wraps the "ToPng" method.

```
    public Task<Image> ToPngAsync(int cellSize = 10)
    {
        return Task.Run(() => ToPng(cellSize));
    }
```

4. Just like above, create a "ToStringAsync" method that wraps the "ToString" method.

```
    public Task<string> ToStringAsync()
    {
        return Task.Run(() => ToString());
    }
```

Note: This method is used for console applications and text output. As earlier, even though the web application does not use this method, we will update it while we are here.

5. In the "GetGraphicalMaze" method of the "MazeGenerator" class, change "ToPng" to "ToPngAsync" (and be sure to "await" it).

```
    public async Task<Image> GetGraphicalMaze(bool includeHeatMap = false)
    {
        if (!isGenerated)
            await GenerateMaze();

        if (includeHeatMap)
        {
            Cell start = mazeGrid.GetCell(mazeGrid.Rows / 2, mazeGrid.Columns /
2);
            mazeGrid.distances = start.GetDistances();
        }
        var result = await mazeGrid.ToPngAsync(30);
        return result;
    }
```

Since the "GetGraphicalMaze" method is already asynchronous, all we need to do is add the "await" and change "ToPng" to "ToPngAsync".

6. Change "GetTextMaze" to use "ToStringAsync" instead of "ToString".

```
    public async Task<string> GetTextMaze(bool includePath = false)
    {
        if (!isGenerated)
            await GenerateMaze();

        if (includePath)
        {
            Cell start = mazeGrid.GetCell(mazeGrid.Rows / 2, mazeGrid.Columns /
2);
            mazeGrid.path =
start.GetDistances().PathTo(mazeGrid.GetCell(mazeGrid.Rows - 1, 0));
        }

        var result = await mazeGrid.ToStringAsync();
        return result;
    }
```

7. Build and run.

At this point, the solution builds successfully. When we run the application, we get the same visual behavior as before. However, behind the scenes, the image creation is happening on a separate thread. This means that we are not blocking a request thread during the image creation.

BONUS CHALLENGE: Additional Methods

Now that we have added async to our application, we can look for other places where we can update to asynchronous methods. One example is in the MazeController class.

1. Open the "MazeWeb/Controllers/MazeController.cs" file.

2. Look at the "ConvertToByteArray" method.

```
private static byte[] ConvertToByteArray(Image img)
{
    using var stream = new MemoryStream();
    img.SaveAsPng(stream);
    return stream.ToArray();
}
```

This method takes an image and converts it to a byte array. The method currently uses a "SaveAsPng" method. If we look at the other options available, we will find a "SaveAsPngAsync" method.

3. Update "ConvertToByteArray" to use the "SaveAsPngAsync" method.

```
private static async Task<byte[]> ConvertToByteArray(Image img)
{
    using var stream = new MemoryStream();
    await img.SaveAsPngAsync(stream);
    return stream.ToArray();
}
```

4. The last step is to "await" this method in the "Index" action.

```
public async Task<IActionResult> Index(int size, string algo, MazeColor color)
{
    int mazeSize = GetSize(size);
    IMazeAlgorithm algorithm = GetAlgorithm(algo);
    Image mazeImage = await GenerateMazeImage(mazeSize, algorithm, color);
    byte[] byteArray = await ConvertToByteArray(mazeImage);
    return File(byteArray, "image/png");
}
```

Since the "Index" action is already asynchronous, we just need to add the "await" before the call to "ConvertToByteArray".

5. Build and run.

The application builds and runs as expected. Try out different maze sizes and algorithms and note that larger mazes take longer to generate and some algorithms take longer than others.

Now that the application has async methods, are there other async methods from the standard libraries that can be used?

Take a look through the code and see what you can find.

End of Lab 02 - Adding Async to an Existing Application
