APPENDIX

CHAPTER 1: HELLO C#

ACTIVITY 1.01: CREATING A GUESSING GAME

Solution:

1. Open Command Prompt and type **`dotnet new console -p Activity1_01`** to create the activity project.
2. Create a variable named **`numberToBeGuessed`**, another named **`remainingChances`**, and a third one called **`numberFound`**, as follows:

```
using System;

var numberToBeGuessed = new Random().Next(0, 10);
var remainingChances = 5;
var numberFound = false;
```

> **NOTE**
>
> You could vary the range for the numbers to be guessed, but, for this activity, you will be using **`0`** to **`10`** as the range, and the user should get five chances to guess the correct number. Additionally, the **`numberFound`** variable should be initialized with the value **`false`**.

3. Now, create a **`while`** loop to ask the user for a number, until they get it right or they run out of chances:

`Program.cs`

```
Console.WriteLine("Welcome to Packt's C# Workshop Guessing Game.");

while (remainingChances > 0 && !numberFound)
{
    Console.WriteLine($"\n You have {remainingChances} chances. Please type a
number between 0 and 10 to try to guess the number generated for you.");

    var number = int.Parse(Console.ReadLine());

    if (number == numberToBeGuessed)
    {
        numberFound = true;
    }
    else
    {
        remainingChances--;
```

`You can find the complete code here:` https://packt.link/BbCm8.

4. Run the app using the **`dotnet run`** command. You will see output like the following:

```
Welcome to Packt's C# Workshop Guessing Game.
You have 5 chances. Please type a number between 0 and 10 to try to
guess the number generated for you.
5
You have 4 chances. Please type a number between 0 and 10 to try to
guess the number generated for you.
3
Congrats! You've guessed the number correctly with 4 chances left.
```

   Note that your output will vary, depending on the number generated randomly by the compiler.

In this activity, you created a guessing game. In this game, first, a random number from one to 10 was generated. The console then prompted you to input a number and then guess which random number has been generated (you were given a maximum of five chances). Upon every incorrect input, a warning message was displayed, letting you know how many chances you are left with. If all five chances were exhausted with incorrect guesses, the program terminated. However, if you guessed it correctly, a success message was displayed before the program terminated.

Chapter 2: Building Quality Object-Oriented Code

Activity 2.01: Merging Two Circles

Solution:

1. Create a value **object**, **`Circle`**, with an immutable radius:

```
public struct Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }
```

2. Add a property to calculate the area:

```
    public double Area => Math.PI * Radius * Radius;
```

3. To add the two circles' areas together, implement the plus (**+**) operator:

```
    public static Circle operator +(Circle circle1, Circle circle2)
    {
        var newArea = circle1.Area + circle2.Area;
        var newRadius = Math.Sqrt((newArea / Math.PI));

        return new Circle(newRadius);
    }
}
```

4. Add a **`Solution`** class with a **`Main`** method to demonstrate that the operator works:

```
public static class Solution
{
    public static void Main()
    {
        var circle1 = new Circle(3);
        var circle2 = new Circle(3);
        var circle3 = circle1 + circle2;

        Console.WriteLine($"Adding circles of radius of {circle1.
Radius} and {circle2.Radius} " +
                          $"results in a new circle with a radius
{circle3.Radius}");
    }
}
```

5. Run the **`Main`** method and the result should be as follows:

```
Adding circles of radius of 3 and 3 results in a new circle with a
radius 4.242640687119285
```

In this activity, you created classes and override operators to solve the following mathematics problem: A portion of pizza dough can be used to create two circular pizza bites each with a radius of three centimeters. What would be the radius of a single pizza bite made from the same amount of dough? You can assume that all the pizza bites are the same thickness.

Chapter 3: Delegates, Events, and Lambdas

Activity 3.01: Creating a Web File Downloader

Solution:

1. Change to the **Chapter03** folder and create a new console app, called **Activity01**, using the CLI **dotnet** command:

```
source\Chapter03>dotnet new console -o Activity01
```

2. Open **Chapter03\ Activity01.csproj** and replace the entire file with these settings:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
</Project>
```

3. Open **Activity01\Program.cs** and clear the contents.

4. You will need to include the following **using** statements:

```
using System;
using System.IO;
using System.Net;
using System.Threading;
```

5. You will use **Chapter03.Activity01** as the namespace:

```
namespace Chapter03.Activity01
{
```

6. Add the progress reporting class (**DownloadProgressChangedEventArgs** was suggested as a name) to help report the progress percentage and bytes received:

```
    public class DownloadProgressChangedEventArgs
    {
        public DownloadProgressChangedEventArgs(int 
progressPercentage, long bytesReceived)
        {
            ProgressPercentage = progressPercentage;
            BytesReceived = bytesReceived;
        }

        public long BytesReceived { get; init; }
        public int ProgressPercentage { get;  init;}
    }
```

7. Now add the **`WebClientAdapter`** class (which hides the internal usage of **`WebClient`**):

```
    public class WebClientAdapter
    {
        public event EventHandler DownloadCompleted;
        public event EventHandler<DownloadProgressChangedEventArgs> 
DownloadProgressChanged;
        public event EventHandler<string> InvalidUrlRequested;
```

   This will act as a **publisher**, so you need to add the three events. Notice that **`DownloadCompleted`** is based on **`EventHandler`**, as you do not need to pass any extra information when publishing the event, whereas the other two events do include extra details.

8. Add a **`DownloadFile`** method is added, which is passed a **`url`** string (as entered by the user) and a destination filename:

```
        public IDisposable DownloadFile(string url, string 
destination)
        {
            if (!Uri.TryCreate(url, UriKind.Absolute, out var uri))
            {
                InvalidUrlRequested?.Invoke(this, url);
                return null;
            }
```

   You need to convert the **`url`** string into a **`Uri`** class using **`Uri.TryCreate`**. You have specified **`UriKind.Absolute`**, as you have a fully qualified web address. If this call fails, you invoke the **`InvalidUrlRequested`** event.

9. Create a **`WebClient`** instance used to initiate the actual file download:

```
            var client = new WebClient();
```

   `WebClient` has a **`DownloadFileCompleted`** event that you subscribe to, to indicate when the download is complete.

10. Define a lambda, which you use to publish your own **DownloadCompleted** event. Remember, you are trying to hide your internal use of **WebClient** with your own events:

```
            client.DownloadFileCompleted += (sender, args) =>
                DownloadCompleted?.Invoke(this, EventArgs.Empty);
```

11. The **WebClient** class' **DownloadProgressChanged** event is subscribed to so that you are given progress updates, which you adapt and publish via your event of the same name:

```
            client.DownloadProgressChanged += (sender, args) =>
                DownloadProgressChanged?.Invoke(this,
                    new DownloadProgressChangedEventArgs(args.
ProgressPercentage, args.BytesReceived));
```

12. Finally, call **WebClient.DownloadFileAsync**, passing in the **URI** and the file destination to start the download request:

```
            client.DownloadFileAsync(uri, destination);

            return client;
        }
    }
```

13. Now add the console app that uses **WebClientAdapter**. Define a new **Program** class with the standard **void Main** entry point:

```
    public class Program
    {
        public static void Main()
        {
```

14. Add a **do** loop that prompts for a URL to be entered:

```
                          string input;
            do
            {
                Console.WriteLine("Enter a URL:");
                input = Console.ReadLine();

                if (!string.IsNullOrEmpty(input))
```

15. Create a temporary filename to store the download based on the source URL, so you extract the last **/** symbol, which is appended to **Path.GetTempPath**:

```
                {
                    string destination;
                    var lastSlash = input.LastIndexOf("/");
                    if (lastSlash > -1)
                    {
                        destination = Path.Join(Path.GetTempPath(), 
input.Substring(lastSlash + 1));
                    }
                    else
                    {
                        destination = Path.GetTempFileName();
                    }
```

    If there is no trailing **/** symbol, you ask for a filename using **Path.GetTempFileName**.

16. Pass the input URL and destination filename to a **Download** method. Define that next:

```
                    Download(input, destination);
                }
            } while (input != string.Empty);
        }
```

17. Next make the **Download** method create a new instance of **WebClientAdapter**:

```
                private static void Download(string url, string 
destination)
        {
            var client = new WebClientAdapter();
```

    You need a way to wait for the requested download to finish before you leave this **Download** method.

18. Now use the **ManualResetEventSlim** to signal that an operation has finished:

```
                        var waiter = new ManualResetEventSlim();
            using (waiter)
            {
```

As with **WebClient**, the **ManualResetEventSlim** class implements the **IDisposable** interface, which means it is effective to wrap it in a **using** statement to ensure that memory and other resources are cleaned up after use.

19. Subscribe the **WebClientAdaptor.InvalidUrlRequested** event to using a lambda:

```
                    client.InvalidUrlRequested += (sender, args) =>
                    {
                        var oldColor = Console.BackgroundColor;
                        Console.BackgroundColor = ConsoleColor.DarkRed;
                        Console.WriteLine($"Invalid URL {args}");
                        Console.BackgroundColor = oldColor;
                    };
```

This makes a note of the current console background color, which temporarily changes to red before writing a warning message.

20. Subscribe to the **DownloadProgressChanged** event, so that you can offer progress reports as the download progresses:

```
                    client.DownloadProgressChanged += (sender, args) =>
                    {
                      Console.WriteLine( $"Downloading...{args.
ProgressPercentage}% complete ({args.BytesReceived:N0} bytes)");
                    };
```

21. For the final event, call **waiter.Set**, which will signal to the **waiter** that you have finished:

```
                                            client.DownloadCompleted +=
(sender, args) =>
                    {
                        Console.WriteLine($"Downloaded to {destination}
");
                        waiter.Set();
                    };
```

22. Next call **WebClientAdapter.DownloadFile**, and you are returned another **IDisposable** object:

```
                                           Console.WriteLine($"Downloading
{url}...");
                    var request = client.DownloadFile(url, destination);
                    if (request == null)
                        return;
```

If the web address turns out to be an invalid one, then this will be null, so you do not need to proceed.

23. Wrap the **`request`** object in a **`using`** statement:

```
                using (request)
                {
                    if (!waiter.Wait(TimeSpan.FromSeconds(10D)))
                    {
                        Console.WriteLine($"Timedout downloading 
{url}");
                    }
                }
            }
        }
    }
}
```

This helps you to call **`waiter.Wait`** to pause for up to 10 seconds or until you receive a **`waiter.Set`** call triggered via the download event.

24. Run the console app with various download requests to produce this output:

```
Enter a URL:
https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/csvfiles/
StormEvents_details-ftp_v1.0_d1950_c20170120.csv.gz
Downloading https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/
csvfiles/StormEvents_details-ftp_v1.0_d1950_c20170120.csv.gz...
Downloading...73% complete (7,758 bytes)
Downloading...77% complete (8,192 bytes)
Downloading...100% complete (10,597 bytes)
Downloaded to C:\Temp\StormEvents_details-ftp_v1.0_d1950_c20170120.
csv.gz

Enter a URL:
https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/csvfiles/
StormEvents_details-ftp_v1.0_d1954_c20160223.csv.gz
Downloading https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/
csvfiles/StormEvents_details-ftp_v1.0_d1954_c20160223.csv.gz...
Downloading...29% complete (7,758 bytes)
Downloading...31% complete (8,192 bytes)
Downloading...54% complete (14,238 bytes)
Downloading...62% complete (16,384 bytes)
Downloading...84% complete (22,238 bytes)
Downloading...93% complete (24,576 bytes)
```

```
    Downloading...100% complete (26,220 bytes)
    Downloaded to C:\Temp\StormEvents_details-ftp_v1.0_d1954_c20160223.
    csv.gz
```

In this activity, you planned to investigate patterns in US storm events. To do this, you downloaded storm event datasets from online sources for later analysis. The National Oceanic and Atmospheric Administration were accessed from https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/csvfiles.

You created a .NET Core console app that allowed a web address to be entered, the contents of which are downloaded to a local disk. To be as user-friendly as possible, the application used events that signal when an invalid address is entered, the progress of a download, and when it completes.

Chapter 4: Data Structures and LINQ

Activity 4.01: Treasury Flight Data Analysis

Solution:

1. Create a new folder called **`Activities`** in the **`Chapter04`** folder.
2. Add a new folder called **`Activity01`** to that folder.
3. Add a new class file called **`Flight.cs`**:

```
namespace Chapter04.Activities.Activity01
{
    internal record Flight (string Agency, double PaidFare,
        string TripType, string RoutingType, string TicketClass,
        string DepartureDate, string Origin, string Destination,
        string DestinationCountry,
        string Carrier);
}
```

   This will be a **`Record`** class with fields that match those in the flight data. A **`record`** type is used as it offers a simple type purely to hold data rather than any form of behavior.

4. Add a new class file called **`FlightLoader.cs`**. This class is responsible for downloading or importing data.
5. Add an interface that exposes the **`Import`** and **`Download`** methods, as follows:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;

namespace Chapter04.Activities.Activity01
{
    internal interface IFlightLoader
    {
        IList<Flight> Import(string filePath);

        IList<Flight> Download(string url, string filePath);
    }
```

6. Now for the **FlightLoader** implementation, use a **static** class to define the index of known field positions in the data file:

```
    internal class FlightLoader : IFlightLoader
    {
        private static class ImportFieldIndex
        {
            public const int Agency = 0;
            public const int PaidFare = 1;
            public const int TripType = 2;
            public const int RoutingType = 3;
            public const int TicketClass = 4;
            public const int DepartureDate = 5;
            public const int Origin = 6;
            public const int Destination = 7;
            public const int DestinationCountry = 8;
            public const int Carrier = 9;
        }
```

   This makes it easier to handle any futures changes in the layout of the data.

7. The **Import** interface must be passed the name of the local file to import. So, **Skip** the first line in the file as that is a header.

8. Split each line into a **string** array, one element for each field:

```
        public IList<Flight> Import(string filePath)
        {
            var flights = new List<Flight>();

            // Skip the header line
            foreach (var line in File.ReadLines(filePath)
                .Skip(1)
                .Where(ln => !string.IsNullOrWhiteSpace(ln)))
            {
                var fields = line.Split(",");
```

   Once each line is split, the number of elements is expected to be exactly nine, and you must ensure that **Agency** (at element **0**) is not a null or empty string.

9. If that is the case, use the **`continue`** statement to skip the current line and move onto the next, as follows:

```
                    if (fields.Length < ImportFieldIndex.Carrier ||
                        string.IsNullOrEmpty(fields[ImportFieldIndex.
Agency]))
                        continue;
```

10. Parse the **`PaidFare`** element from a string into a **`double`** value and create a new **`Flight`** record:

`FlightLoader.cs`

```
                double.TryParse(fields[ImportFieldIndex.PaidFare],
                                out double paidFare);
                var flight = new Flight(
                    fields[ImportFieldIndex.Agency],
                    paidFare,
                    fields[ImportFieldIndex.TripType],
                    fields[ImportFieldIndex.RoutingType],
                    fields[ImportFieldIndex.TicketClass],
                    fields[ImportFieldIndex.DepartureDate],
                    fields[ImportFieldIndex.Origin],
                    fields[ImportFieldIndex.Destination],
                    fields[ImportFieldIndex.DestinationCountry],
                    fields[ImportFieldIndex.Carrier]);
```

`You can find the complete code here:` https://packt.link/waV1y.

11. Next, pass a URL and destination file to the **`Download`** method.

12. Use **`WebClient.DownloadFile`** to download the data file and then defer to **`Import`** to process the downloaded file, as follows:

```
        public IList<Flight> Download(string url, string filePath)
        {
            using var client = new WebClient();
            client.DownloadFile(url, filePath);

            return Import(filePath);
        }
    }
}
```

13. Add a new class file called **FilterCriteria.cs** as follows:

```
using System;
namespace Chapter04.Activities.Activity01
{
    internal enum FilterCriteriaType
    {
        Class,
        Origin,
        Destination
    }

    internal record FilterCriteria(FilterCriteriaType Filter,
                                   string Operand);
}
```

This class contains an **enum** and a **string** value, **Operand**. Use a **List** of these to define the user's filter criteria. The **enum** determines the type of filter that has been requested, and **Operand** represents the **string** to search for, such as **Economy**.

14. Now, for the main filtering class, add a new class file called **FlightQuery.cs**:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Chapter04.Activities.Activity01
{
    internal class FlightQuery
    {
        private readonly List<Flight> _flights = new();
        private readonly List<FilterCriteria> _filters = new();
        private readonly IFlightLoader _loader;

        public FlightQuery(IFlightLoader loader)
        {
            _loader = loader;
        }
```

Here the constructor will be passed a **FlightLoader** instance (declared using the **IFlightLoader** interface). The **_flights** list contains the data imported via the **FlightLoader** instance. **_filters** is a list **List<FilterCriteria>** instance that represents each of the criteria items that are added every time the user specifies a new filter condition.

15. Next, let the **Import** and **Download** methods be called by the console at startup, allowing previously downloaded data to be processed via the **_loader** instance:

```
        public void Import(string path)
        {
            _flights.Clear();
            _flights.AddRange(_loader.Import(path));
        }

        public void Download(string url, string filePath)
        {
            _flights.Clear();
            _flights.AddRange(_loader.Download(url, filePath));
        }
```

The **Count** variable simply returns the number of flight records that have been imported. This is defined as follows:

```
        public int Count => _flights.Count;
```

16. When you specify a filter to add, the console should call **AddFilter**, passing an **enum** to define the criteria type and the **string** value being filtered for. The code for this can be added as follows:

```
        public void AddFilter(FilterCriteriaType filter, string 
operand)
        {
            _filters.Add(new FilterCriteria(filter, operand));
            Console.WriteLine($"Added filter: {filter}={operand}");
        }

        public void ClearFilters()
        {
            _filters.Clear();
            Console.WriteLine("Cleared filters");
        }
```

17. Add **RunQuery** as the main method that returns those flights that match the user's criteria.
18. Use the built-in **StringComparer.InvariantCultureIgnoreCase** comparer to ensure string comparison ignores any case differences.
19. Also, define a query variable that calls **Select** on the flights. At present, this results in a filtered result set:

```
        public IList<Flight> RunQuery()
        {
            var comparer = StringComparer.InvariantCultureIgnoreCase;
            var query = _flights.Select(f => f);
```

All types of available filters are string-based, so you must extract all string items.

20. If there are any items to filter, add an extra **Where** call to the query for each type (**Class**, **Destination**, or **Origin**).
21. Use a **Contains** predicate for each **Where** clause, which examines the associated property. Add the following code for this:

FlightQuery.cs

```
            var filterClasses = GetFiltersByType(FilterCriteriaType.Class);
            if (filterClasses.Any())
            {
                query = query.Where(flt =>
                  filterClasses.Contains(flt.TicketClass, comparer));
                Console.WriteLine($"Classes: {FormatFilters(filterClasses)}");
            }

            var filterDestinations = GetFiltersByType(FilterCriteriaType.
Destination);
            if (filterDestinations.Any())
            {
                query = query.Where(flt =>
                   filterDestinations.Contains(flt.Destination, comparer));
                Console.WriteLine($"Destinations:
{FormatFilters(filterDestinations)}");
            }
```

You can find the complete code here: https://packt.link/AVXlv.

22. Next, add the two helper methods used by **RunQuery**:

```
        private IList<string> GetFiltersByType(FilterCriteriaType
filter)
            => _filters
                .Where(f => f.Filter == filter)
                .Select(f => f.Operand)
                .ToList();
```

In the preceding snippet, **`GetFiltersByType`** is passed each of the **`FilterCriteriaType`** enums that represent a known type of criteria type and finds any of these in the list of filters using the **`.Where`** method. For example, if the user added two **`Destination`** criteria such as **`India`** and **`Germany`**, this would result in the two strings **`India`** and **`Germany`** being returned.

23. Now add **`FormatFilters`** as shown in the next snippet:

```
        private string FormatFilters(IEnumerable<string> filterValues)
            => string.Join(" OR ", filterValues);
    }
}
```

This simply joins a list of **`filterValues`** strings into a user-friendly string with the word **OR** between each item, such as **`London OR Dublin`**.

24. Now create the main console app. To do so, add a new class called **`Program.cs`**, which allows the user to input requests and process their commands, as follows:

```
using System;
using System.IO;
using System.Linq;

namespace Chapter04.Activities.Activity01
{
    class Program
    {
        public static void Main()
        {
```

25. Hardcode the download URL and destination filename as follows:

```
            const string FlightDataUrl = "https://www.gov.uk/
government/uploads/system/uploads/attachment_data/file/245855/HMT_-
_2011_Air_Data.csv";
            const string FlightDataFilePath = "hm-treasury-flight-
data-2011.csv";
```

26. Create the main **FlightQuery** class, passing in a **FlightLoader** instance:

```
            var flightQuery = new FlightQuery(new FlightLoader());

            if (File.Exists(FlightDataFilePath))
            {
                Console.WriteLine($"Importing {FlightDataFilePath}");
                flightQuery.Import(FlightDataFilePath);
                Console.WriteLine();
            }
            else
            {
                Console.WriteLine($"Downloading {FlightDataUrl}");
                flightQuery.Download(FlightDataUrl, 
FlightDataFilePath);
                Console.WriteLine();
                Console.WriteLine($"Downloaded to 
{FlightDataFilePath}...");
            }
```

If the app has been run before, you can **Import** the local flight data, or use **Download** otherwise.

27. Show a summary of the records imported and the available commands:

```
            Console.WriteLine($"Found {flightQuery.Count} flight 
records");

            const string GoCommand = "go";
            const string ClearCommand = "clear";
            const string ClassCommand = "class";
            const string OriginCommand = "origin";
            const string DestinationCommand = "destination";

            Console.WriteLine($"Commands: {GoCommand} | 
{ClearCommand} | {ClassCommand} value | {OriginCommand} value | 
{DestinationCommand} value");
```

28. Use a **do** loop to allow the user to enter their command and any required arguments:

```
                string input;
                do
                {
                    Console.Write("Enter a command:");
                    input = Console.ReadLine().ToLower();
```

    When the user enters a command, you can expect that this may also have specified an argument, such as **destination united kingdom**, where **destination** is the command and **united kingdom** is the argument.

29. In order to determine this, use the **IndexOf** method to find the location of the first space character in the input (if any):

```
                    string command;
                    string argument;
                    var spaceIndex = input.IndexOf(' ');
                    if (spaceIndex == -1)
                    {
                        command = input;
                        argument = null;
                    }
                    else
                    {
                        command = input[..spaceIndex].Trim();
                        argument = input[spaceIndex..].Trim();
                    }
```

    This allows a range to extract the first letters into a **command** variable and the last characters into an **argument** variable.

30. Add a **switch** statement to process the commands.

31. For the **GoCommand**, call **RunQuery** and use various aggregation operators on the results returned, as follows:

```
                    switch (command)
                    {
                        case GoCommand:
                            var flights = flightQuery.RunQuery();
                            if (flights.Any())
                            {
```

```
                            var average = flights.Average(fl=>fl.
PaidFare);
                            var min = flights.Min(fl => fl.PaidFare);
                            var max = flights.Max(fl => fl.PaidFare);
                            Console.WriteLine($"Results:
Count={flights.Count}, Avg={average:N2}, Min={min:N2}, Max={max:N2}");
                        }
                        else
                        {
                            Console.WriteLine("No matching flights
found");
                        }
                        break;
```

32. For the remaining commands, clear or add filters as requested.
33. If **ClearCommand** is specified, call the query's **ClearFilters** method, which clears the list of criteria items:

```
                    case ClearCommand:
                        flightQuery.ClearFilters();
                        break;
```

34. If a **class** filter command is specified, call **AddFilter** specifying the **FilterCriteriaType.Class** enum and the string **argument**:

```
                    case ClassCommand:
                        flightQuery.AddFilter(FilterCriteriaType.
Class,
                                              argument);
                        break;
```

35. Use the same pattern for the **Origin** and **Destination** commands.
36. Call **AddFilter**, passing in the required **enum** value and the argument:

```
                    case OriginCommand:
                        flightQuery.AddFilter(FilterCriteriaType.
Origin,
                                              argument);
                        break;

                    case DestinationCommand:
                       flightQuery.AddFilter(FilterCriteriaType.
Destination,
                                             argument);
                        break;
```

```
                }
            } while (input != string.Empty);
        }
    }
}
```

37. Run the console app by adding various filter criteria. This produces the following output:

```
Importing hm-treasury-flight-data-2011.csv
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
..........................
Found 716 flight records
Commands: go | clear | class value | origin value | destination value
```

38. Enter **go** to see the unfiltered results. You should get a result similar to the following:

```
Enter a command:go
Results: Count=716, Avg=693.58, Min=-1,678.19, Max=5,669.23
```

39. Next, add two class filters, for **`economy`** or **`Business Class`**:

```
Enter a command:class economy
Added filter: Class=economy
Enter a command:class Business Class
Added filter: Class=business class
```

As you can see, the output shows the user input in lower case, although the flight query class is case insensitive when searching.

40. Now, enter **go** to show the filtered results:

```
Enter a command:go
Classes: economy OR business class
Results: Count=695, Avg=685.74, Min=-1,678.19, Max=5,669.23
```

41. Add **London** as an origin and **Zurich** as a destination:

```
Enter a command:origin london
Added filter: Origin=london
Enter a command:destination zurich
Added filter: Destination=zurich
```

    Again, the output shows the user input in lowercase.

42. Finally, enter **go** once more to show the filtered aggregate values:

```
Enter a command:go
Classes: economy OR business class
Destinations: zurich
Origins: london
Results: Count=16, Avg=266.92, Min=-74.71, Max=443.49
```

    You have now created a console app that allows the user to download publicly available flight data files and apply statistical analysis to the files.

In this activity, you created a console app to download publicly available flight data files and apply statistical analysis to the files. This data file contained details of flights made by the UK's HM Treasury department between January 1 to December 31, 2011 (there are 714 records.) You used a **WebClient.DownloadFile** to download the data from https://packt.link/y7xm9. This analysis calculated a count of the total records found, along with the average, minimum, and maximum fare paid within that subset.

Chapter 5: Concurrency: Multithreading Parallel and Async Code

Activity 5.01: Creating Images from a Fibonacci Sequence

Solution:

1. In your **`source\Chapter05 folder`**, run the dotnet command to add the **`System.Drawing.Common`** reference:

```
source\Chapter05>dotnet add package System.Drawing.Common
```

2. Download **`FibonacciSequence.cs`** and **`ImageGenerator.cs`** from GitHub and place them in the **`Chapter05\Activities\Activity01`** folder.
3. Create the main console app to prompt user inputs and to generate the images.
4. Add a new class called **`FibonacciConsole.cs`**.
5. Include **`System.IO`** to create a temporary folder and **`System.Globalization`** to parse input strings into numbers:

```
using System;
using System.Drawing.Imaging;
using System.Globalization;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter05.Activities.Activity01
{
    public static class FibonacciConsole
    {
```

6. Add the standard static **`Main`** entry point (you won't need the **`async`** keyword as you are not awaiting at this point):

```
        public static void Main()
        {
```

7. Use **`Path.GetTempPath()`** to find the name of the **`Temp`** folder and add **`Fibonacci`** as a subfolder name:

```
            var tempImagePath = Path.GetTempPath() + "Fibonacci\\";
```

8. If that temporary folder doesn't exist, create the folder (this will work on all systems, not just Windows):

```
            if (!Directory.Exists(tempImagePath))
            {
                Console.WriteLine($"Creating temp folder:
{tempImagePath}");
                Directory.CreateDirectory(tempImagePath);
            }
            else
            {
                Console.WriteLine($"Using temp folder:
{tempImagePath}");
            }
```

9. Use **`CancellationTokenSource`** to signal that the calculation and image export task should stop.
10. Declare it but leave it as null at this stage:

```
            CancellationTokenSource tokenSource = null;
```

11. Start a **`do`**-loop:

```
            do
            {
               Console.Write("Phi (eg 1.0 to 6.0) (x=quit,
enter=cancel):");
               var input = Console.ReadLine();
```

   You want the user to be able to enter a value for **`phi`** as many times as they want to. Prompt for a value between **`1.0`** and **`6.0`**, as these produce interesting patterns, and show the quit and cancel message.

12. If the user enters a blank value, use that as an opportunity to cancel any other pending calculations:

```
                if (string.IsNullOrEmpty(input))
                {
                    Console.WriteLine("Cancelling previous tasks");
                    tokenSource?.Cancel();
                    continue;
                }
```

   This allows the other tasks to complete in time. Notice the null check before calling **`Cancel`**. If this is the first time the loop has run, the token will be null.

13. Check to see if **x** was entered and if so, **break** out of the **do**-loop to end:

```
                if (input == "x")
                {
                    break;
                }
```

14. Use **double.TryParse** to convert the user's input into a double number:

```
                if (!double.TryParse(input, NumberStyles.Any,
CultureInfo.CurrentCulture, out double phi))
                {
                    continue;
                }
```

15. Prompt for the number of images to create, using **int.TryParse** to convert the **Console.ReadLine()** string into an integer, and tell the user you are about to start the process:

```
                Console.Write("Image Count (eg 1000):");
                input = Console.ReadLine();
                if (!int.TryParse(input, NumberStyles.Any,
CultureInfo.CurrentCulture, out int imageCount))
                {
                    continue;
                }
                Console.WriteLine($"Creating {imageCount}
images...");
```

16. You can't reuse a token once it has been cancelled, so create a new token source.
17. Attach an event handler to the **cancel** token for extra information. This token can then be passed to **Task.Run** and to the **GenerateImageSequences** method:

```
                tokenSource = new CancellationTokenSource();
                tokenSource.Token.Register(
                    () => Console.WriteLine("Cancelled!"));
                     var token = tokenSource.Token;
                Task.Run(() => GenerateImageSequences(tempImagePath,
                               phi, imageCount, token),
                          token);
```

```
            }
            while (true);
        }
```

18. Add the **`GenerateImageSequences`** method:

```
    private static async Task GenerateImageSequences(string
tempImagePath, double phi, int imageCount, CancellationToken token)
        {
            const double PhiIncrement = 0.015D;
            const int Points = 3000;
            const int ImageSize = 800;
            const int PointSize = 5;;
            const string FileExtension = ".png";
            var fileFormat = ImageFormat.Png;
```

    The **`GenerateImageSequences`** method does the work of creating multiple sequences and exporting their resulting image. Here, you have defined constants for some of the parameters. It may have been too much to prompt the user to enter these values repeatedly. Note that you need the **`async`** keyword as this contains **awaitable** code.

19. Use a **`for`** loop to iterate through the number of images requested.
20. In each loop, await a call to **`FibonacciSequence.Calculate`** passing in the points per image and the current value of **`phi`**:

```
            for (var i = 0; i < imageCount; i++)
            {
                phi += PhiIncrement;
                var sequence = await Task.Run(
                      () => FibonacciSequence.Calculate(Points, phi),
                      token);
```

    You are returned a list of **`Fibonacci`** items. You saw earlier that loop-based code can check a token's **`IsCancellationRequested`** status.

21. If that is the case, use the **`break`** statement to stop any more image files from being created. The user may have requested a cancellation:

```
                if (token.IsCancellationRequested)
                {
                    break;
                }
```

22. Now add another awaitable block using **`Task.Run`**:

```
                var imagePath =
               $"{tempImagePath}Fibonacci_{Points}_{phi:N3}
{FileExtension}";

                await Task.Run(
                   () => ImageGenerator.ExportSequence(sequence,
imagePath,
                            fileFormat, ImageSize, ImageSize,
PointSize),
                    token);
            }
        }
    }
}
```

This will call **`ImageGenerator.ExportSequence`**, passing in a unique filename based on the number of points, the current value of **`phi`**, and required image details.

23. Run the console app to produce this output:

```
Using temp folder: C:Temp\Fibonacci\
Phi (eg 1.0 to 6.0) (x=quit, enter=cancel):1
Image Count (eg 1000):1000
Creating 1000 images...
Phi (eg 1.0 to 6.0) (x=quit, enter=cancel):
20:36:19 [04] Saved Fibonacci_3000_1.015.png
20:36:19 [06] Saved Fibonacci_3000_1.030.png
20:36:19 [05] Saved Fibonacci_3000_1.045.png
20:36:20 [08] Saved Fibonacci_3000_1.060.png
20:36:20 [08] Saved Fibonacci_3000_1.075.png
20:36:20 [06] Saved Fibonacci_3000_1.090.png
20:36:20 [05] Saved Fibonacci_3000_1.105.png
20:36:20 [05] Saved Fibonacci_3000_1.120.png
20:36:20 [05] Saved Fibonacci_3000_1.135.png
20:36:20 [05] Saved Fibonacci_3000_1.150.png
20:36:20 [07] Saved Fibonacci_3000_1.165.png
20:36:20 [07] Saved Fibonacci_3000_1.180.png
20:36:20 [07] Saved Fibonacci_3000_1.195.png
Cancelling previous tasks
Cancelled!
```

Try entering different values for **`phi`** to see how that affects the style of images created. From the output, you can see that various threads have been used to create the images and many threads are used more than once. By pressing **`Enter`**, the process is stopped:

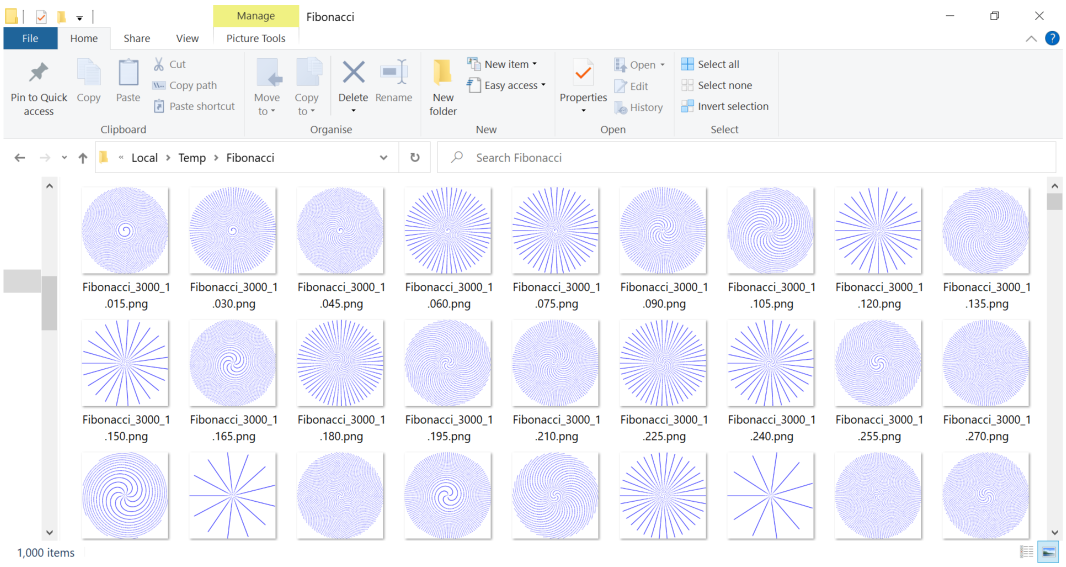


Figure 5.4: Windows 10 Explorer image folder contents (a subset of images produced)

In this activity, you created a console application that allowed various inputs to be passed to a sequence calculator. Once you had entered your parameters, the app will start the time-consuming task of creating 1,000 images. You used the **`FibonacciSequence.cs`** and **`ImageGenerator.cs`** files and created a console app that will generate the .png files.

Chapter 6: Entity Framework with SQL Server

Activity 6.01: Tracking System for Trucks Dispatched

Solution:

1. Create a **`Person`** class:

```
[Table("Person", Schema = "TruckLogistics")]
public class Person
{
    public int Id { get; set; }
    [MaxLength(300)]
    public string Name { get; set; }
    public DateTime DoB { get; set; }
}
```

2. Create a **`Truck`** class:

```
[Table("Truck", Schema ="TruckLogistics")]
public class Truck
{
    public int Id { get; set; }
    [MaxLength(100)]
    public string Brand { get; set; }
    [MaxLength(100)]
    public string Model { get; set; }
    public int YearOfMaking { get; set; }
}
```

3. Create a **`TruckDispatch`** class:

```
[Table("TruckDispatch", Schema = "TruckLogistics")]
public class TruckDispatch
{
    public int Id { get; set; }
    public Truck Truck {get;set;}
    public Person Driver { get; set; }
    public int DriverId { get; set; }
    public int TruckId { get; set; }
    [MaxLength(200)]
```

```
        public string CurrentLocation { get; set; }
        public DateTime Deadline { get; set; }
    }
```

4. Create a **`TruckDispatchesDbContext`** schema with three tables:

   `TruckDispatchDbContext.cs`

```
public class TruckDispatchDbContext : DbContext
{
    public DbSet<Truck> Trucks { get; set; }
    public DbSet<Person> People { get; set; }
    public DbSet<TruckDispatch> TruckDispatches { get; set; }

    public TruckDispatchDbContext()
        : base(UsePostgreSqlServerOptions())
    {
    }

    protected static DbContextOptions UsePostgreSqlServerOptions()
    {
        return new DbContextOptionsBuilder()
            .UseNpgsql(Program.TruckLogisticsConnectionString)
            .Options;
```

   `You can find the complete code here:` https://packt.link/H7RKm.

5. Create a connection string (ideally, from environment variables):

```
public static string TruckLogisticsConnectionString { get;
} = Environment.GetEnvironmentVariable("TruckLogistics",
EnvironmentVariableTarget.User);
```

6. Create the environment variable:

```
Host=localhost;Username=postgres;Password=***;Database=TruckLogistics
```

7. Add a database migration:

```
dotnet ef migrations add InitialMigration -c TruckDispatchDbContext
-o Activities/Migrations
```

8. Generate a database from the migration:

```
dotnet ef database update -c TruckDispatchDbContext
```

9. Connect to a database:

```
var db = new TruckDispatchDbContext();
```

10. Seed the database with the initial data:

Demo.cs

```
private static void SeedData(TruckDispatchDbContext db)
{
    var wasSeeded = db.Trucks.Any();
    if(!wasSeeded)
    {
        var person = new Person { DoB = DateTime.UtcNow, Id = 1, Name = 
"Stephen King" };
        db.People.Add(person);

        var truck = new Truck() { Id = 1, Brand = "Scania", Model = "R 500 
LA6x2HHA", YearOfMaking = 2009 };
        db.Trucks.Add(truck);

        var dispatch = new TruckDispatch()
        {
```

You can find the complete code here: https://packt.link/yAFAh.

11. Get all the data from the database:

```
private static IEnumerable<TruckDispatch> 
GetAll(TruckDispatchDbContext db)
{
    var dispatches = db
        .TruckDispatches
        .Include(td => td.Driver)
        .Include(td => td.Truck)
        .ToList();

    return dispatches;
}
```

12. Print the results:

```
private static void Print(IEnumerable<TruckDispatch> truckDispatches)
{
    foreach(var dispatch in truckDispatches)
    {
        Console.WriteLine($"Dispatch: {dispatch.Id} {dispatch.
CurrentLocation} {dispatch.Deadline}");
        Console.WriteLine($"Driver: {dispatch.Driver.Name} {dispatch.
Driver.DoB}");
        Console.WriteLine($"Truck: {dispatch.Truck.Brand} {dispatch.
Truck.Model} {dispatch.Truck.YearOfMaking}");
    }
}
```

13. Dispose of the **`TruckDispatchesDbContext`** schema (that is, disconnect):

```
db.Dispose();
```

14. Execute the **`Demo`** method to run the activity and print the results:

```
public static class Demo
{
    public static void Run()
    {
        var db = new TruckDispatchDbContext();
        SeedData(db);
        var dispatches = GetAll(db);
        Print(dispatches);
        db.Dispose();
    }
    //..
}
```

15. The following output gets displayed:

```
Dispatch: 1 1,1,1 2021-11-02 21:45:42
Driver: Stephen King 2021-07-25 21:45:42
Truck: Scania R 500 LA6x2HHA 2009
```

In this activity, you created a database for dispatched trucks, seeded it with a few dispatches, and proved its performance by getting all possible data from it.

Chapter 7: Creating Modern Web Applications with ASP.NET

Activity 7.01: Creating a Page to Edit an Existing Task

Solution:

1. Create a new file called **`Edit.cshtml`** with the same form as **`Create.cshtml`**:

   `Edit.cshtml`

```
@model EditModel
@{
    ViewData["Title"] = "Task";
}

<h2>Edit</h2>
<div>
    <h4>@ViewData["Title"]</h4>
    <hr />
    <dl class=»row»>
        <form method=»post» class=»col-6»>
            <div class=»form-group»>
                <label asp-for=»Task.Title»></label>
                <input asp-for=»Task.Title» class=»form-control» />
                <span asp-validation-for=»Task.Title» class=»text-danger»></
span>
```

   `You can find the complete code here:` https://packt.link/0fvE7.

2. Change the route at the page directive to receive **`"/tasks/{id}"`**:

```
@page "/tasks/{id}"

@model EditModel
@{
    ViewData["Title"] = "Task";
}

…

</div>
```

3. Create the code-behind file that loads a task by the **OnGet** ID from the database schema using the **ToDoDbContext** (**Pages/Tasks/Edit.cshtml.cs**):

   Edit.cshtml.cs

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using ToDoListApp.Data;
using ToDoListApp.Models;

namespace ToDoListApp.Pages.Tasks;
{
    public class EditModel : PageModel
    {
        private readonly ToDoDbContext _context;

        public EditModel(ToDoDbContext context)
        {
            _context = context;
```

   You can find the complete code here: https://packt.link/SlvSr.

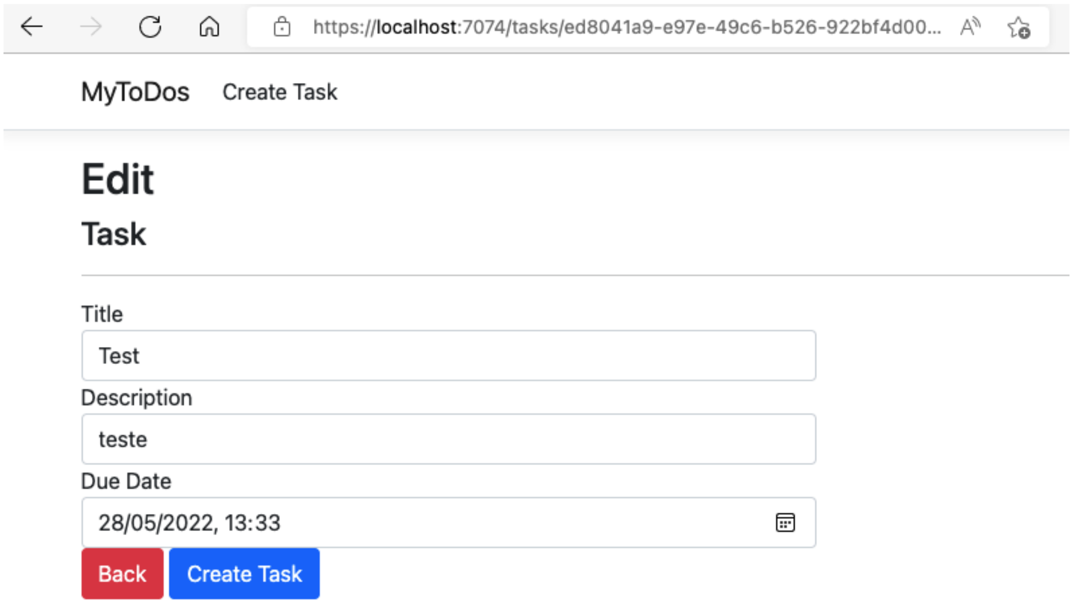
   If the ID does not return a task, redirect it to the **Create** page.

4. To display the contents from the Model on the Post form, add the following code to recover the task from the database, update its values, send a success message, and redirect to the Index view afterward (**Pages/Tasks/Edit.cshtml.cs**):

```
public async Task<IActionResult> OnPostAsync(Guid id)
         {
             var task = _context.Find<Task>(id);

             if (await TryUpdateModelAsync(task, "Task"))
             {
                 _context.Update(task);

                 await _context.SaveChangesAsync();

                 TempData["SuccessMessage"] = $"Task successfully
updated";
             }

             return RedirectToPage("Index");
         }
```

   Here views help to display the contents from the Model.

The output of a page is displayed as follows:


https://localhost:7074/tasks/ed8041a9-e97e-49c6-b526-922bf4d00...

MyToDos Create Task

Edit

Task

Title

Test

Description

teste

Due Date

28/05/2022, 13:33

Back Create Task

Figure 7.11: The Edit Task Page as output to the activity

ACTIVITY 7.02: WRITING A VIEW COMPONENT TO DISPLAY TASK LOG

Solution:

1. Create a new class under the **`Models`** folder named **`ActivityLog`**:

`ActivityLog.cs`

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ToDoListApp.Models
{
    public class ActivityLog
    {
        public ActivityLog()
        {
            Id = Guid.NewGuid();
        }

        [Key]
        public Guid Id { get; set; }
        public string EntityId { get; set; }
```

`You can find the complete code here:` https://packt.link/NbiKq.

This class should have the following properties: **Guid Id**, **String EntityId**, **DateTime Timestamp**, **String Property**, **String OldValue** and **String NewValue**. The preceding code creates the properties.

2. Create a new **DbSet<ActivityLog>** property for this model under **ToDoDbContext** (placed under **Data/ToDoDbContext.cs/**):

ToDoDbContext.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using ToDoListApp.Models;

namespace ToDoListApp.Data
{
    public class ToDoDbContext : DbContext
    {
        public ToDoDbContext(DbContextOptions<ToDoDbContext> options) :
base(options)
        {
```

You can find the complete code here: https://packt.link/kGNs6.

3. Under the **ToDoDbContext**, create a method to generate activity logs for the modified properties of **Entries** under the Entity Framework's **ChangeTracker** with **EntityState.Modified**:

```
private IEnumerable<ActivityLog> GenerateActivityLog()
        {
            var changes = ChangeTracker
                .Entries()
                .Where(e => e.State == EntityState.Modified)
                .ToList();

            foreach (var entity in changes)
            {
                var changedProperties = entity
                    .Properties
```

```
                        .Where(p => p.IsModified && !p.CurrentValue.
Equals(p.OriginalValue));

                    foreach (var property in changedProperties)
                    {
```

4. Override **SaveChangesAsync()** in **DbContext**, adding the generated logs to **DbSet** right before calling the **base** method:

```
public override Tasks.Task<int> SaveChangesAsync(CancellationToken
cancellationToken = default)
        {
            var logs = GenerateActivityLog();

            Activities.AddRange(logs);

            return base.SaveChangesAsync(cancellationToken);
        }
```

5. Create a new Entity Framework Core migration and update the database to support this migration **under Migrations/20220508121354_FirstMigration.cs**:

20220508125713_ActivityLog.cs

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace ToDoListApp.Migrations
{
    public partial class ActivityLog : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AlterColumn<DateTime>(
                name: "DueTo",
                table: "Tasks",
                type: "TEXT",
```

You can find the complete code here: https://packt.link/LlUHq.

6. Create the **ViewComponent** class (under **ViewComponents/ActivityLogViewComponent.cs**), which should load all logs for a given **taskId** passed on the invocation and return them to the **ViewComponent**:

ActivityLogViewComponent.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;
using ToDoListApp.Data;

namespace ToDoListApp.ViewComponents
{
    public class ActivityLogViewComponent : ViewComponent
    {
        private readonly ToDoDbContext _context;
        private readonly ILogger<ActivityLogViewComponent> _logger;

        public ActivityLogViewComponent(ToDoDbContext context,
```

You can find the complete code here: https://packt.link/7ieDH.

NOTE

In the preceding code snippet, you can also use **_logger.LogWarning** instead of **LogInformation**.

7. Create the **ViewComponent** view, which should take a collection of **ActivityLog** (under **Pages/Components/ActivityLog**) as a model and display them in a Bootstrap table, if any exists:

default.cshtml

```
@model System.Collections.Generic.IEnumerable<ToDoListApp.Models.ActivityLog>

@if (Model.Count() > 0)
{
  <table class="table">
    <thead>
        <tr>
            <th scope="col">#</th>
            <th scope="col">Timestamp</th>
            <th scope="col">Property</th>
            <th scope="col">Old Value</th>
            <th scope="col">New Value</th>
        </tr>
    </thead>
    <tbody>
```

You can find the complete code here: https://packt.link/E6QbP.

If no logs were recorded, show an alert saying that no log is available.

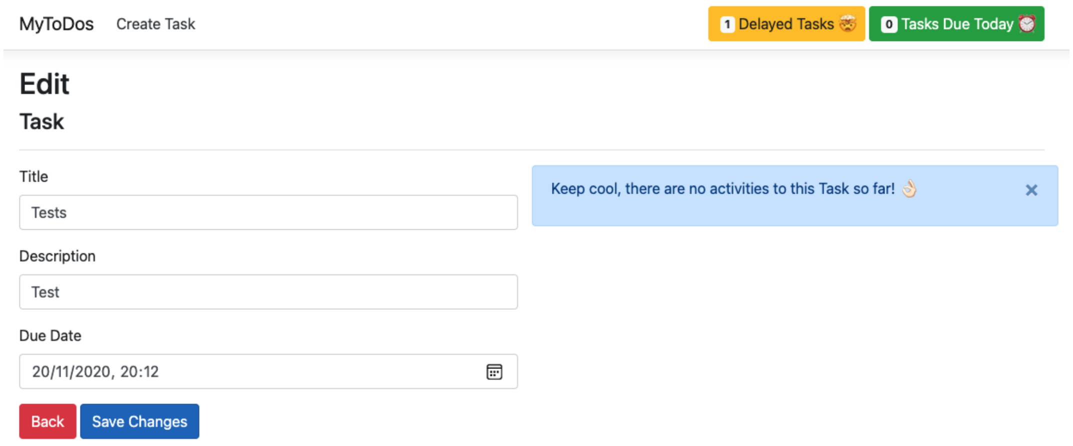
8. Add the view component to the **Edit** page (under **Pages/Tasks**), passing the **taskId** property:

Edit.cshtml

```
@page "/tasks/{id}"
@model EditModel
@{
    ViewData["Title"] = "Task";
}

<h2>Edit</h2>
<div>
    <h4>@ViewData["Title"]</h4>
    <hr />
    <dl class="row">
        <form method="post" class="col-6">
            <div class="form-group">
                <label asp-for="Task.Title"></label>
                <input asp-for="Task.Title" class="form-control" />
```

You can find the complete code here: https://packt.link/WLNXq.

9. Run the application and check the final output by opening a task's details:



Edit

Task

Title: Tests

Description: Test

Due Date: 20/11/2020, 20:12

Keep cool, there are no activities to this Task so far! 👌

Figure 7.12: The Activity log being displayed with no logs

You will see a box on the right with your activity logs. If there were no activity logs recorded for that task yet, a message with no logs will be displayed.

This activity was based on a common task in real-world applications—to have a log of user activities. You wrote every change the user did to a field in the database and displayed it in a view.

Chapter 8: Creating and Using Web API Clients

Activity 8.01: Reusing HttpClient for the Rapid Creation of API Clients

Solution:

1. Create a base client, **`BaseHttpClient`**:

```
public class BaseHttpClient : IDisposable
{
    private readonly HttpClient _client;

    public BaseHttpClient(string url)
    {
        _client = new HttpClient { BaseAddress = new Uri(url) };
    }
```

2. Implement a method to create **`HttpRequestMessage`** out of a URL:

```
    protected HttpRequestMessage CreateGetRequest(string url)
    {
        return new HttpRequestMessage(HttpMethod.Get, new Uri(url, 
UriKind.Relative));
    }
```

3. Implement a method to send **`HttpRequestMessage`** by adding error handling and deserializing:

```
    protected async Task<T> SendRequest<T>(HttpRequestMessage 
request)
    {
        var response = await _client.SendAsync(request);

        if (!response.IsSuccessStatusCode)
        {
            throw new HttpRequestException(response.ReasonPhrase);
        }

        var content = await response.Content.ReadAsStringAsync();
        var apiResult = JsonConvert.DeserializeObject<T>(content);

        return apiResult;
    }
```

4. Create a method to also simplify sending a request to get multiple items:

```
    protected Task<ApiResult<IEnumerable<T>>>
SendGetManyRequest<T>(HttpRequestMessage request)
        => SendRequest<ApiResult<IEnumerable<T>>>(request);
}
```

5. Implement the **Dispose** method:

```
public void Dispose()
        {
            _client.Dispose();
        }
```

Using the lambda operator, you can also write it as follows:

```
public void Dispose() => _client.Dispose();
```

6. Use **BaseHttpClient** to simplify **StarWarsClient**:

```
public class StarWarsClient : BaseHttpClient
{
    public StarWarsClient():base("https://swapi.dev/api/")
    {
    }

    public async Task<ApiResult<IEnumerable<Film>>> GetFilms()
    {
        var request = CreateGetRequest("films");
        var films = await SendGetManyRequest<Film>(request);

        return films;
    }
}
```

7. Create a class to demo the new client getting all the Star Wars films:

```
public static class Demo
{
    public static async Task Run()
    {
        using var client = new StarWarsClient();
        var filmsResponse = await client.GetFilms();
        var films = filmsResponse.Data;
        foreach (var film in films)
```

```
            {
                    Console.WriteLine($"{film.ReleaseDate} {film.Title}");
            }
        }
    }
```

8. If you run the demo once again with the new **`StarWarsClient`**, you should see the same films returned:

```
1977-05-25 A New Hope
1980-05-17 The Empire Strikes Back
1983-05-25 Return of the Jedi
1999-05-19 The Phantom Menace
2002-05-16 Attack of the Clones
2005-05-19 Revenge of the Sith
```

In this activity, you learnt how to implement a base client by rewriting **`StarWarsClient`** using **`BaseHttpClient`**. You used the **`BaseHttpClient`** class to generalize error handling and deserializing responses and requests. This significantly simplified different HTTP calls that you made.

ACTIVITY 8.02: THE COUNTRIES API USING RESTSHARP TO LIST ALL COUNTRIES

Solution:

1. Create a base client for countries and initialize **`RestClient`** as you initialize **`CountriesClient`**:

```
public class CountriesClient
    {
        private readonly RestClient _client;

        public CountriesClient()
        {
            _client = new RestClient("https://restcountries.com/
v3/");
        }
```

2. Create methods for getting countries in different ways:

```
        public Task<IEnumerable<Country>> Get() => GetCountries("/
all");

        public Task<IEnumerable<Country>> GetByLanguage(string
language) => GetCountries($"/lang/{language}");

        public Task<IEnumerable<Country>> GetByCapital(string
capital) => GetCountries($"capital/{capital}");

        private async Task<IEnumerable<Country>> GetCountries(string
endpoint)
        {
            var request = new RestRequest(endpoint);
            var result = await _client.
GetAsync<IEnumerable<Country>>(request);

            return result;
        }
    }
```

3. Use a tool such as https://json2csharp.com/ to generate the models from any example response:

CountriesClient.cs

```
    public class CommonNames
    {
        public NativeName aym { get; set; }
        public NativeName que { get; set; }
        public NativeName spa { get; set; }
    }

    public class Name
    {
        public string common { get; set; }
        public string official { get; set; }
        public CommonNames nativeName { get; set; }
    }

    public class PEN
```

You can find the complete code here: https://packt.link/kXXzT.

4. Create a demo running all three methods and print the countries within each response:

Demo.cs

```
public static class Demo
{
    public static async Task Run()
    {
        var client = new CountriesClient();
        IEnumerable<Country> countries;

        Console.WriteLine("All:");
        countries = await client.Get();
        Print(countries);

        Console.WriteLine($"{Environment.NewLine}Lithuanian:");
        countries = await client.GetByLanguage("Lithuanian");
        Print(countries);
```

You can find the complete code here: https://packt.link/QBML8.

5. Run **Demo.Run()** and you will see the following:

```
All:
Aruba Americas Oranjestad
Afghanistan Asia Kabul

Lithuanian:
Lithuania Europe Vilnius

Vilnius:
Lithuania Europe Vilnius
```

In this activity, using the Web API address https://restcountries.com/v3/, you displayed a list of all countries and could access a country by its capital city and language. You printed only the first two country names, their regions, and their capitals, and implemented a strongly typed client to access this API using RestSharp. The aim of this activity was to make you feel more comfortable using third-party libraries (RestSharp), as they often save a lot of time.

ACTIVITY 8.03: THE COUNTRIES API USING REFIT TO LIST ALL COUNTRIES

Solution:

1. Create an interface to which **`Refit`** will bind:

```
    public interface ICountriesClient
    {
        [Get("/all")]
        public Task<IEnumerable<Country>> Get();

        [Get("/lang/{language}")]
        public Task<IEnumerable<Country>> GetByLanguage(string 
language);

        [Get("/capital/{capital}")]
        public Task<IEnumerable<Country>> GetByCapital(string 
capital);
    }
```

2. Create a demo running all three methods and printing the countries within each response:

`Demo.cs`

```
public static class Demo
{
    public static async Task Run()
    {
        var client = RestService.For<ICountriesClient>("https://restcountries.
com/v3/");
        IEnumerable<Country> countries;

        Console.WriteLine("All:");
        countries = await client.Get();
        Print(countries);

        Console.WriteLine($"{Environment.NewLine}Lithuanian:");
        countries = await client.GetByLanguage("Lithuanian");
        Print(countries);

        Console.WriteLine($"{Environment.NewLine}Vilnius:");
```

`You can find the complete code here:` https://packt.link/NX1Wa.

The following result gets displayed:

```
All:
Aruba Americas Oranjestad
Afghanistan Asia Kabul

Lithuanian:
Lithuania Europe Vilnius

Vilnius:
Lithuania Europe Vilnius
```

In this activity, you accessed the Countries API and displayed all countries—countries by their language, and their capital city. The aim of this activity was to show how practical Refit can be for rapid prototyping when it comes to consuming simple APIs.

Activity 8.04: Using an Azure Blob Storage Client to Upload and Download Files

Solution:

1. Navigate to **`Azure Storage Accounts`**:

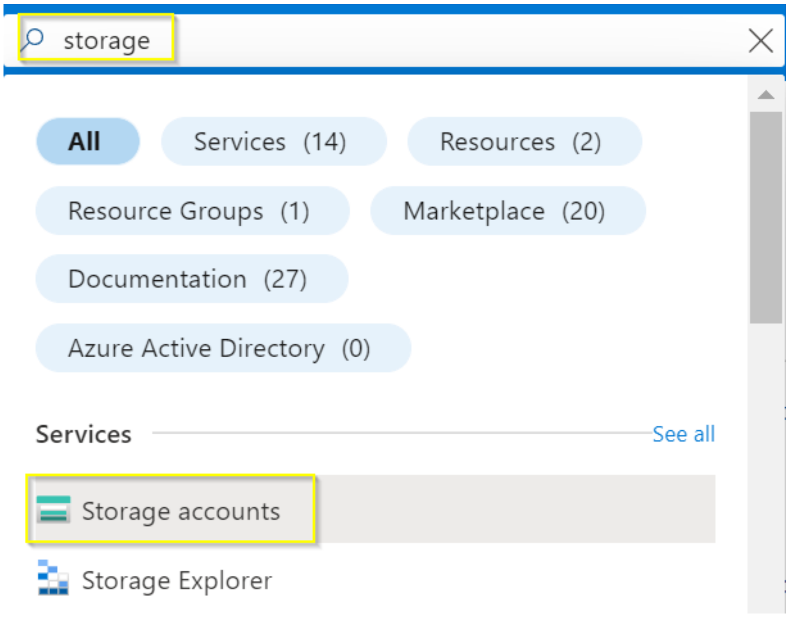
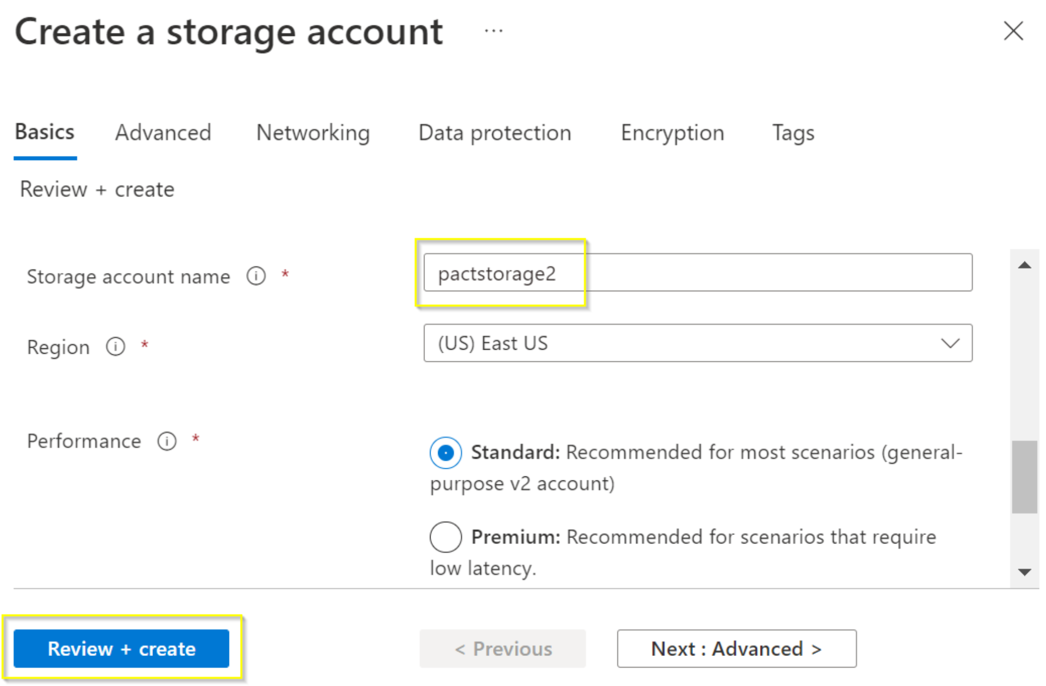


Figure 8.30: Navigating to the Storage accounts window in Azure

2. Create a new Azure Storage Account. The minimum information required here is the account name. In this case, it is **`packtstorage2`**.
3. Next click the **`Review + create`** button:



Storage account name ⓘ * pactstorage2

Region ⓘ * (US) East US

Performance ⓘ *

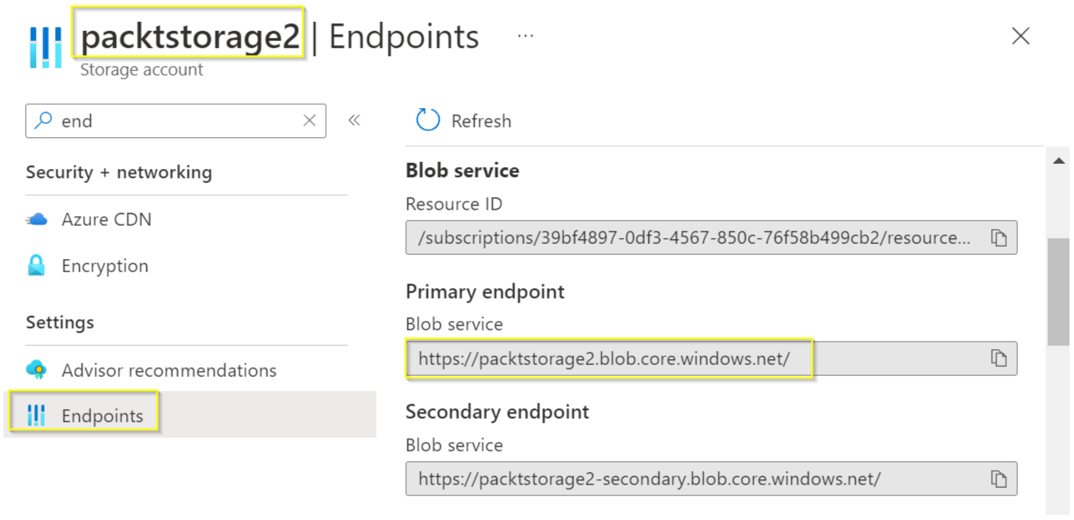
● **Standard:** Recommended for most scenarios (general-purpose v2 account)

○ **Premium:** Recommended for scenarios that require low latency.

Figure 8.31: Creating a new storage account

4. Open the new storage account. You can do this by clicking on a notification of a new storage account creation or by typing **`Storage accounts`** in the search bar of Azure and opening **`packtstorage2`**.
5. Navigate to the **`Settings`** section and open **`Endpoints`**.

6. Copy the **`Blob service`** endpoint to keep it for later use:



packtstorage2 | Endpoints

Storage account

Blob service

Resource ID

/subscriptions/39bf4897-0df3-4567-850c-76f58b499cb2/resource...

Primary endpoint

Blob service

https://packtstorage2.blob.core.windows.net/

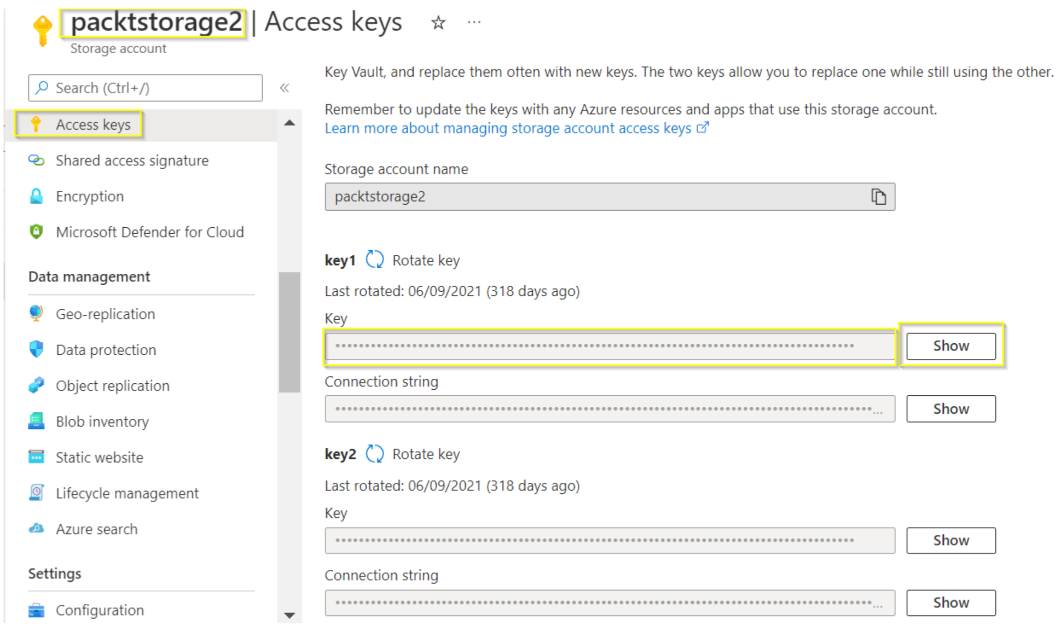
Secondary endpoint

Blob service

https://packtstorage2-secondary.blob.core.windows.net/

Figure 8.32: The packtstorage2 account endpoints, specifically for the Blob service

7. Scroll down the left pane until you see the **`Security + networking`** section and open **`Access keys`**:



packtstorage2 | Access keys

Storage account

Key Vault, and replace them often with new keys. The two keys allow you to replace one while still using the other.

Remember to update the keys with any Azure resources and apps that use this storage account.
Learn more about managing storage account access keys

Storage account name: packtstorage2

key1 Rotate key

Last rotated: 06/09/2021 (318 days ago)

Key: •• Show

Connection string: •••... Show

key2 Rotate key

Last rotated: 06/09/2021 (318 days ago)

Key: •• Show

Connection string: •••... Show

Figure 8.33: Revealing API access key of blob storage

8. Click **Show** under the **key1** key.
9. Store the first key, **key1**, in your environment variables.

> **NOTE**
>
> There is nothing wrong with using **key2** either. However, keep in mind that the two keys are used for rotation purposes. Therefore, if you use **key1** and **key2** for different applications, one of them will lose access (due to a key being rotated out).

10. Name the environment variable **BlobStorageKey**.
11. Install the Azure Blob storage client with the following command:

```
dotnet add package Azure.Storage.Blobs
```

12. Create the **FilesClient** class for storing fields for blobs client and default container client.
13. Azure has a few levels of clients for working with blobs:
    - First, you need a service client, which will be used to create container clients.
    - Then, use a container client to create blob clients.
    - And finally, a blob client will be used to upload or download blobs.
14. For this scenario, create a service client, **BlobServiceClient**, that is ready to be used and a default container (directory), **BlobContainerClient**, where you will store all the blobs by default:

```
public class FilesClient
{
    private readonly BlobServiceClient _blobServiceClient;
    private readonly BlobContainerClient _defaultContainerClient;
```

15. Create a constructor to initialize the two clients:

```
    public FilesClient(string defaultContainer)
    {
        var endpoint = "https://packtstorage2.blob.core.windows.
net/";
        var account = "packtstorage2";
        var key = Environment.GetEnvironmentVariable("BlobStorageKey",
EnvironmentVariableTarget.User);
```

```
        var storageEndpoint = new Uri(endpoint);
        var storageCredentials = new 
StorageSharedKeyCredential(account, key);
        _blobServiceClient = new BlobServiceClient(storageEndpoint, 
storageCredentials);
        _defaultContainerClient = 
CreateContainerIfNotExists(defaultContainer).Result;
    }
```

`StorageSharedKeyCredential` is a container for storing the account name and an access key. It is then passed to create a service client as shown in the next step.

16. Using the service client, get a default container. Create a **`CreateContainerIfNotExists`** method to create a container or get an existing one (if one already exists):

```
    private async Task<BlobContainerClient> 
CreateContainerIfNotExists(string container)
    {
        var lowerCaseContainer = container.ToLower();
        var containerClient = _blobServiceClient.
GetBlobContainerClient(lowerCaseContainer);
        if (!await containerClient.ExistsAsync())
        {
            containerClient = await _blobServiceClient.
CreateBlobContainerAsync(lowerCaseContainer);
        }


        return containerClient;
    }
```

The container, just like the storage account, must contain names in lowercase. That is why the first thing you do in this method is to convert the name to lowercase. **`GetBlobContainerClient()`** returns a container regardless of whether one exists (it does not ever return **`null`**). An explicit call on a container, **`Exists()`**, is needed. Here, you either get an existing container or create one if no container exists.

17. Create a method to upload a file. Call it **`UploadFileInternal`** since you plan to create two other methods—one to upload to a specific container and another to upload a file to a default container:

```
private Task UploadFileInternal(string file, BlobContainerClient 
client)
{
    var data = new BinaryData(File.ReadAllBytes(file));
```

```
        return client.UploadBlob(Path.GetFileName(file), data);
    }
```

UploadBlob expects the **BinaryData** object to pass the bytes of a file you want to upload when initializing **BinaryData**, and then pass that to the **UploadBlob** method. When uploading from a full path to a file, take only its name to store it in an Azure Blob Container. You do not need any information about the directory.

18. Create a method to upload a file to a specific container:

```
        public async Task UploadFile(string file, string container)
        {
            var containerClient = await 
    CreateContainerIfNotExists(container);
            await UploadFileInternal(file, containerClient);
        }
```

19. Create an **UploadFile** method to upload a file to a default container:

```
        public Task UploadFile(string file)
        {
            return UploadFileInternal(file, _defaultContainerClient);
        }
```

This method is useful for those who just want to upload files to Azure, without worrying about the container (directory) they end up in.

20. Create the **DownloadFileInternal** method for reusability. This is similar to the **UploadFile** method:

FilesClient.cs

```
    private Task DownloadFileInternal(string filename, string 
downloadDirectory, BlobContainerClient client)
    {
        var blobClient = client.GetBlobClient(filename);
        var downloadedFile = Path.Combine(downloadDirectory, filename);

        if (!Directory.Exists(downloadDirectory))
        {
            Directory.CreateDirectory(downloadDirectory);
        }

        if (!File.Exists(downloadedFile))
        {
            var stream = File.Create(downloadedFile);
            stream.Dispose();
        }
```

You can find the complete code here: https://packt.link/q9yt9.

Individual blobs operate on their clients. That is why, in order to download a blob, you first need to get its client using the **`GetBlobClient()`** method. For that, you need to create a directory to which a file could be downloaded (if no directory exists).

21. Create a **`DownloadFile`** method to download a file from a specific container:

```
    public async Task DownloadFile(string filename, string container,
string downloadDirectory)
    {
        var containerClient = await
CreateContainerIfNotExists(container);
        await DownloadFileInternal(filename, downloadDirectory,
containerClient);
    }
```

22. Create a **`DownloadFile`** method to download a file from a default container:

```
    public Task DownloadFile(string filename, string
downloadDirectory)
    {
        return DownloadFileInternal(filename, downloadDirectory, _
defaultContainerClient);
    }
```

This method is useful for those who use a default container to upload files. In other words, those users who don't mind where a file in Azure is stored can use this as well to download such a file in the same, simplified way.

23. Create a **`Demo`** class with paths to download and upload directories:

```
public static class Demo
{
    public const string Downloads = "Activities/Activity04/Data/
Downloads";
    public const string Uploads = "Activities/Activity04/Data/
Uploads";
```

NOTE

For this solution, **`Downloads`** and **`Uploads`** constants could be made **`private`**, but for testing purposes (you reference them in tests) it has been made public.

24. Create a **`Run`** method to upload a text file and then download it locally:

```
            public static async Task Run()
            {
                var client = new FilesClient("Activity04");
                var filename1 = "Test1.txt";
                await client.DeleteFile(filename1);
                var fullpath1 = Path.Combine(Uploads, filename1);
                await client.UploadFile(fullpath1);
                await client.DownloadFile(filename1, Downloads);
```

    You used **`Path.Combine()`** instead of concatenating two strings because it solves the problem of having too many slashes or a missing slash at the end. It is ideal for concatenating a filename with a directory.

25. Add code to upload an image and then download it locally to the **`Run`** method:

```
                var filename2 = "Morning.jpg";
                var container = "Activity04B";
                var fullpath2 = Path.Combine(Uploads, filename2);
                await client.DeleteFile(filename2, container);
                await client.UploadFile(fullpath2, container);
                await client.DownloadFile(filename2, container,
    Downloads);
            }
```

26. Now add test data, namely the two files provided under **`Activities/Activity04/Data/Uploads`**.

27. To copy the files to the **`/bin`** directory on build, add the following code anywhere within the **`<Project>`** tags to the **`Chapter08.csproj`** file:

```
    <ItemGroup>
      <None Update="Activities\Activity04\Data\Uploads\Morning.jpg">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      </None>
      <None Update="Activities\Activity04\Data\Uploads\Test1.txt">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      </None>
    </ItemGroup>
```

28. Now, run the code to get the output shown in *Figure 8.34*:

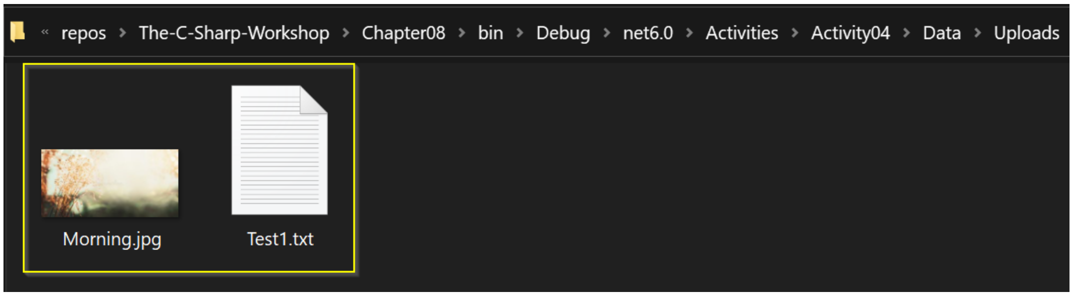

« repos › The-C-Sharp-Workshop › Chapter08 › bin › Debug › net6.0 › Activities › Activity04 › Data › Uploads
Morning.jpg
Test1.txt

Figure 8.34: packtstorage2 access keys for calling the Blob service

The aim of this activity was to familiarize you with working on files through cloud storage. You learned about a cloud service on Azure for storing different files—logs, images, music, and whole drives—called Azure Blob Storage. You also gained knowledge about Azure Storage Container, which is like a directory where other files are stored.

Chapter 9: Creating API Services

Activity 9.01: Implementing the File Upload Service Using Microservice Architecture

Solution:

1. Create a **.csproj** project file for **Microsoft.NET.Sdk.Web** using the .NET 6.0 framework with the following packages:
   - **Hellang.Middleware.ProblemDetails**
   - **Microsoft.Identity.Web**
   - **Swashbuckle.AspNetCore**
   - **Azure.Storage.Blobs**
2. To make Swagger generate documentation, add the **GenerateDocumentationFile** flag, setting it to **true**:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
  <PropertyGroup>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
    <NoWarn>$(NoWarn);1591</NoWarn>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Hellang.Middleware.ProblemDetails" 
Version="6.2.0" />
    <PackageReference Include="Microsoft.Identity.Web" 
Version="1.17.0" />
    <PackageReference Include="Swashbuckle.AspNetCore" 
Version="6.2.1" />
    <PackageReference Include="Azure.Storage.Blobs" Version="12.9.1" 
/>
  </ItemGroup>
</Project>
```

3. Copy the https://packt.link/cTa4a files to the new project. This is the core logic.

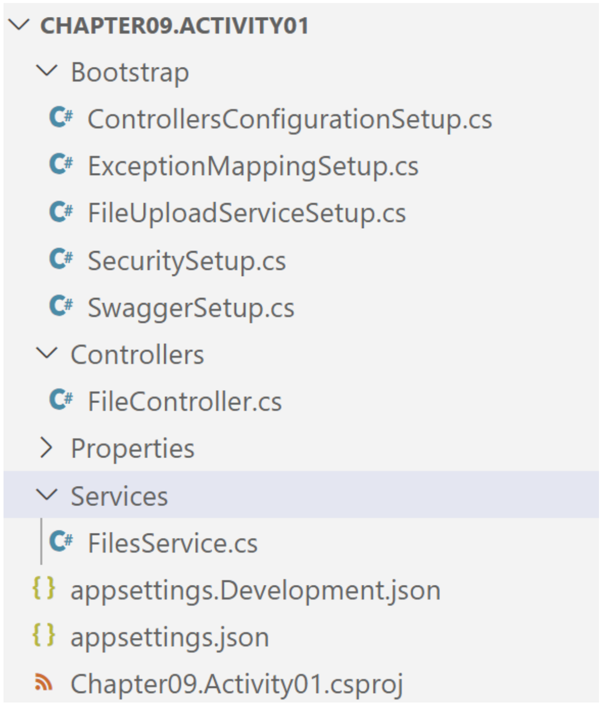
4. Copy the following files from https://packt.link/iQK5X to the new project:
    - **`ControllersConfigurationSetup.cs`**
    - **`ExceptionMappingSetup.cs`**
    - **`FileUploadServiceSetup.cs`**
    - **`SecuritySetup.cs`**
    - **`SwaggerSetup.cs`**
5. Wire all the modules in the **`Program`** class:

`Program.cs`

```
var builder = WebApplication.CreateBuilder(args);

var services = builder.Services;
var configuration = builder.Configuration;
var environment = builder.Environment;

services
    .AddControllersConfiguration()
    .AddRequestValidators()
    .AddSwagger()
    .AddWeatherService(configuration)
    .AddExceptionMappings(environment)
    .AddHttpClients(configuration)
    .AddModelMappings()
    .AddFileUploadService()
    .AddSecurity(configuration, environment);

var app = builder.Build();
```

`You can find the complete code here:` https://packt.link/oK1B1.

6. If you did everything correctly, the service in Solution Explorer should look like this:



```
CHAPTER09.ACTIVITY01
  Bootstrap
    ControllersConfigurationSetup.cs
    ExceptionMappingSetup.cs
    FileUploadServiceSetup.cs
    SecuritySetup.cs
    SwaggerSetup.cs
  Controllers
    FileController.cs
  Properties
  Services
    FilesService.cs
  appsettings.Development.json
  appsettings.json
  Chapter09.Activity01.csproj
```

Figure 9.56: Chapter09.Activity01 project layout through Solution Explorer

When you start the service, you should be able to access all the functionality from *Exercise 9.03*, except the functionality that is within its own microservice.

In this activity, you extracted a piece of code into a microservice that manages files through the web (delete, upload, and download). This served as an overall effective checklist of creating a new microservice.

Chapter 10: Automated Testing

Activity 10.01: Creating a Bank Teller

Solution:

1. In **Chapter10.Lib** project, add a new file called **Activity01.cs** with the respective namespaces—**System.Collections.Generic** for generic **List** and **System.Linq** for LINQ's **Sum()** method—to calculate an account's balance:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Chapter10.Lib
{
```

2. Define a class named **BankAccount**:

```
    public class BankAccount
    {
        public BankAccount(double openingBalance)
            => OpeningBalance = openingBalance;

        public double OpeningBalance { get; init;}
```

   The constructor is passed an **openingBalance**, which is assigned to an **init-only** property, as you do not expect the opening balance to change.

3. Add a **Transactions** property to track transactions on the account and a **Balance()** method that sums the transactions and the opening balance together:

```
        public List<double> Transactions { get; } = new 
List<double>();

        public double Balance()
            => OpeningBalance + Transactions.Sum();
    }
```

4. Add a **BankTeller** class with a **Transfer** method that accepts **from** and **to** **BankAccount** instances and an **amount** to transfer:

```
public class BankTeller
    {
        public static void Transfer(BankAccount from, BankAccount to,
                                    double amount)
```

   The **BankTeller** class will perform the **Transfer** actions and update the two **BankAccount** instances.

5. Here, you compare the **amount** value and throw an **ApplicationException** if the amount is less than or equal to zero (rule number one stated that if the amount to transfer is less than or equal to zero, then an **Exception** should be thrown):

```
        {
            if (amount <= 0D )
                throw new ApplicationException("Cannot transfer
negative
                                                amounts");
```

6. Compare the **from.Balance** with the amount passed, and throw an **ApplicationException** if required (rule number two stated that if the **from** account's balance is less than the amount requested to transfer out, then an **Exception** should be thrown):

```
            if (from.Balance() < amount)
                throw new ApplicationException("Insufficient funds");
```

7. Add a negative-signed amount to the **from** account's **Transactions**, and add the opposite amount to the **to** account as follows:

```
            from.Transactions.Add(amount * - 1D);
            to.Transactions.Add(amount);
        }
    }
}
```

8. In **Chapter10.Tests**, add a new **Activity01Tests.cs** file.

9. Here, you can add tests that verify the key requirements of the BankTeller's **`Transfer`** method. You will need the **`NUnit.Framework`** and **`Chapter10.Lib`** namespaces:

```
using NUnit.Framework;
using Chapter10.Lib;
using System;
using System.Linq;

namespace Chapter10.Tests
{
    [TestFixture]
    public class BankTellerTests
    {
```

10. Start with a test that verifies the **`Transfer`** method with a valid balance. Now it is ready to pass random numbers for the two **`OpeningBalance`** values:

```
        [Test]
        public void Transfer_ValidBalance_AddsToTransactions()
        {
            // ARRANGE
            Random random = new();

            var fromOpeningBalance = random.NextDouble() * 1000D;
            var fromAccount = new BankAccount(fromOpeningBalance);

            var toOpeningBalance = random.NextDouble() * 500D;
            var toAccount = new BankAccount(toOpeningBalance);
```

11. Create a random amount to transfer and call the **`Transfer`** method, as follows:

```
            var transferAmount = random.NextDouble() * 10D;

            // ACT
            BankTeller.Transfer(fromAccount, toAccount,
transferAmount);
```

12. Assert that the **`from`** account is as expected:

```
            // ASSERT
            Assert.That(fromAccount.Transactions.Count,
Is.EqualTo(1));
            Assert.That(fromAccount.Transactions.
Contains(transferAmount *-1D));
            Assert.That(fromAccount.Balance,
Is.EqualTo(fromOpeningBalance - transferAmount));
```

13. Assert that the **`toAaccount`** is as expected:

```
            Assert.That(toAccount.Transactions.Count, Is.EqualTo(1));
            Assert.That(toAccount.Transactions.
Contains(transferAmount));
            Assert.That(toAccount.Balance, Is.EqualTo(toOpeningBalance
+ transferAmount));
        }
```

14. Create the test using **`Assert.Throws`** that confirms that an **`ApplicationException`** is thrown when a negative amount is requested:

```
        [Test]
        public void Transfer_NegativeAmount_ThrowsException()
        {
            Assert.Throws<ApplicationException>( () =>
            {
                var from = new BankAccount(1);
                var to = new BankAccount(2);

                BankTeller.Transfer(from, to, -1D);

                Assert.That(from.Transactions.Any(), Is.False);
                Assert.That(to.Transactions.Any(), Is.False);
            });
        }
```

15. Finally, create a test that confirms that an **`ApplicationException`** is thrown if the requested amount is greater than the **`from`** account's balance. The test method is named **`Transfer_BalanceTooLow_ThrowsException`** as this is a descriptive name.

16. Start by adding an **`Assert.Thows<ApplicationException>`** block:

```
        [Test]
        public void Transfer_BalanceTooLow_ThrowsException()
        {
            Assert.Throws<ApplicationException>( () =>
            {
```

17. The body of **`Assert.Throws`** is passed a delegate to invoke. This creates a **`from`** **`BankAccount`** with a balance of **`1`** and a **`to`** **`BankAccount`** with a balance of **`2`**:

```
                var from = new BankAccount(1);
                var to = new BankAccount(2);
```

18. Now invoke the **`BankTeller.Transfer`** method, passing the **`from`** and **`to`** instances along with a value that is greater than the balance of the **`from`** account:

```
                BankTeller.Transfer(from, to, from.Balance() + 1D);

                Assert.That(from.Transactions.Any(), Is.False);
                Assert.That(to.Transactions.Any(), Is.False);
            });
        }
    }
}
```

When **`Transfer`** is called, you expect an **`ApplicationException`** to be thrown. As an extra safety check, the **`to.Transactions`** and **`from.Transactions`** are checked to confirm whether they contain any elements by using the **`Any`** method.

19. You can run the unit tests using the **`dotnet test`** command or the **`Run Tests in Context`** option from VS Code to see the following output:

```
----- Test Execution Summary -----
Chapter10.Tests.BankTellerTests.Transfer_BalanceTooLow_
ThrowsException:
    Outcome: Passed
Chapter10.Tests.BankTellerTests.Transfer_NegativeAmount_
ThrowsException:
    Outcome: Passed
Chapter10.Tests.BankTellerTests.Transfer_ValidBalance_
AddsToTransactions:
    Outcome: Passed
Total tests: 3. Passed: 3. Failed: 0. Skipped: 0
```

In this activity, you created a BankTeller class that allowed money to be transferred from one account to another.

Chapter 11: Production-Ready C#: from Development to Deployment

Activity 11.01: CD Workflow with GitHub Actions and Azure

Solution:

1. Inside the **.github/workflows** directory, create a file named **cd.yaml**.
2. Here, copy the **ci.yaml** file as a base, as you need to change only a few parts of it.
3. Pass **cd** as the name of the file, instead of **ci**:

```
name: cd
on: [push]
jobs:
    Build:
        runs-on: ubuntu-latest
        Steps:
            - name: Checkout git repository
              uses: actions/checkout@v2
            - name: Setup dotnet
              uses: actions/setup-dotnet@v1
              With:
                dotnet-version: '6.0.x'
            - name: Build solution
              run: dotnet build
            - name: Test solution
              run: dotnet test
```

4. Ensure this workflow runs exclusively after commits on the master branch by replacing the **on** clause with the specification that this workflow will be restricted to the master branch. To do so, paste the following code there:

```
name: cd
on:
    push:
        branches:
            - 'main'
            - 'master'
jobs:
    Build:
            runs-on: ubuntu-latest
            Steps:
                - name: Checkout git repository
                  uses: actions/checkout@v2
                - name: Setup dotnet
                  uses: actions/setup-dotnet@v1
                  With:
                    dotnet-version: '6.0.x'
                - name: Build solution
                  run: dotnet build
                - name: Test solution
                  run: dotnet test
```

5. Keep the steps of the CI pipeline just as they were.
6. Add an extra step that will publish your application to an output directory so that you can deploy the content of that directory to Azure. Place the following step after the test step of your pipe:

```
    - name: Publish API
      run: dotnet publish -c Release -o ${{env.DOTNET_ROOT}}/app
```

Here, the **`-c`** flag is used to specify the build configuration whereas **`Release`** and the **`-o`** flag is used to specify the location where you want the publishing artifacts to be placed. The **`env`** keyword allows you to access the repository **environment variables**. The **`DOTNET_ROOT`** variable is automatically set to where the .NET SDK is located in your build server.

> **NOTE**
>
> Here publishing artifacts refer to **`*.dll`** or **`.exe`** files along with other deployable resources.

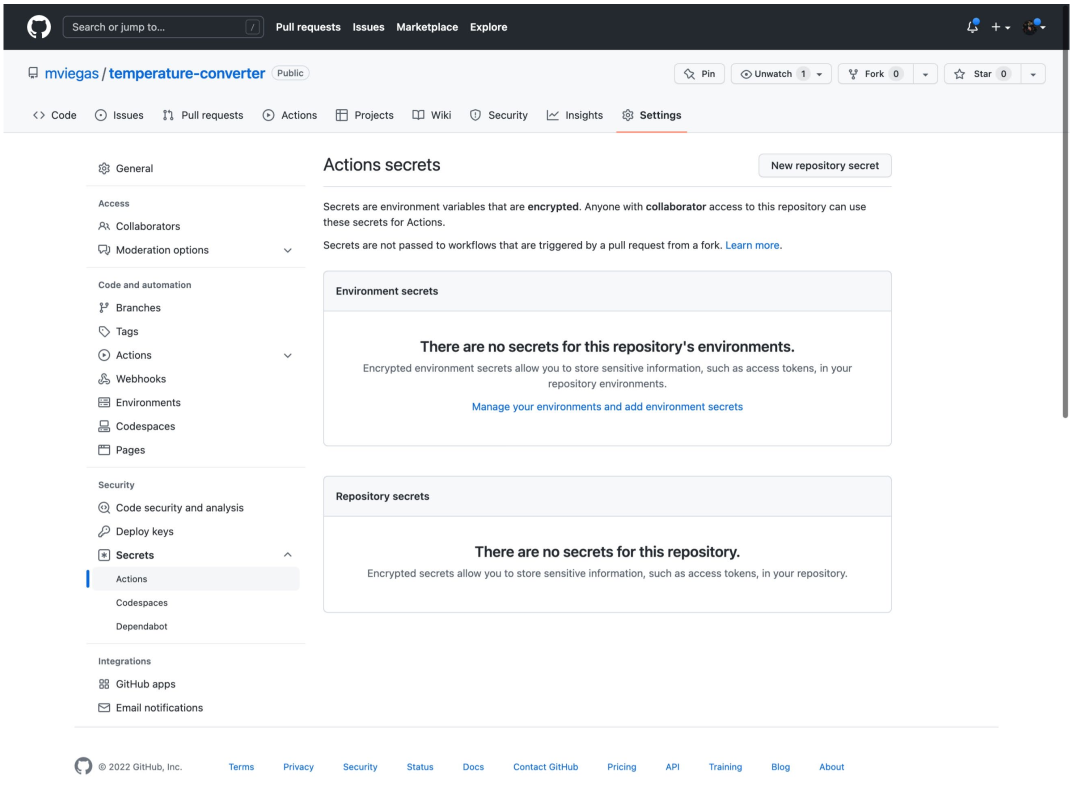
7. Add the deploy step, which will push the changes to your Azure web application as the last step of your job:

```
- name: Deploy to Azure
   uses: azure/webapps-deploy@v2
   with:
     app-name: ${{ secrets.AZURE_WEBAPP_NAME }}
  publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE  }}
package: ${{env.DOTNET_ROOT}}/app
```

Note the **`secrets`** keyword. This keyword allows you to access environment variables you defined within your repository configuration in GitHub. The difference here from a default environment variable is that a secret is encrypted.

8. To set the secrets that you can see in the preceding snippet, navigate to your repository on GitHub.

9. Go to the **`Settings`** option and choose the **`Secrets`** option:



Actions secrets

Secrets are environment variables that are **encrypted**. Anyone with **collaborator** access to this repository can use these secrets for Actions.

Secrets are not passed to workflows that are triggered by a pull request from a fork. Learn more.

Environment secrets

There are no secrets for this repository's environments.

Encrypted environment secrets allow you to store sensitive information, such as access tokens, in your repository environments.

Manage your environments and add environment secrets

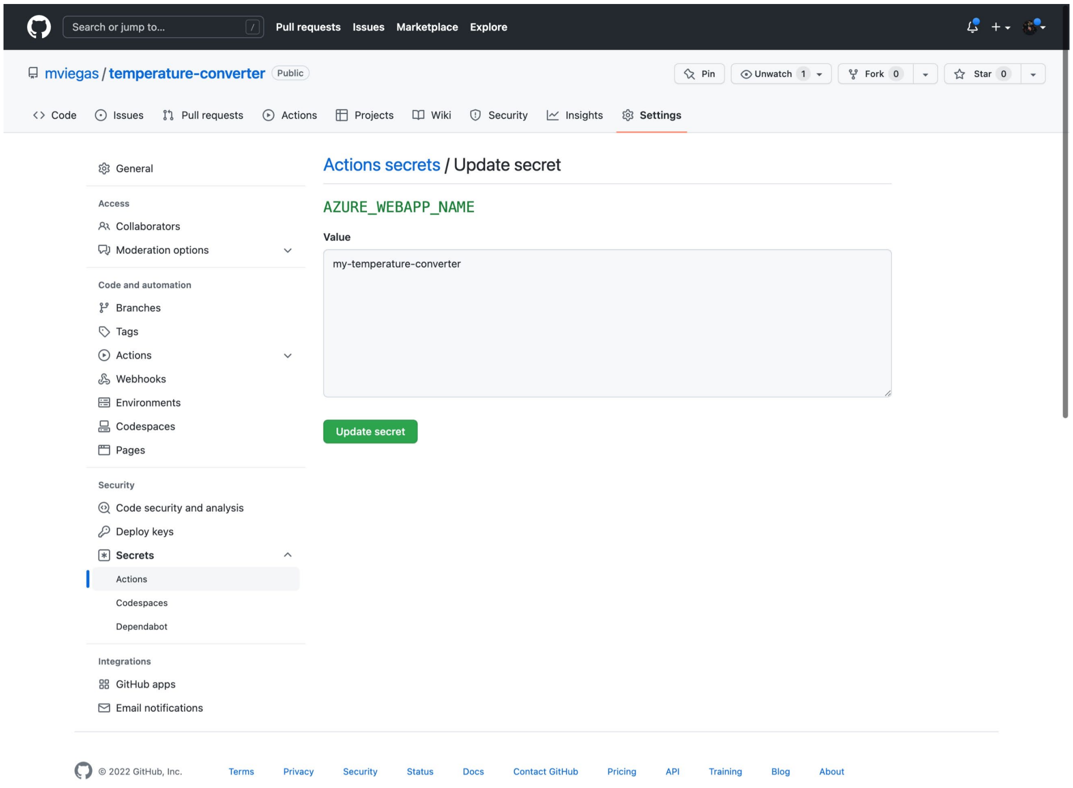
Repository secrets

There are no secrets for this repository.

Encrypted secrets allow you to store sensitive information, such as access tokens, in your repository.

Figure 11.41: Setting secrets to the repository

10. Click on the **`New Repository Secret`** option
11. Enter **`AZURE_WEBAPP_NAME`** as the secret name:



Actions secrets / Update secret

AZURE_WEBAPP_NAME

Value

my-temperature-converter

Figure 11.42: Creating GitHub Actions secrets

12. Navigate to the web application overview page on the Azure portal.
13. Enter the application name as the value for the preceding secret.

14. On the GitHub secrets page, create a new repository secret with the name **`AZURE_WEBAPP_PUBLISH_PROFILE`**:

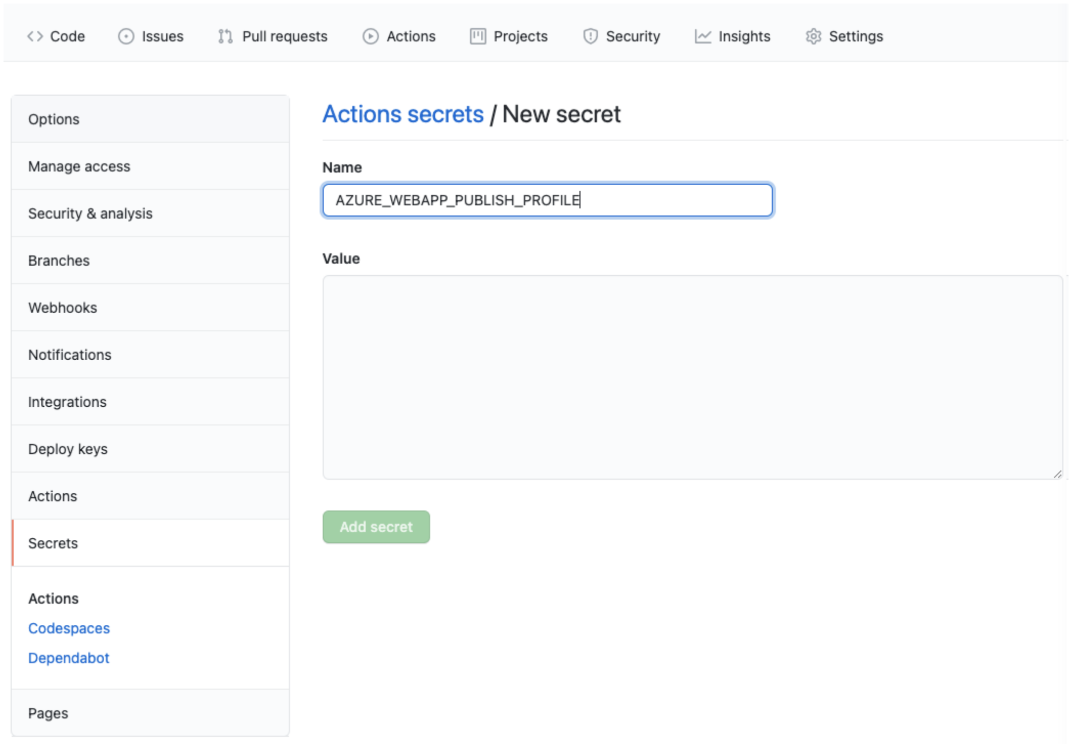
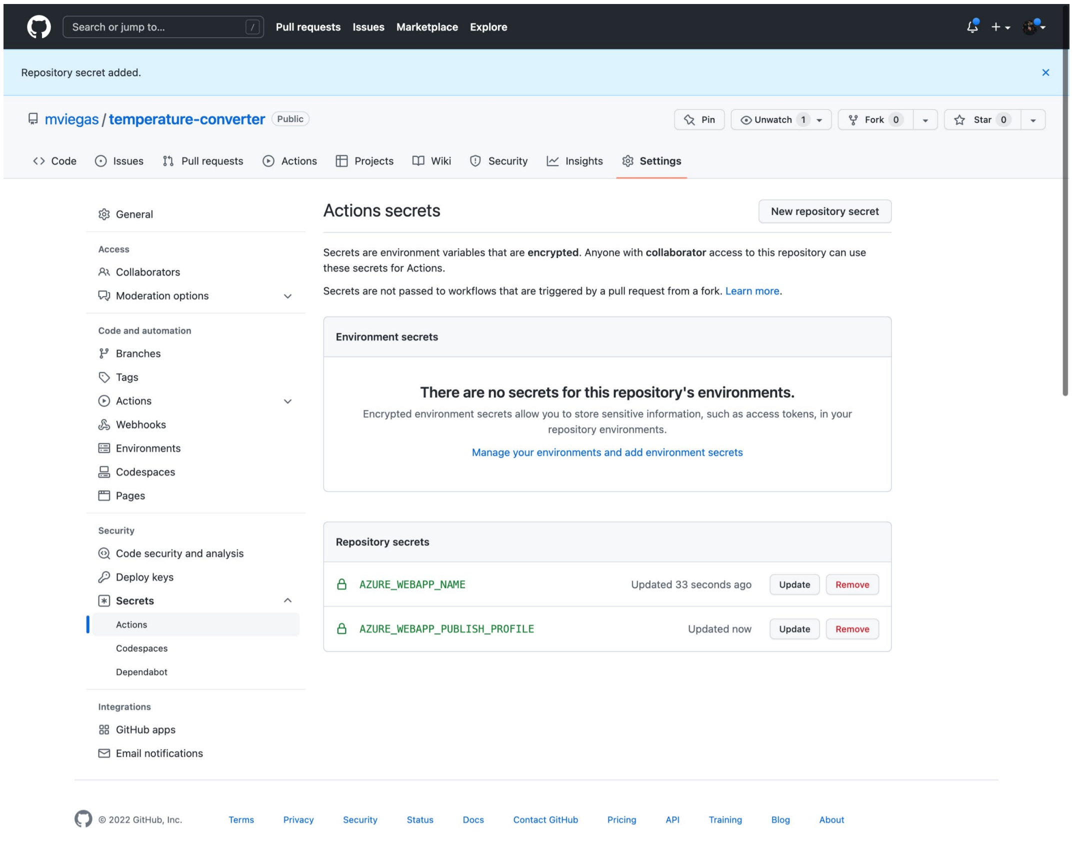


Figure 11.43: Creating a new repository secret

15. On the Azure portal, still on the web application overview page, click on **`Get publish profile`**. A file will get downloaded.

16. Open this file and place its contents as the value of the **`AZURE_WEBAPP_PUBLISH_PROFILE`** variable you created in the previous step. You will be able to see both the repository secrets in *Figure 11.44*:



Repository secret added.

Actions secrets

Secrets are environment variables that are **encrypted**. Anyone with **collaborator** access to this repository can use these secrets for Actions.

Secrets are not passed to workflows that are triggered by a pull request from a fork. Learn more.

Environment secrets

There are no secrets for this repository's environments.

Encrypted environment secrets allow you to store sensitive information, such as access tokens, in your repository environments.

Manage your environments and add environment secrets

Repository secrets

AZURE_WEBAPP_NAME	Updated 33 seconds ago
AZURE_WEBAPP_PUBLISH_PROFILE	Updated now

Figure 11.44: Both repository secrets visible on the page

17. Commit your changes to the application repository with the following commands:

```
git add .
git commit -am 'Added CD workflow'
```

18. Push it to the remote repository on GitHub and watch the action run.

19. After the action succeeds, navigate to your application URL, and see it deployed:

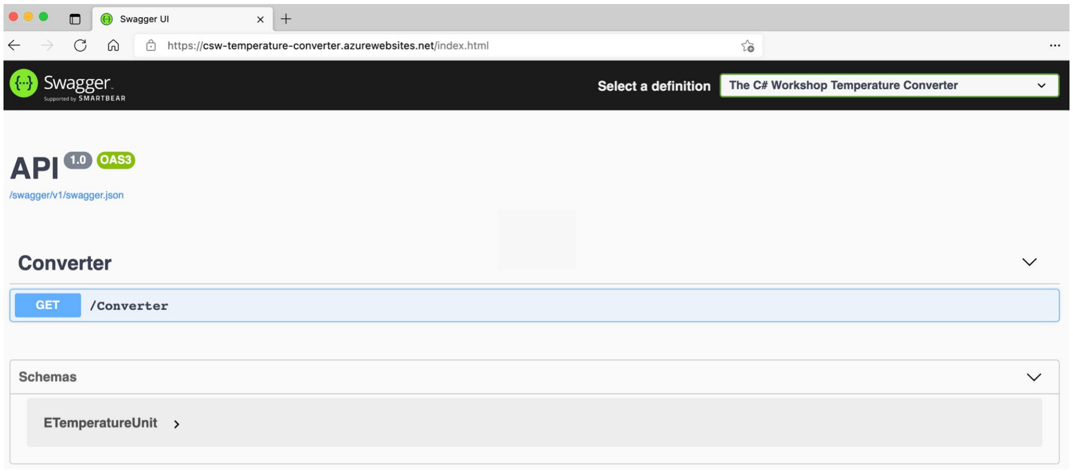


Figure 11.45: Final output of the deployed web application

Your complete workflow file is as follows:

cd.yml

```
name: cd
on:
    push:
        branches:
            - 'main'
            - 'master'
jobs:
    deploy:
        runs-on: ubuntu-latest
        steps:
            - name: Checkout git repository
              uses: actions/checkout@v2

            - name: Setup dotnet
              uses: actions/setup-dotnet@v1
```

You can find the complete code here: https://packt.link/xcTE1.

In this activity, you deployed an application to Azure automatically. The deployment happened every time a commit reached the master branch and it successfully built and tested the application inside the workflow. You also noticed that when the tests failed, the application did not get published.

You've just completed the last activity of the book.