Lesson 2

Some of the exercises in this and upcoming lessons are part of the mandatory assignments. These lessons are marked as *Mandatory*.

Exercise 3, *a list of phone numbers* (Mandatory)

1. Assign a phone number to the *phone* property of each customer in the controller class and update the view to display a customer's list with phone numbers beneath the address labels:

   [*firstname*] [*lastname*] ([*phone*])

2. There is a problem with the model, because each customer can only have a single phone number. Let's correct that.

   Open the `Customer.cs` file and delete the `Phone` property and add a new `PhoneNumbers` *property*. Instead of `sting,` the type must be a `List` collection of type `string`.

   The `PhoneNumbers` property must be read-only:

```
// read only property
1. public List<string> PhoneNumbers { get; } = new List<string>();
```

   The `PhoneNumbers` list is `null` as long as it is not initiated. Therefore a new object of type `List<string>` is assigned to `PhoneNumbers`, meaning the default value of PhoneNumbers will be an empty List of string element.s

   With this change, it is now possible to store more than one phone number for each customer.

3. Add a new method AddPhone with the header:

```
public void addPhone(string phone)
```

4. Change the view by adding a list of phone numbers for each customer:

```
<p>[firstname] [lastname] ([comma separated list of phone
numbers])</p>
```

   Tip:
   To display a comma-separated list of strings you can use the `join` method of the `string` object:

```
string.Join (", ", theList);
```

The first parameter is the string that separates the items in the `List`, and the second parameter is the `List` you want to join as a single string.

Exercise 4, *calculate age for birthday* (Mandatory)

1. Open the Customer class file and add a field birthDate and a property BirthDate of type DateTime.

2. Write a read-only property, `Age` that returns the age of the Customer.

   Tip: To calculate the age of a customer from birth date you can use this algorithm:

```
DateTime birthDate; // field

// code inside the get block of the Age property
DateTime now = DateTime.Now;
int age;

age = now.Year - birthDate.Year;

// calculate to see if the customer hasn't had birthday yet
// subtract one year if that is so
if (now.Month < birthDate.Month ||
     (now.Month == birthDate.Month && now.Day < birthDate.Day))
{
     age--;
}
```

3. Modify the `BirthDate` property to ensure that only dates that calculate an age between 0 and 120 are accepted.

   Tip: Use the `set` block of the `BirthDate` property to validate for a realistic age:

```
public DateTime BirthDate
{
   set {
        if (expression)
        {
           throw new Exception("Age not accepted");
        }
        else
```

```
            {
                birthDate = value;
            }
        }
        get { return birthDate; }
    }
```

4. Test the code by assigning a birth date to one or more `Customer` objects.

5. Add Razor code to the view and display the age of one of the `Customer` objects, like: Peter Thompson is 24 years old.

Exercise 5, Rolling a Dice (Optimal)

In this exercise, you must simulate dice rolls, and keep track of the result of a sequence of rolls.

1. Create a new empty ASP.NET MVC project for this exercise and name it *lesson02* or *diceroll* as you prefer.

2. Modify the `Startup.cs` class to match this initial setup:

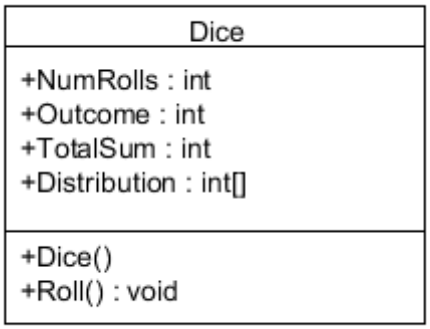
```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
        }

        app.UseStaticFiles();
        app.UseMvcWithDefaultRoute();
    }
}
```

3. Create `Controllers`, `Models` and `Views` folders.

4. In the Models folder, create a new class `Dice` as an implementation of this UML-diagram:



In the constructor, you must set the right initial values:

```
public Dice() {
      eyes = 6;
      numRolls = 0;
      totalSum = 0;
      distribution = new int[eyes];
}
```

The `numRolls` is the total number of rolls of the dice. The `outcome` is what the dice landed at, and `totalSum` is the total sum of eyes in all dice rolls. The `distribution` field is used to keep track of how many times the dice landed on each of the possible outcomes (1, 2, 3, 4, 5, and 6):

Index = Eye minus 1	Outcome
0	Number of ones
1	Number of twos
2	Number of threes
3	Number of fours
4	Number of fives
5	Number of sixes

Write the `Roll()` method. Inside that method, you must use the `Random` class to set the `outcome` field, and update the `numRolls`, `totalSum`, and `distribution` fields.

5. Create a new *DiceRoll* controller for the program.

You must have two Index action methods. One for the initial request and one for post requests:

```
public IActionResult Index()
{
    return View();
}
[HttpPost]
public IActionResult Index(IFormCollection fc) {
    return View();
}
```

The overall result of dice rolls is stored in as a Dice object.

6. To store that result between dice rolls you can use Sessions. To do that you must install the NuPackage named Microsoft.AspNetCore.Session.

7. Open the Startup.cs file and add these statements in the beginning of the ConfigureServices method

```
services.AddDistributedMemoryCache();

services.AddSession(options => {
    options.IdleTimeout = TimeSpan.FromMinutes(1); //You can set Time
      });
```

and add this statement:

```
app.UseSession();
```

to the Configure method.

8. As we want to store Dice – which is a complex object – as a session, we need to create an Session extension that enables us to serialize the object into a JSON string, and deserialize it back to a Dice object.

To do that you must first create a new folder named Infrastructure, and add this SessionExtensions class that serialize and deserialize any object:

```
public static class SessionExtensions
{
    public static void SetJson(this ISession session, string key, object value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T GetJson<T>(this ISession session, string key)
```

```
        {
            var value = session.GetString(key);

            return value == null ? default(T) :
JsonConvert.DeserializeObject<T>(value);
        }
    }
```

Read more about sessions

- How To Use Sessions In ASP.NET Core (c-sharpcorner.com)
- Session and app state in ASP.NET Core (docs.microsoft.com)

9. You can now use session variables and the `SetJson` and `GetJson` extesions methods from inside the controller. If you include the `Infrastructure` namespace, you can insert this code into the second action method reacting on HTTP POST requests:

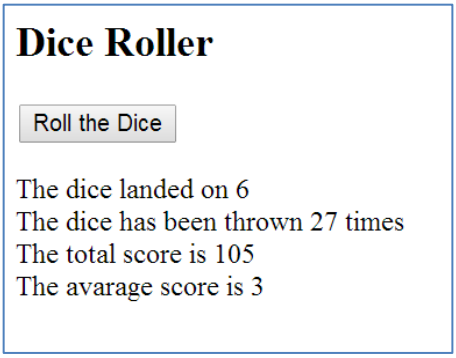
```
Dice dice;
     if (HttpContext.Session.GetJson<Dice>("dice") == null)
     {
         dice = new Dice();
         dice.Roll();
         HttpContext.Session.SetJson("dice", dice);
     }
     else
     {
         dice = HttpContext.Session.GetJson<Dice>("dice");
         dice.Roll();
         HttpContext.Session.SetJson("dice", dice);
     }
```

   If the session variable is not set, you create a new instance of `Dice`, roll the dice and store the result in the session variable named "`dice`". Otherwise, you read the `Dice` object from the session variable and roll the dice, and store the state of the `Dice` object back to the session.

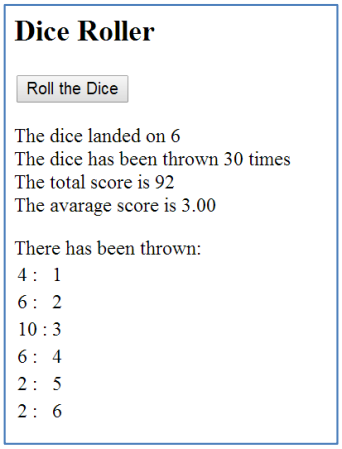
10. You can then store the `Dice` object as a `ViewBag` key:

```
ViewBag.Dice = dice;
```

11. Create a view with a button that calls the controller, which rolls the Dice. Return a view which displays the result:

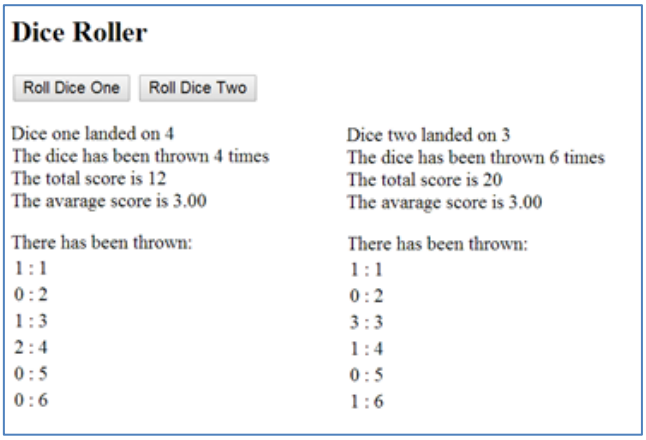


12. Update the `distribution` field, and display the distribution of eyes:



Exercise 6, Rolling Dices (Optimal)

Create a new *DicesRoll* controller with a view and add one more button and a dice, and present the result of the two independent rolls:



/ Jes Arbov, 5 September 2019