Blazor Fundamentals

Building interactive web apps with C#

"Just enough to be dangerous…"

https://github.com/andrewstellman/blazor-training

©2023 Stellman and Greene Consulting LLC, all rights reserved

Andrew Stellman

I've been training people on software development and other technical topics for over 20 years. *Head First C#*, one of the six books I've written for O'Reilly, is going into its 5th edition. I'm also a full-time software developer and team lead, and I'm passionate about all things code.

What's in this course

A really quick overview of what we're going to learn… and then we'll jump right into it.

What is Blazor?

Blazor is a technology that lets you build interactive web pages using C#.

- You can use Blazor to build web pages that use HTML and CSS for **design** and C# for **behavior**.
- Blazor is part of **ASP.NET**, the free, cross-platform, open-source framework for building web apps and services with .NET and C#.

“Just enough to be dangerous…”

- The goal of this course is to give you a **fast start** with Blazor.
- **This is <u>not</u> a comprehensive course in Blazor.**
- What we <u>will</u> do: get you up and running quickly and give you the tools to start making interactive web pages today.
- What we <u>won’t</u> do: make you an expert in Blazor or do a deep dive into specific Blazor features.
- You’ll probably have some questions that we won’t have time to answer. But we’ll give you enough knowledge so you can start answering those questions yourself.

This is <u>not</u> a course in web design!

- If you're new to building web pages, this course will include **just enough of the basics of web design** so you can build great-looking Blazor apps.
- If you're an expert in designing web pages, we'll make those parts quick enough so you **don't get bored**!
- We're assuming that you understand the absolute basic ideas behind how a web browser works: that it downloads text marked up with HTML and styled with CSS.
- **You don't need to have experience writing HTML or CSS.** We'll give you all of the HTML and CSS code you need to build the apps in this course, along with tools and explanation to help you understand it.

How this course is structured

- In the main **Presentation** sections, I'll demonstrate writing code in Blazor. You're encouraged to follow along in Visual Studio.
- In the **Discussion** sections, I'll talk a little more about what I just showed you.
- In the **Exercise** sections, I'll give you time to keep working on your own. You can download the code and slides from the GitHub page for this course: https://bit.ly/blazor-training
- This course is divided into four parts. Each part ends with a **Q&A** section where I answer your questions.

How this course is structured

We'll jump right into Blazor, creating a Blazor WebAssembly app that uses C# code to create an interactive web page.

Games are a great way to learn, so we'll start with a simple math quiz game. Then we'll get started on a bigger project, building an animal matching game.

We'll modify the animal matching game to handle mouse clicks and respond to events.

Finally, we'll learn about .NET minimal APIs and use them to turn the animal matching game into a full-stack web app.

Part 1
Blazor basics

Let’s start building web pages that use C#

Spin up your first Blazor app

Get up and running *fast*.

Install the ASP.NET components for Visual Studio 2022

This is an interactive training session. If you want to follow along on your own, you'll need to make sure you have the **ASP.NET and web development** components installed in Visual Studio 2022. We'll be building one console app as well, so make sure **.NET desktop development** is also installed.

This is a great time to kick off the installation if you haven't done so already.

Create and run a new Blazor app in Visual Studio 2022

Create a C# Blazor WebAssembly App

Choose a project name and location – name it **My_First_Blazor_App**.

Choose default options:

- Target the current version of .NET
- Select “None” for “Authentication type”
- Make sure “Configure for HTTPS” is checked

Once the project is created, start the app just like any other app – choose Debug >> Start Debugging (F5) or click the Run button.

Create and run a new Blazor app in Visual Studio for Mac

Create a C# Blazor WebAssembly App

Choose default options

- Target the current version of .NET
- Select “No Authentication”
- Make sure “Configure for HTTPS” is checked

Choose a project name and location – name it My_First_Blazor_App

Once the project is created, start the app just like any other app – choose Debug >> Start Debugging (F5) or click the Run button.

If you don’t see “Microsoft Edge” or “Google Chrome” displayed here, make sure you have Edge or Chrome installed, then click the browser name and choose one of them.

Run your app! And trust the self-signed certificate

The first time you run a Blazor app from Visual Studio, you may get prompted to trust the ASP.NET Core SSL certificate.

Trust the certificate, and answer “Yes” to security warnings.

You can also create a Blazor app from the command line

Run the following command:

```
dotnet new blazorserver -o
My_First_Blazor_App
```

The .NET CLI will create the My_FirstBlazor_App folder with a new Blazor app.

The first time you create a Blazor app, you'll need to trust the developer HTTPS certificates:

```
dotnet dev-certs https --trust
```

Make sure either Edge or Chrome is set as your default browser. Use the .NET CLI to run the app:

```
dotnet watch
```

This will launch a browser window with the new app. When you're done, close the browser window, then go back to the .NET CLI and press ^C.

C:\Windows\system32\cmd.exe - dotnet watch

```
C:\Users\Public\Projects>dotnet new blazorserver -o My_First_Blazor_App
The template "Blazor Server App" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/aspnet
core/6.0-third-party-notices for details.

Processing post-creation actions...
Running 'dotnet restore' on C:\Users\Public\Projects\My_First_Blazor_App\My_First_Blazor_App.csp
roj...
  Determining projects to restore...
  Restored C:\Users\Public\Projects\My_First_Blazor_App\My_First_Blazor_App.csproj (in 79 ms).
Restore succeeded.


C:\Users\Public\Projects>dotnet dev-certs https --trust
Trusting the HTTPS development certificate was requested. A confirmation prompt will be displaye
d if the certificate was not previously trusted. Click yes on the prompt to trust the certificat
e.
The HTTPS developer certificate was generated successfully.

C:\Users\Public\Projects>cd My_First_Blazor_App

C:\Users\Public\Projects\My_First_Blazor_App>dotnet watch
watch : Hot reload enabled. For a list of supported edits, see https://aka.ms/dotnet/hot-reload.
 Press "Ctrl + R" to restart.
watch : Building...
  Determining projects to restore...
  All projects are up-to-date for restore.
  My_First_Blazor_App -> C:\Users\Public\Projects\My_First_Blazor_App\bin\Debug\net6.0\My_First_
Blazor_App.dll
watch : Started
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7007
```

We'll be using Visual Studio (<u>not</u> VSCode) to debug Blazor apps. If you're following along and doing the exercises, please make sure you're using Visual Studio 2022 or Visual Studio for Mac 2022.

Index

https://localhost:7123

My_First_Blazor App

About

Home

Counter

Fetch data

Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our brief survey and tell us what you think.

Take a closer look at the URL. It's got a secure HTTPS connection to localhost – that means it's connecting to a server running on the same computer as the browser. In this screenshot it's connecting on port 7123, your browser may be connecting on a different port.

Try clicking on these navigation menu items to bring up different pages.

When you're done, close the browser window that popped up. By default, Visual Studio will automatically stop your app when the browser window closes.

When you change this line in Pages/Index.razor and rerun your app, you'll see a change on the landing page.

Blazor uses **Razor**, a markup syntax that lets you embed C# code in your web pages.

- Files that end with .razor define **Razor components**, which are sometimes informally called **Blazor components**.
- In this default Blazor web app, the main landing page is defined in the Pages/Index.razor file.

Digging into Blazor code

Let's have a look at that Counter page

Open the Counter page in your app

The page title is "Counter" and the URL ends with /counter.

Run your app again and click the Counter item in the navigation menu.

Click the button and the counter will go up.

Opening Counter.razor to see how it works

Everything between the opening <h1> tag and closing </h1> tag is displayed as a Heading 1. You can use <h2> (which is smaller than Heading 1) through <h6> (the smallest heading).

My_First_Blazor_App - Counter.razor

Counter.razor

```
1   @page "/counter"
2
3   <PageTitle>Counter</PageTitle>
4
5   <h1>Counter</h1>
6
7   <p role="status">Current count: @currentCount</p>
8
9   <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
10
11  @code {
12      private int currentCount = 0;
13
14      private void IncrementCount()
15      {
16          currentCount++;
17      }
18  }
```

108 % No issues found Ln: 19 Ch: 1 SPC CRLF

Everything between the opening <button> tag and closing </button> tag is displayed inside a button. The class="..." is an HTML class – in this case, "btn" tells it to render it as a button, and "btn-primary" tells it to style it as a primary button.

Everything between the opening tag and closing tag is a paragraph element. It's displayed as text with a line break at the end.

How the Counter page sets its title and URL

This is the @page directive. It tells the Blazor app that the URL for this page is /counter.

My_First_Blazor_App - Counter.razor

Counter.razor

```
1   @page "/counter"
2
3   <PageTitle>Counter</PageTitle>
4
5   <h1>Counter</h1>
6
7   <p role="status">Current count: @currentCount</p>
8
9   <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
10
11  @code {
12      private int currentCount = 0;
13
14      private void IncrementCount()
15      {
16          currentCount++;
17      }
18  }
```

108 % No issues found Ln: 19 Ch: 1 SPC CRLF

This is a PageTitle component. Blazor apps use it to set the title of the page.

How the Counter page runs C# code

My_First_Blazor_App - Counter.razor

Counter.razor

```
1   @page "/counter"
2
3   <PageTitle>Counter</PageTitle>
4
5   <h1>Counter</h1>
6
7   <p role="status">Current count: @currentCount</p>
8
9   <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
10
11  @code {
12      private int currentCount = 0;
13
14      private void IncrementCount()
15      {
16          currentCount++;
17      }
18  }
```

108 % No issues found Ln: 19 Ch: 1 SPC CRLF

The @code directive is followed by C# code inside { brackets }. This code is part of the page. It includes a private int field and a method called IncrementCount that increments that field.

@currentCount tells the page to insert the value of the currentCount variable in the C# code. The page will update any time the value changes.

The @onclick event tells the button to call the IncrementCount method in the C# code.

The C# and .NET code runs on a .NET IL interpreter implemented in WebAssembly

Modify the code for the Counter page

Add these lines of code!

My_First_Blazor_App - Counter.razor*

Counter.razor*

```
1   @page "/counter"
2
3   <PageTitle>Counter</PageTitle>
4
5   <h1>Counter</h1>
6   <h4>Updated at @timestamp</h4>
7
8   <p role="status">Current count: @currentCount</p>
9
10  <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
11
12  @code {
13      private int currentCount = 0;
14      private string timestamp = DateTime.Now.ToLongTimeString();
15
16      private void IncrementCount()
17      {
18          currentCount++;
19          timestamp = DateTime.Now.ToLongTimeString();
20      }
21  }
```

108 % No issues found Ln: 22 Ch: 1 SPC CRLF

Modify the HTML for the Counter page

Notice how the counter and timestamp reset when you navigate away from the page and back again.

Change the HTML markup to style the timestamp differently – try each of these:

```
<h4>Updated at @timestamp</h4>
<h2>Updated at @timestamp</h2>
<button class="btn btn-primary">
   Updated at @timestamp
</button>
<p>Updated at @timestamp</p>
```

Instead of restarting the app, try clicking Hot Reload (Windows only) to reload:

Debug your C# code in Visual Studio

Place a breakpoint on a line in the @onclick event handler method.

- If your app isn't running, start it up and navigate to the Counter page.
- Click the button. The breakpoint triggers!
- Use Step Over, Step Out, watches, and other standard debugging tools just like any other .NET app.

Discussion

What did we just do?

Your Blazor WebAssembly app runs entirely in the browser

- Blazor WebAssembly (WASM) apps run on the client side.
- That means all of your app's C# code runs inside the browser using a **WebAssembly-based .NET runtime** that gets loaded along with the web page and interprets the compiled IL code.
- All of the files are static and are downloaded by the browser.
- When you run your Blazor WebAssembly app inside of Visual Studio, it's running on an IIS Express development web server (that's what the https://localhost URL connects to).

There are other ways that Blazor runs

- In a Blazor Server app, code is executed on the sever, and UI updates are sent to the browser.
- A Blazor Hybrid app blends native controls and web technology. We're not covering Blazor Hybrid apps.
- The goal is to get you up and running quickly with Blazor WebAssembly apps so you can start experimenting and learning on your own. We'll come back to Blazor Server apps at the end of the session when we talk about full-stack Blazor.

Your web app is running in *IIS Express*

IIS Express is a lightweight, self-contained web server optimized for developers.

- IIS Express is made to help you develop and test your web apps.
- It doesn't have to run as a service, and it doesn't require administrator access.
- When you set up your Blazor project, the template automatically generated Properties/launchSettings.json which contains the IIS Express configuration.
- Visual Studio 2022's default setting is to automatically close the browser when debugging stops, and automatically stop debugging when the browser stops.

The self-signed certificate lets you debug your app using HTTPS

HTTPS is the secure protocol that websites use to encrypt web traffic. Almost every website you visit uses it.

HTTPS requires a trusted certificate. When you connect to a web page, your browser connects to a certificate authority—a server run by a small number of companies and nonprofit organizations—from a list that came pre-installed with it and downloads the certificate for that website.

Your computer (localhost) doesn't have a certificate registered with a CA that your browser knows about. Installing the self-signed certificate lets your browser and IIS Express communicate using HTTPS.

You can trust the cert from the command line:

```
dotnet dev-certs https --trust
```

And you can remove it from the command line:

```
dotnet dev-certs https --clean
```

Restart Visual Studio after each of those commands. If you get a "debugger operation failed" error, try cleaning and trusting the certificate again.

Blazor apps use Razor components

- The Counter page you modified is a **Razor component** (which people will sometimes call a “Blazor component”).
- A component is a self-contained portion of a web page’s user interface (UI) with processing logic to enable dynamic behavior.
- Components can be reused.
- Components can be nested – one component can contain another component.
- Razor components live in files with the .razor extension, which have a combination of C# and HTML.

Add a component to your app

You'll add a component that takes input – and get a really quick crash course in HTML if you need it

Add a component

Use the Solution Explorer to add a new Razor component. Name it Controls.razor. It will be created with just a header and empty @code section.

Right-click on Pages and choose Add >> Razor Component. It will add a .razor file to the folder.

Add a @page directive and <PageTitle> tag, just like the ones **you** saw in Counter.razor:

```
@page "/controls"

<PageTitle>Experiment with Controls</PageTitle>
```

In Visual Studio 2022 for Mac, right-click on Pages in the Solution window, choose Add >> New File... and choose Razor Component from the ASP.NET Core section.

Add the component to the navigation menu

Expand the Shared folder and open NavMenu.razor. This file contains the HTML for the navigation menu on the side of the web page.

- Find the three navigation items – they all start with and end with
- Copy one of them and paste it just above the closing </nav> tag.
- Replace the href="..." link with **"controls"** and the text with **Controls** – the screenshot to the right shows the copied section with the two changes.
- Save it and run your app. Now you'll see a fourth navigation item called Controls, and when you click it you'll navigate to your Razor component.

Add a field and an event handler method

- We saw how the Counter component uses the @onchange event to call an event handler method in the C# code.
- The @onchange event handler can take a ChangeEventArgs parameter, which contains information about a change event – for example, every time a text box changes.
- The ChangeEventArgs object has a Value property that contains the value that was changed. Its type is object? so we need to call its ToString method.
- We'll make our UpdateValue method update a string field called displayValue.

```
@code {

    private string displayValue = "";

    private void UpdateValue(ChangeEventArgs e)
    {
        displayValue = e.Value.ToString();
    }

}
```

Here's the rest of the HTML code for the page

If you're not used to looking at HTML, this may look complex. We'll spend a few minutes breaking it down.

Almost all of the HTML code we look at will be very similar to this.

```
<div class="container">
    <div class="row">
      <h1>Experiment with controls</h1>
    </div>
    <div class="row mt-2">
       <div class="col">
          Enter text:
       </div>
       <div class="col">
          <input type="text" class="form-control"
                 @onchange="UpdateValue"/>
       </div>
    </div>
    <div class="row mt-5">
       <h2>
          Here's the value: <strong>@displayValue</strong>
       </h2>
    </div>
</div>
```

Your Blazor web app uses Bootstrap

Bootstrap is a very popular open-source toolkit for building fast, responsive sites. Our app uses the Bootstrap **grid system** to lay out the page contents.

```
<div class="container">
    <div class="row">
        ...
    </div>
    <div class="row mt-2">
        ...
    </div>
    <div class="row mt-5">
        ...
    </div>
</div>
```

The tag at the top has its class set to "container" so Everything between the opening tag at the top and closing tag at the bottom is contained inside of it. Each row has its own opening and closing tag.

Learn more about Bootstrap: https://getbootstrap.com/

Let's take a closer look at the middle <row>

The bootstrap grid system has rows. Each row can have up to 12 evenly spaced columns. The col class defines a column that spans half of those 12 columns.

```
<div class="row mt-2">
  <div class="col">
    Enter text:
  </div>
  <div class="col">
    <input type="text" ... />
  </div>
</div>
```

```
<div class="container">
  <div class="row">
    <h1>Experiment with controls</h1>
  </div>
  <div class="row mt-2">
    <div class="col">
      Enter text:
    </div>
    <div class="col">
      <input type="text" class="form-control"
             @onchange="UpdateValue"/>
    </div>
  </div>
  <div class="row mt-5">
    <h2>
      Here's the value: <strong>@displayValue</strong>
    </h2>
  </div>
</div>
```

This row contains a level 1 header with the text "Experiment with controls" – and since no columns were specified the text spans the entire row.

This row has 2 units of top spacing and contains two columns. They both have the class "col" so of them takes up half of the width of the row. The left column has the text "Enter text:" and the right column has a text input box with its @onchange event hooked up to the UpdateValue method.

The bottom row has 5 units of top spacing. It contains a level 2 header that has the text "Here's the value:" followed by the contents of the displayValue field. The tag make the text bold.

Add a slider (or “range”) control to enter numbers

Here’s HTML markup to add a new row. It should go just below the closing tag from the row with the text input.

It’s almost identical to the row with the text input—in fact, you can copy and paste the entire section, then make three changes:

- Change the text in the left column to “Pick a number:”
- Change the input type to “range”
- Make its @onchange event call an event handler called UpdateNumericValue

```
<div class="row mt-2">
    <div class="col">
        Pick a number:
    </div>
    <div class="col">
        <input type="range" class="form-control"
         @onchange="UpdateNumericValue"/>
    </div>
</div>
```

Add the event handler for the slider control

Add the UpdateNumericValue event handler method to the @code section.

- Since e.Value is an object, we can use int.TryParse to convert it to an integer value.
- The range (slider) control will always return a numeric value, so we didn't necessarily have to do that (just like we're not doing null checks because we know the controls will pass values to the event handler methods).
- What do you think would happen if we switched the @onchange event for the text box to call UpdateNumericValue instead?

```
private void UpdateNumericValue(ChangeEventArgs e)
{
    if (int.TryParse(e.Value.ToString(), out int result))
    {
        displayValue = result.ToString();
    }
}
```

Go to https://bit.ly/blazor-training for all the code that you've seen so far.

Exercise

This is your chance to try out Blazor! Switch to Visual Studio and create a new Blazor WebAssemby App. If you've been following along throughout this first part, this will give you time to finish up. If you're already done, check the GitHub page for a few ways to experiment with your code.

Q&A

If you have any questions, enter them in the Q&A widget.

5 minute break

Grab some water, check your email, have a look at your social media… we'll be back here in five minutes.

Part 2
Using Blazor in a bigger app

We'll build a math quiz game, then get started on the bigger animal matching game

The console app

We're starting with a math quiz game that works as a console app. This game is adapted from a puzzle in Chapter 5 of *Head First C#*.

The Question class generates questions and checks answers

- This class picks a random operator, either addition or multiplication, and stores it in the Operator property.
- It also picks two random numbers from 1 to 9, which it stores in N1 and N2.
- It has a nullable string property Answer
- Its Check method returns true if Answer contains the correct answer to the question N1 Operator N2
- So if N1 is 3, Operator is *, and N2 is 5, Check will only return true if Answer is "15".

```
class Question
{
    public Question()
    {
        Operator = (Random.Shared.Next(2) == 1) ? "+" : "*";
        N1 = Random.Shared.Next(1, 10);
        N2 = Random.Shared.Next(1, 10);
    }

    public int N1 { get; private set; }

    public string Operator { get; private set; }

    public int N2 { get; private set; }

    public string? Answer { get; set; }

    public bool Check()
    {
        if (int.TryParse(Answer, out int i))
        {
            if (Operator == "+") return (i == N1 + N2);
            else return (i == N1 * N2);
        }
        else return false;
    }
}
```

This console app quiz game uses the Question class

Here's a console app that uses the Question class to implement a math quiz game. It has a loop where it creates a new Question object, prints the question, and keeps asking for answers until Check returns true. Once the player gets the right answer, it creates a new Question object to get a new random math question. If the player presses enter it ends the game.

```
Question quiz = new();
while (true)
{
    Console.Write($"{quiz.N1} {quiz.Operator} {quiz.N2} = ");
    quiz.Answer = Console.ReadLine();
    if (quiz.Answer == "")
    {
        Console.WriteLine("Thanks for playing!");
        return;
    }
    if (quiz.Check())
    {
        Console.WriteLine("Right!");
        quiz = new Question();
    }
    else Console.WriteLine("Wrong! Try again.");
}
```

Running the console app quiz game

Microsoft Visual Studio Debug Console

```
Right!
1 * 9 = 9
Right!
5 + 7 = 12
Right!
6 + 4 = 11
Wrong! Try again.
6 + 4 = 10
Right!
7 * 7 = 49
Right!
3 * 5 =
Thanks for playing!

C:\Users\Public\Projects\MathQuiz\MathQuiz\bin\Debug\net6.0\MathQuiz.exe
(process 24584) exited with code 0.
Press any key to close this window . . .
```

Using the Quiz class in Blazor

Blazor apps use C# classes just like any other .NET app. Let's start our Blazor math quiz game by adding the Question class we just used to a new Blazor app.

Can you figure out how to change Blazor_Math_Quiz to something else?

Create a new Blazor app and update the navigation menu

We'll start by creating a new Blazor WebAssembly app, just like we did in Part 1—call it Blazor_Math_Quiz. We'll adjust Shared/NavMenu.razor to add the menu item for the math quiz and delete the Counter and Fetch Data items.

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="mathquiz">
        <span class="oi oi-puzzle-piece" aria-hidden="true"></span> Math Quiz
    </NavLink>
</div>
```

Changing the class to "oi oi-puzzle-piece" changes the nav menu icon to a jigsaw puzzle piece. This icon comes from Open Iconic, an open-source icon set. You can browse the icons in the Open Iconic website: https://github.com/iconic/open-iconic

Add a Razor component for your math quiz page.

Right-click on Pages and add a Razor component called MathQuiz.razor. The code for it is on the right—right now you'll just have a Bootstrap table with a single row.

You can also delete the components that you're not using: Pages/Counter.razor and Pages/FetchData.razor. The FetchData component retrieves data from a file called weather.json, so you can also delete it from the wwwroot/sample-data folder.

```
@page "/mathquiz"


<PageTitle>Math Quiz</PageTitle>


<div class="container">
    <div class="row">
        <h1>Math quiz!</h1>
    </div>
</div>


@code {


}
```

Add the Question class to the project

The web app version of the math quiz game will reuse the same Question class as the Console App version.

Add the class as usual, by right-clicking on the project and choosing *Add >> Class…* from the menu (in Visual Studio for Mac, choose *Add >> New Class…* from the menu). Name it Question.cs.

Visual studio will add an empty class called Question in a namespace that matches your project name. We'll reuse the class from the console math quiz game by pasting it into the new Question.cs file.

```
namespace Blazor_Math_Quiz
{
    class Question
    {
        public Question()
        {
            Operator = (Random.Shared.Next(2) == 1) ? "+" : "*";
            N1 = Random.Shared.Next(1, 10);
            N2 = Random.Shared.Next(1, 10);
        }

        public int N1 { get; private set; }

        public string Operator { get; private set; }

        public int N2 { get; private set; }

        public string? Answer { get; set; }

        public bool Check()
        {
            if (int.TryParse(Answer, out int i))
            {
                if (Operator == "+") return (i == N1 + N2);
                else return (i == N1 * N2);
            }
            else return false;
        }
    }
}
```

This is the exact same Question class from the console app quiz game.

Create an instance of Question and use it on the page.

We'll **add a field** to the @code section to hold a reference to an instance of Question. We'll name it q.

Then we'll **add a row** to our Bootstrap grid that displays them, along with an input. We'll use the Question object's properties: N1 and N2 for the two numbers in the quiz, and Operator which is either "+" or "*".

Notice how the Razor page includes the properties from the Question object:

```
@q.N1 @q.Operator @q.N2 =
```

You could also include parentheses after the @, but Razor is smart enough to figure out that @q.N1 is the same thing as @(q.N1).

```
@page "/mathquiz"

<PageTitle>Math Quiz</PageTitle>

<div class="container">

    <div class="row">
        <h1>Math quiz!</h1>
    </div>

    <div class="row mt-2">
        <div class="col-2">
            @q.N1 @q.Operator @q.N2 =
        </div>
        <div class="col-2">
            <input type="text" class="form-control"/>
        </div>
        <div class="col-6">
            What's the answer?
        </div>
    </div>
</div>

@code {
    private Question q = new();
}
```

We made the code you added earlier lighter on this slide so you could see it more clearly.

This adds three columns: two narrow columns for the question and input box, and a wide column for the message. We played around with different column sizes and liked this one the best – you might like a different layout. The col-2 class makes the column span two of the 12 Bootstrap columns, while the col-6 class spans six of the 12 columns.

Add a field for the Question object.

Here's what we have so far…

Bootstrap layouts are built to be **responsive**, which means they adjust to different browser sizes. Resize your browser to make it narrower or wider and watch how the columns adjust.

Use data binding to get the answer from the input box

In the last project you used @onchange to call an event handler when an input changed, and used its e.Value parameter to figure out the new value.

But the problem is that we don't just want to get the answer. We also want to be able to clear the input—and while that's technically possible using @onchange, it's a lot of work.

Luckily, Blazor gives us an easier way to get values in and out of an input: **binding**.

When you use @bind to bind an input to a field or property, any time the input changes the field/property changes, and any time the field/property changes the input changes.

Add @bind="q.Answer" to the input:

```
<input type="text" class="form-control"
        @bind="q.Answer"/>
```

This **binds** the input the q.Answer.

Now every time the input box changes, the Answer property in the Question object will get updated *as soon as it loses focus*. And the binding is two-way – updating the Answer property will cause the input box to be updated with its new value.

Use a Blazor conditional to display a message

We want the page to display the message "Correct!" if the user enters a correct answer and hits enter or navigates to another field.

Luckily, Blazor has **expressions** that help make your markup smarter. To make the message work, we'll use an **@if conditional** that works just like an if statement in C#: it checks a condition, and if that condition is true it inserts markup.

We'll add the CheckAnswer method to the @code section, then update the right column so it uses an @if conditional to display "Correct!" in the column if the answer is correct.

```
<div class="col-6">
    @if (q.Check())
    {
        <span>Correct!</span>
    }
</div>
```

Replace the "What's the answer?" text with this @if conditional that writes Correct! only if the answer is correct.

You can put code between the {brackets} after the @if, but if you want it to just print HTML make sure you include markup (so just adding "Correct!" won't work, because it will look for a field called Correct).

Here's the C# code to add to the @code section

- We're replacing a single instance of the Question class with a collection of Question objects.
- The GenerateQuestions method clears the collection and creates a new set of questions.
- We want the page to generate questions when it loads. Luckily, there's a convenient method called OnInitialized that gets called every time the page loads. We just need to override it and have it call GenerateQuestions.

The questions field points to a collection of Question object references. We'll use it to display multiple questions on the page.

```
private List<Question> questions = new();
private const int QUESTION_COUNT = 6;


private void GenerateQuestions()
{
    questions.Clear();
    for (int i = 0; i < QUESTION_COUNT; i++)
        questions.Add(new());
}


protected override void OnInitialized()
{
    GenerateQuestions();
}
```

Add a @foreach expression around the question rows

Razor expressions are really intuitive, and the @foreach expression works exactly the way you'd expect it to work.

It iterates through a sequence (just like a C# foreach statement), repeating the contents of the brackets

```
@foreach (Question q in questions) {
    <div class="row mt-2">
        <div class="col-2">
            @q.N1 @q.Operator @q.N2 =
        </div>
        <div class="col-2">
            <input type="text"
             class="form-control" @bind="q.Answer"/>
        </div>
        <div class="col-6">
            @if(q.Check())
            {
                <p>Correct!</p>
            }
        </div>
    </div>
}
```

We're just surrounding the existing row with a @foreach that iterates through the Question collection referenced by the questions field.

Your quiz now has six questions

Razor expressions like @bind and @foreach help you make your pages do what you want them to do.

Add a method that needs a using statement

We're going to finish our quiz game by adding a message that displays the count of the number of questions the player got right, and when they got all the questions right it will be replaced with a button to generate a new quiz.

There are plenty of ways to find the number of correct questions. But for this example, let's use LINQ – and for that, we need to include a using statement.

The Razor @using directive works exactly like the using keyword does in C#. Adding a @using directive to a Razor component lets you use classes from that namespace in your C# code.

The @using directive works just like it does in C#.

```
@using System.Linq

@code {
    private int NumberCorrect() =>
            questions.Where(q => q.Check()).Count();

    private List<Question> questions = new();
    private const int QUESTION_COUNT = 6;
```

This method uses LINQ to find the number of Question objects where the Check method returns true, then return the count.

You can also put @using directives in the _Imports.razor file. Any @using directives in _Imports.razor are applied to every *.razor file in your project (but not *.cs files).

Use an @if directive with an else to display either text or a button.

We'll add one more row to the bottom of the page with a single column, giving it class "col-6" so it takes up half the width of the page.

The column contains an @if directive that calls the NumberCorrect method. If it returns a number less than the QUESTION_COUNT constant, it will display a message showing the number correct.

If the player answered all of the questions correctly, NumberCorrect() will return the same value as QUESTION_COUNT, so the else clause will trigger, causing it to display a button with an @onclick event that calls the same GenerateQuestions method that gets called when the page is initialized.

```
<div class="row mt-5">
    <div class="col-6">
        @if(NumberCorrect() < QUESTION_COUNT)
        {
            <h6>
                @NumberCorrect() out of @QUESTION_COUNT
            </h6>
        } else
        {
            <button class="btn btn-primary"
                    @onclick="GenerateQuestions">
                Generate New Quiz
            </button>
        }
    </div>
</div>
```

Refactor the C# code to call the LINQ method once

If you've used LINQ a lot, you probably didn't like the code on the last slide: it calls the NumberCorrect method twice, so it *enumerates the same LINQ sequence twice*. Yuck!

In our little math quiz game, that won't make a noticeable performance difference, but it's still not a great practice. Let's refactor it by adding a variable.

You can use @{ … } to add C# code to your Razor page, and it acts exactly the way you would expect it to—if you declare a variable at the top of the page, it will be in scope further down. If you move the @{ … } section to the bottom of the page, you'll get exactly the compiler error that you would expect.

```
<div class="row mt-5">
    <div class="col-6">
        @{
            var numberCorrect = NumberCorrect();
        }
        @if(numberCorrect < QUESTION_COUNT)
        {
            <h6>
                @numberCorrect correct out of @QUESTION_COUNT
            </h6>
        }
        else
        {
            ...
```

This code calls NumberCorrect just once and stores its results in a variable, which gets used in the @if conditional and the text.

This is the compiler error you get if you move the numberCorrect variable declaration to the bottom of the HTML markup in the Razor component.

CS0841	Cannot use local variable 'numberCorrect' before it is declared	Blazor_Math_Quiz	MathQuiz.razor	32
CS0841	Cannot use local variable 'numberCorrect' before it is declared	Blazor_Math_Quiz	MathQuiz.razor	34

The final math quiz game

A message with the number of correct answers appears here instead of a button until the player gets all the questions right.

Discussion

Let's have a look at some of the Razor syntax that we used, and a few other features of Razor that will be familiar to C# developers.

@if, else if, else, and @switch

- We used @if to display a “Correct!” message if a question was correct, and @if/else to display either a message or a button.
- Blazor @if expressions also support else if. Like else, they don’t require an additional @.
- Blazor @switch statements work just like regular C# switch statements. Razor does <u>not</u> support switch expressions.

```
@switch(r) {
    case 1:
        <h1>One</h1>
        break;
    case 2:
        <h1>Two</h1>
        break;
    case > 4:
        <h1>Greater than 4</h1>
        break;
    default:
        <h1>Three or four</h1>
        break;
}
```

Razor supports foreach, for, and while

- We used a foreach loop to turn a single question into a series of questions. Razor also has @for and @while loops.
- Loops can be nested. Here's an example of nested loops, based on an exercise from the section of ***Head First C#*** about loops.

```
@{
    var forCount = 0;
    var whileCount = 0;
    var p = 2;
}


@for (var q = 2; q < 32; q *= 2)
{
    forCount++;
    @while (p < q)
    {
        whileCount++;
        p *= 2;
    }
    q = p - q;
}

<h3>The for loop executed @forCount times.</h3>
<h3>The while loop executed @whileCount times.</h3>
```

Razor has syntax for exception handling

- Razor @try/catch/finally directives let you execute code that might throw an exception, then handle that exception.
- The syntax is almost identical to normal C# exception handling.
- Learn more about Razor syntax here: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor

```
@try {
    DoSomethingRisky();
}
catch (InvalidOperationException ex) {
    <div class="alert alert-primary" role="alert">
        Something went wrong: @ex.Message
    </div>
} finally {
    <h3>But you need to know something either way...</h3>
}
```

Bootstrap has an alert class that displays a styled alert message.

The Razor component lifecycle

Razor pages go through a lifecycle that calls a series of methods as the page goes through its various states. There are both synchronous and asynchronous versions of those lifecycle methods. You can override those methods so your page can handle those specific events.

The image on this slide comes from the Microsoft documentation on the Razor component lifecycle. You'll notice that it contains the **OnInitialized** – which we used to generate the questions when the page loads:

```
protected override void OnInitialized()
{
    GenerateQuestions();
}
```

You're overriding the OnInitialized method, but there's no need to call base.OnInitialized because it's a virtual empty method—it doesn't do anything.

Image source: https://docs.microsoft.com/en-us/aspnet/core/blazor/components/lifecycle

Start building a bigger project

For the rest of this course, we'll be using Blazor to build a more interactive (and more fun!) game.

The Animal Matching Game

For the rest of this training session, we'll work on a game that's more interactive—and more fun!

When you open the page, you'll see a grid of sixteen animals, eight pairs of them. As soon as you click an animal, a timer will start. Your goal is to click matching pairs of animals. When you click an animal and then click its match, they both disappear. If you click an animal but then click another button with a different animal, both buttons get deselected.

As soon as you click the last pair, the timer stops. Can you beat your fastest time?

Start by cleaning up the project

We want the game to start up as soon as you run the app, so we'll do our work in the Index.razor component.

You won't need the other components, so you can delete Counter.razor and FetchData.razor, as well as the sample data folder for the Fetch Data page.

We also don't need the SurveyPrompt component, so you can delete SurveyPrompt.razor from the Shared folder.

Finally, we won't need other pages (yet), so go ahead and remove the Counter and Fetch Data menu options from the nav menu, just like we did earlier.

This is all stuff we've done already! We'll just do it again for this project.

- Create a new Blazor WebAssembly project called AnimalMatchingGame.
- Delete Pages/Counter.razor, Pages/FetchData.razor, Shared/SurveyPrompt.razor, and the wwwroot/sample-data folder.
- Modify the nav menu to remove the Counter and Fetch Data menu options.
- Change the navigation bar brand to "Animal Matching Game" (on line 3 of Shared/NavMenu.razor).

```
<a class="navbar-brand" href="">Animal Matching Game</a>
```

Add a List of animal emoji to the page's @code

Create a new List<string> that contains the animal emoji for the game. It should contain exactly 16 elements, eight matched pairs of emoji.

If you're using Windows, press ⊞ Win + . (windows key and period) to bring up the Windows emoji keyboard.

If you're using MacOS, choose "Emoji & Symbols" from the input menu menu. If you don't see the input menu, open Preferences enable "Show input menu in menu bar" in the Input Sources section of the Keyboard preferences panel.

The Index.razor page created by the Blazor WebAssembly App template doesn't have a @code section, so you'll need to add it – add it to the end of the file.

```
@code {
    List<string> animalEmoji = new()
    {
        "🐶", "🐶",
        "🐺", "🐺",
        "🐮", "🐮",
        "🦊", "🦊",
        "🐱", "🐱",
        "🦁", "🦁",
        "🐯", "🐯",
        "🐹", "🐹",
    };
}
```

When you use emoji or other Unicode characters from higher code pages, Visual Studio may prompt you to save in a different format. You'll generally want to save using UTF-8 – and you can learn more about what that means in ***Head First C#***.

Use a @foreach loop to add a button for each emoji

Take a close look at the HTML and Razor markup to add a set of buttons, one for each emoji in the animalEmoji list. It should all look familiar—the only thing you haven't seen yet is the btn-outline-dark class, which adds a dark outline to a <button>.

You might notice that there all the <button> controls are contained inside one row. Each button has the "col-3" class, and since Bootstrap rows have 12 columns, four buttons will fit on each row. Bootstrap will automatically wrap the columns to the next row, so you'll end up with four rows of buttons.

```
@page "/"

<PageTitle>Animal Matching Game</PageTitle>

<div class="container">
    <div class="row">
        @foreach (var animal in animalEmoji)
        {
            <div class="col-3">
                <button type="button"
                  class="btn btn-outline-dark">
                    <h1>@animal</h1>
                </button>
            </div>
        }
    </div>
</div>
```

Replace everything above the new @code section that you added with this HTML and Razor markup.

Your app now has 16 animal buttons

Each button is 3 columns wide, so Bootstrap will wrap them into groups of 4 buttons.
The space between the buttons looks a little weird. Can we get rid of it?

Use a <style> tag to include CSS styles in your page

CSS gives you a lot of fine-grained control over your layouts, and Blazor makes it convenient to include CSS in your Blazor page.

The Razor **<style>** tag lets you include CSS that will get added to the page.

We'll use CSS to put the buttons into a uniform grid with no space between them.

If you're familiar with CSS, try experimenting with adding additional styles, like changing the background color (but we won't include this in any of the screenshots in this training deck):

```
body {
    background-color: lightblue;
}
```

```
<style>
    .container {
        width: 400px;
    }
        button {
        width: 100px;
        height: 100px;
        font-size: 50px;
    }
</style>
```

This styles the that surrounds row to make it 400 pixels wide. CSS pixels are relative to the viewing device, so they're 1/96th of an inch (unscaled).

This styles every <button> on the page, making each button a 100 pixel square. It also makes the font size 50px to change the size of the animal picture in the button. Try playing with the font size to see what you think looks best.

Now the buttons are in a nice grid

This is the starting point for the game.

What do you think we'll do next?

Q&A

If you have any questions, enter them in the Q&A widget.

Go to https://bit.ly/blazor-training for all the code that you've seen so far.

Exercise

This is your chance to start the building the Animal Matching Game app on your own, or you can also try the Math Quiz game. You can download the code for both of them from the URL above.

5 minute break

Take a quick breather, have a stretch… we'll be back here in five minutes.

Part 3
Continuing the animal matching game

We'll keep rolling with the animal matching game, introducing more features of Blazor that will help you build more interactive web apps.

Handling mouse clicks

We've already seen how to handle mouse clicks with the @onclick event. Let's take a deeper dive into it.

Add code to shuffle the animals

The game won't be very fun if the animals to match are always next to each other, so let's add code to shuffle the emoji—and we can do this all using tools we've already seen today.

We'll do the following:

- Add @using System.Linq so we can use LINQ methods.
- Add a new collection called shuffledAnimals.
- Create a SetUpGame method that shuffles the emoji and stores the list of shuffled emoji in shuffledAnimals.
- Override OnInitialized so it calls the new SetUpGame method.

```
@using System.Linq

@code {

    List<string> shuffledAnimals = new();

    private void SetUpGame()
    {
        shuffledAnimals = animalEmoji
            .OrderBy(item => Random.Shared.Next())
            .ToList();
    }

    protected override void OnInitialized()
    {
        SetUpGame();
    }
```

You used this @using directive earlier.

Add the shuffledAnimals field and the SetUpGame and OnInitialized methods to the @code section.

We're taking advantage of the Razor component lifecycle to set up the game when the page loads.

Modify the @foreach loop to use the shuffled emoji

Now we just need to modify the @foreach so it iterates through the collection of shuffled animals:

```
@foreach (var animal in shuffledAnimals)
```

Run the program again—now the animal buttons are shuffled. Refresh the page a few times to see the app re-shuffle them.

How mouse clicks will work

1. The player clicks the first button.
The player clicks buttons in pairs. When they click the first button, the game keeps track of that particular button's animal.

2. The player clicks the second button.
The game looks at the animal on the second button and compares it against the one that it kept track of from the first click.

3. The game checks for a match.
If the animals *match*, the game goes through all of the emoji in its list of shuffled animal emoji. It finds any emoji in the list that match the animal pair the player found and replaces them with blanks. If the animals *don't match*, the game doesn't do anything.
In *either case*, it resets its last animal found so it can do the whole thing over for the next click.

Add a click event handler method

This is the **most complex** **piece of code** in the entire app. We added `comments` explaining how it works.

Each button has an animal emoji, but we need more than the emoji to identify a button—otherwise the user could just double-click on a button and the app would see it as a “match” (which would be a bug!).

If the lastAnimalFound field is empty, the player is making their first click, so it stores the emoji and description of the button in the fields. If it isn’t, then either the player clicked a match, in which case the app replaces the matching emoji in the collection with empty strings, otherwise the method just resets lastAnimalFound.

```
string lastAnimalFound = string.Empty;
string lastDescription = string.Empty;
private void ButtonClick(string animal, string animalDescription)
{
    if (lastAnimalFound == string.Empty)
    {
        // First selection of the pair. Remember it.
        lastAnimalFound = animal;
        lastDescription = animalDescription;
    }
    else if ((lastAnimalFound == animal)
                && (animalDescription != lastDescription))
    {
        // Match found! Reset for next pair.
        lastAnimalFound = string.Empty;


        // Replace found animals with empty string to hide them.
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();
    }
    else
    {
        // User selected a pair that don't match, reset selection.
        lastAnimalFound = string.Empty;
    }
}
```

Update the loop to add an @onclick event to each button

We need to add a unique description to each button, so the ButtonClick method can differentiate two buttons with the same animal emoji.

To do this, we'll change the @for loop into a @foreach. Then we'll **declare two variables** inside the foreach loop: animal is the emoji, and uniqueDescription is a string (e.g. "Button #4") that's unique to each button.

Take a close look at the @onclick event. It needs to call a method that takes parameters that don't match the standard event handler arguments. To do this, we'll use a lambda to define an anonymous method that calls ButtonClick with the parameters we want.

```
<div class="container">
    <div class="row">
        @for (var animalNumber = 0;
              animalNumber < shuffledAnimals.Count;
              animalNumber++)
        {
            var animal = shuffledAnimals[animalNumber];
            var uniqueDescription =
                   $"Button #{animalNumber}";
            <div class="col-3">
                <button type="button"
                @onclick="@(() =>
                    ButtonClick(animal, uniqueDescription))"
                class="btn btn-outline-dark">
                    <h1>@animal</h1>
                </button>
            </div>
        }
    </div>
</div>
```

We're using a lambda to define an anonymous method that calls ButtonClick with the correct parameters.

Now when you click on a pair of emoji, the app blanks out their buttons.

Keeping track of matches found

Let's recap what we've done so far by adding a row to the bottom of the page that displays the number of matches found, and resets the game when the player found all of the matches.

Keep track of the player's progress

The game will display the number of matches the player found so far underneath the buttons, and reshuffle and reset the emoji once all the matches have been found.

Add a field to keep track of the number of matches found

We need to keep track of the number of matches found, so we'll need to store it somewhere. A field will do nicely, so we'll add a matchesFound field.

We also know that it needs to reset when the game gets reset, and we'll be resetting the game by calling SetUpGame(), so we'll make sure to reset it in that method.

Nothing in this section is new, it's all a recap to help help these ideas sink in.

```
List<string> shuffledAnimals = new();
int matchesFound = 0;

private void SetUpGame()
{
    shuffledAnimals = animalEmoji
        .OrderBy(item => Random.Shared.Next())
        .ToList();
    matchesFound = 0;
}
```

This should be getting a little familiar, maybe even repetitive—and that's good! It means you're starting to get the ideas behind how Blazor works.

Modify ButtonClick to update the matchesFound field

The @onclick event handler is a lambda that calls the ButtonClick method, which checks for matches and hides them when they're found.

The "else if" section of that method is called every time a match is found. Let's modify it to increment the matchesFound field, then reset the game by calling the SetUpGame method if all eight matches have been found.

```
private void ButtonClick(string animal, string animalDescription)
{
    if (lastAnimalFound == string.Empty)
    {
        // First selection of the pair. Remember it.
        lastAnimalFound = animal;
        lastDescription = animalDescription;
    }
    else if ((lastAnimalFound == animal)
             && (animalDescription != lastDescription))
    {
        // Match found! Reset for next pair.
        lastAnimalFound = string.Empty;

        // Replace found animals with empty string to hide them.
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();

        matchesFound++;
        if (matchesFound == 8)
        {
            SetUpGame();
        }
    }
    else
    {
        // User selected a pair that don't match.
        // Reset selection.
        lastAnimalFound = string.Empty;
    }
}
```

Display the number of matches found

Now we're keeping track of the number of matches found and resetting the game if all eight matches were found. Since we're using a field to track that number, we can display it to the user to give them more information about their progress.

We can do that by adding a row to display @matchesFound at the bottom of the page.

```
<div class="container">
    <div class="row">
        @for (var animalNumber = 0;
              animalNumber < shuffledAnimals.Count;
              animalNumber++)
        {
            var animal = shuffledAnimals[animalNumber];
            var uniqueDescription =
                   $"Button #{animalNumber}";
            <div class="col-3">
                <button type="button"
                @onclick="@(() =>
                    ButtonClick(animal, uniqueDescription))"
                class="btn btn-outline-dark">
                    <h1>@animal</h1>
                </button>
            </div>
        }
    </div>
    <div class="row">
        <h2>Matches found: @matchesFound</h2>
    </div>
</div>
```

Now the game shows the number of matches found

Discussion

Let's take a minute and review the ideas we've covered so far—and give all of these ideas a chance to really sink in.

Our @onclick event used a lambda

- We used a lambda expression to handle the @onclick event:

```
@onclick="@(() => ButtonClick(animal, uniqueDescription))"
```

- The reason we did it that way was because every time the player clicks an animal button, it needs to call ButtonClick with arguments that tell it the animal and unique description, so it's able to figure out what animal was clicked.
- If you hover over @onclick in Visual Studio, you can see the definition:

> EventCallback<MouseEventArgs> Microsoft.AspNetCore.Components.Web.EventHandlers.onclick
> Sets the '@onclick' attribute to the provided string or delegate value. A delegate value should be of type 'Microsoft.AspNetCore.Components.Web.MouseEventArgs'.

- It uses an EventCallback, which just takes a single MouseEventArgs parameter. The lambda lets us pass our own parameters—and it has the additional benefit of being pretty readable.

Using callbacks after initialization

In a lot of web applications, something happens in the background—like an action happening on a timer, or a message coming in from another system—that require a callback.

When you need to this callback can be invoked any time after the component is initialized, it must use the combination of InvokeAsync and StateHasChanged.

InvokeAsync

Executes the supplied work item on the associated renderer's synchronization context. The task to be executed needs to be passed to it.

StateHasChanged

Notifies the component that its state has changed. When applicable, this will cause the component to be re-rendered. It needs to be called inside the task after all of the updates are complete.

```
@page "/"

<h1>@randomNumber</h1>

@using System.Timers
@code {
    Timer timer = new Timer(1000);
    int randomNumber = Random.Shared.Next();


    protected override void OnInitialized() {
        timer.Elapsed += Timer_Elapsed;
        timer.Start();
    }


    private void Timer_Elapsed(object? source,
                               ElapsedEventArgs e) {
        InvokeAsync(() => {
            randomNumber = Random.Shared.Next();
            StateHasChanged();
        });
    }
}
```

Here's a simple page that uses a System.Timer to display a random number that updates every second.

The Timer's updates happen on another thread, so it needs to use InvokeAsync/StateHasChanged.

Add a timer

The game is pretty good. Adding a timer will make it more exciting. It puts the pressure on players to go faster and beat their best times.

Add fields to use a Timer and keep track of the time

We'll be using a System.Timers.Timer to add the time elapsed. It's not the most elegant solution: we'll set it to fire every tenth of a second and increment a field called tenthsOfSecondsElapsed in the event handler. But it will get the job done.

We'll start by adding a row to the page, and adding fields for the timer, elapsed time, and a user-friendly display that we can show at the bottom of the page.

```
<div class="row">
    <h2>Matches found: @matchesFound</h2>
</div>
<div class="row">
    <h2>Time: @timeDisplay</h2>
</div>
```

This row displays a user-friendly time display, which is kept in the timeDisplay field.

You'll need fields for the Timer, the time elapsed, and the time display.

```
@using System.Linq
@using System.Timers

@code {

    Timer timer = new Timer(100);
    int tenthsOfSecondsElapsed = 0;
    string timeDisplay = "";
```

Reset the timer when the game starts

The SetUpGame method sets up the game, so this is a great place to reset the timer. We'll do that by creating a new instance of System.Timer and resetting the field that stores the tenths of seconds that have elapsed since the player clicked the first animal.

```
private void SetUpGame()
{
    shuffledAnimals = animalEmoji
        .OrderBy(item => Random.Shared.Next())
        .ToList();
    matchesFound = 0;
    timer = new Timer(100);
    timer.Elapsed += Timer_Elapsed;
    tenthsOfSecondsElapsed = 0;
}
```

Make the timer run when the player is finding matches

The ButtonClick method contains a single if/else if/else conditional. We'll add a few lines to it so the timer starts when the player clicks the first animal, and ends when they get the last match

The if clause is execution when the player makes their first click, so that's a good time to start the timer. It's actually executed every time the player clicks on the first animal, but it's okay to keep calling the timer's Start method even if it's running.

The else if clause has a nested if statement that checks if the player found all eight pairs. This is a good place to stop the timer and add *"Play Again?"* to the time display.

All in all, we're just adding **three lines** to the ButtonClick method.

```
private void ButtonClick(string animal, string animalDescription)
{
    if (lastAnimalFound == string.Empty)
    {
        // First selection of the pair. Remember it.
        lastAnimalFound = animal;
        lastDescription = animalDescription;
        timer.Start();
    }
    else if ((lastAnimalFound == animal) && (animalDescription != l
    {
        // Match found! Reset for next pair.
        lastAnimalFound = string.Empty;


        // Replace found animals with empty string to hide them.
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();


        matchesFound++;
        if (matchesFound == 8)
        {
            timer.Stop();
            timeDisplay += " - Play Again?";
            SetUpGame();
        }
    }
    else
    {
        // User selected a pair that don't match.
        // Reset selection.
        lastAnimalFound = string.Empty;
    }
}
```

Update the time display when the timer ticks

The game implements a pretty crude (and not 100% accurate!) timer by setting it to tick every 1/10th of a second and incrementing a count of tenths of seconds elapsed. .NET timers are guaranteed to tick no more frequently than their interval, but if there's a lot of load on the client it's possible that the ticks could be slightly more than 1/10th of a second apart. But that's okay for this game.

Since System.Timer calls its Tick event handler on a separate thread, we need to use InvokeAsync to make sure the changes happen in the correct context. Once the bound variable, tenthsOfSecondsElapsed, is updated, we call StateHasChanged to tell the page a variable has changed and it can re-render.

We're running this on a separate thread, so we need to call InvokeAsync so it renders in the right context.

```
private void Timer_Elapsed(object? source,
                           ElapsedEventArgs e)
{
    InvokeAsync(
        () =>
        {
            tenthsOfSecondsElapsed++;
            timeDisplay =
                   (tenthsOfSecondsElapsed / 10F)
                       .ToString("0.0s");
            StateHasChanged();
        }
    );
}
```

Once the update has completed, we need to calls StateHasChanged to notify that the component's state has changed so it can re-render.

Q&A

If you have any questions, enter them in the Q&A widget.

Go to https://bit.ly/blazor-training for all the code that you've seen so far.

Exercise

It's okay if you don't finish right now—the most important thing is getting started while this is still fresh in your mind, that's the best way to get these new concepts to stick.

5 minute break

Take a bio break, give the dog a quick walk… we'll be back here in five minutes.

Part 4
Go full stack with .NET minimal APIs

Full-stack web applications connect to services on the back end, and .NET minimal APIs are a great way to build them.

Create an HTTP API

Everything on your app has been running entirely in the web browser, just like a JavaScript app. Now let's get some code running on the server.

Create a new ASP.NET Core Web API project

We'll start by creating an **ASP.NET Core Web API** project, because it includes all the dependencies that we need to create our HTTP API.

We'll name the project My_Web_API.

ASP.NET Core Web App projects can also be created from the command line:

```
dotnet new webapi -o My_Web_API
```

(Note: make sure you're creating an ASP.NET Core Web API project, and not an ASP.NET Core Web APP project. You can use that kind of project to create a minimal .NET API too, but the code in Program.cs won't match these slides.)

This is not a course in ASP.NET. We're going to give you just enough code and explanation to get started with .NET minimal APIs.

Running the API opens a Swagger UI in a browser

Web APIs don't have a user interface, which is why the .NET template for web API projects comes with Swagger, which provides a web page that describes your APIs and lets you execute them.

When you created your project, the template included these lines in Program.cs to add Swagger:

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

...

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

You can learn more about Swagger from its website: https://swagger.io/

Add a “Hello, world!” endpoint

Close your browser to stop the web app (if you configured Visual Studio to keep the app running after the browser is closed, stop the app).

Open Program.cs and find the last line:

```
app.Run();
```

This line runs the web API app. Add this statement just before that last line:

```
app.MapGet("/hello",
      () => "Hello world!");
```

Now run your app again. You’ll see a section for your new /hello API endpoint. Expand it and click Execute to execute a request and see the response.

Your APIs can use classes, and you can debug the code

Here's an API endpoint /random that returns a random number. We put a breakpoint on the line of code that generates the random number.

If you put a breakpoint on the code, then when you run the app, expand the /random endpoint in the Swagger UI, and click Execute, Visual Studio will break on that line.

If you disable the breakpoint, then go back to the Swagger UI and hit Execute a few times, you'll see the random number update.

You can access the API directly in the browser: replace /swagger/index.html with /random in the URL.

Your APIs can take parameters

The second parameter to the MapGet query is a System.Delegate, which means it can take parameters. And the way it works is pretty intuitive.

Here's a simple API endpoint that takes an integer value and multiplies it by three:

```
app.MapGet("/times_three",
     (int value) => value * 3);
```

We've been using lambdas, but you can also pass delegates into MapGet:

```
int TimesThree(int value)
{
    return value * 3;
}
app.MapGet("/times_three", TimesThree);
```

Discussion

We just did a LOT of server-side development without writing much code at all. Let's take a closer look at what's going on.

.NET Minimal APIs make it easy to create API endpoints

- We've been using the MapGet method to add entry points to your web applications. It's useful to see how it works:

(extension) RouteHandlerBuilder IEndpointRouteBuilder.MapGet(string pattern, Delegate handler) (+ 1 overload)
Adds a RouteEndpoint to the IEndpointRouteBuilder that matches HTTP GET requests for the specified pattern.

Returns:
A RouteHandlerBuilder that can be used to further customize the endpoint.

- It takes two arguments: a pattern and a Delegate. If you pass it an object, it will generally do a good job of converting it to JSON:

```
app.MapGet("/getobject", () =>
    new { Value = 123, GhostSays = "Boo", Cow = "🐄" });
```

The ASP.NET Program.cs isn't magic

- We're won't do a deep dive into how an ASP.NET app is set up, but it's useful to know at least the basic structure of its entry point, so it doesn't feel intimidating.
- It starts by creating a WebApplicationBuilder:

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

- Once services like Swagger are added to that builder, it creates the WebApplication:

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

- The WebApplication class has methods like MapGet that let you add APIs, and properties like Environment that let you check if it's a development or production environment.
- Once the app is set up, calling `app.Run()` Starts the web application.

Create a full-stack Blazor Server app for Blazor WebAssembly and .NET APIs

- In the last part of this training session, we're going to combine everything we've done so far into one app.
- We'll create a Blazor Server app that combines a Blazor front end with APIs provided by an ASP.NET back end.
- We'll be adding APIs, so we're going to **add Swagger** so we can test them and make sure they work.
- Your Razor component will need to make HTTP requests to access those APIs, so we'll **inject an HTTP client** that it can use.

Let’s turn our game into a full-stack Blazor app

We’re going to tie put everything we’ve done so far together by updating our Blazor animal matching game to make an API call to track the best time.

Create a new Blazor Server App

Create a new Blazor Server App project

Name it AnimalMatchingGame_Server

Blazor Server App projects can also be created from the command line:

```
dotnet new blazorserver -o
AnimalMatchingGame_Server
```

Run the game – it should look very familiar! It's the same Blazor app you've seen throughout this whole training that has the familiar index page with the navigation links to same "Counter" and "Fetch data" components that you're used to.

Copy the Index.razor component and paste it into your app

Expand the Pages folder – you'll see a few extra files (_Host.cshtml, _Layout.cshtml, and Error.cshtml), but you'll also see the familiar Razor components. Go ahead and make the same changes you made in Part 2:

- Delete Counter.razor and FetchData from the Pages folder.
- Delete SurveyPrompt.razor from the Shared folder.
- Modify Shared/NavMenu.razor to remove the links to Counter and Fetch Data and change the header to "Animal Matching Game"

Switch back to your AnimalMatchingGame project, open Index.razor, and **copy the entire file** into the buffer. Then go back to your new project and **paste over the contents of Index.razor**.

Now run your game – all the code runs on the client and is in that component, so it still works!

Add Swagger to your ASP.NET Core Blazor Server app

Since we're going to be working with APIs, we're going to want to have Swagger installed. Luckily, it's easy to add to our Blazor Server app: we need to install a NuGet packages and add a few lines to Program.cs.

Let's also add the "Hello, world!" endpoint so we can make sure Swagger is working:

```
app.MapGet("/hello",
      () => "Hello world!");
```

First add the **Swashbuckle.AspNetCore** NuGet package. Then you can add the following lines to your Program.cs just above and below *var app = builder.Build()*:

```
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Create an HTTP client and inject it into your Blazor app

Go back to your Program.cs file, find the line where you create the WebApplication, and add the code on the right to create a new **HttpClient service**. This is will make an HTTP client available to your web application.

Then go to your Index.razor component and add the following line just above the @code section:

```
@inject HttpClient Http
```

Now you'll be able to use the Http object in your C# code in the component to make HTTP requests.

```
builder.Services.AddHttpClient();

var baseUrls = builder.WebHost
       .GetSetting(WebHostDefaults.ServerUrlsKey);
var baseUrl = (baseUrls ?? "").Split(";").First();

builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri(baseUrl)
    });

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

Add a class to track the best score

Add the ScoreTracker class to your project. It's a really simple class with properties to keep track of the lowest score, and a message to display if the player got the best time.

If you use Visual Studio's "Add Class" feature, it will add a namespace declaration – remove it and add ScoreTracker to **the <u>top-level</u> namespace** (so your code can match our code).

Once you've added ScoreTracker, update Program.cs to add the API:

```
public static class ScoreTracker
{
    public static float LowestScore { get; set; } = 0;

    public static string SubmitScore(float score)
    {
        if ((LowestScore == 0) || (score < LowestScore))
        {
            LowestScore = score;
            return "You got the best time!";
        }
        else
            return string.Empty;
    }
}
```

Since MapGet takes a Delegate as its second parameter, you can just pass it a reference to the static SubmitScore method directly from the ScoreTracker class.

```
app.MapGet("/scoretracker", ScoreTracker.SubmitScore);
```

Use Swagger to test your new API

- Run your app, then go to the Swagger UI and test your API. Try putting a breakpoint inside ScoreTracker to see what happens.

GET /scoretracker

Parameters

Cancel

The first time you submit an API request the breakpoint should trigger.

Name	Description
score * required number($float) (query)	14.3

```
1 reference
public static string SubmitScore(float score)
{
    if ((LowestScore == 0) || (score < LowestScore))
    {
        LowestScore = score;
        return "You got the best time!";
    }
    else
        return string.Empty;
}
```

LowestScore 14.3

It set the LowestScore property and returns a message, which you'll see in the response body when you continue.

Response body

```
You got the best time!
```

Download

Update Index.razor to display the message and start updating the code

Here's the code to add just above the top Bootstrap row in your HTML.

```
@if(!string.IsNullOrEmpty(bestTimeMessage))
{
    <h1>@bestTimeMessage</h1>
}
@if (ScoreTracker.LowestScore > 0)
{
    <h3>Best score: @ScoreTracker.LowestScore</h3>
}
```

ScoreTracker is static and in the top-level namespace, so it's already in scope and you don't need a property for it.

And you'll also need a field to bind @bestTimeMessage to.

```
@using System.Linq
@using System.Timers

@inject HttpClient Http


@code {

    string bestTimeMessage = string.Empty;
```

Make your button's @onclick event handler call the API

- We're going to make three changes to the ButtonClick event handler and the @onclick event that calls it. First we'll change the declaration to make it asynchronous:

```
private async Task ButtonClick(string animal, string animalDescription)
```

- We need to reset the bestTimeMessage field

```
// First selection of the pair. Remember it.
bestTimeMessage = string.Empty;
```

- And finally, we'll actually call the API. The HTTP client's GetStringAsync method calls a GET API – we'll pass it the player's time once they get all the matches:

```
if (matchesFound == 8)
{
    bestTimeMessage = await Http.GetStringAsync($"/scoretracker?score={tenthsOfSecondsElapsed / 10F}");
```

This is a GET request, so we can just add the score to the parameter list, in this case using string interpolation.

Make your button's @onclick event call an asynchronous method

- We made the ButtonClick method asynchronous, so we'll need to use the async and await keywords in the @onclick event as well:

```
@onclick="@(async () => await ButtonClick(animal, uniqueDescription))"
```

- And that's it! We've added an API endpoint, tested it with Swagger, injected the HTTP client into the Razor page, and updated it to call the API endpoint to save data.

Now your game remembers the fastest time.

Run your game again – the first time you play, you'll get the best time, and it will show the "You got the best time!" message returned by the endpoint and show the best score.

If you refresh the page, it will still remember the best score. Open two or three other browser windows—they'll all show the same best score. If you beat it, then when you refresh the other pages they'll get the updated score from the server.

Congratulations, you've now built a full-stack Blazor app!

Q&A

If you have any questions, enter them in the Q&A widget.

Additional resources

The goal of this course was to give you a "just enough to be dangerous" introduction to Blazor. Want to learn more? Here are some great places to start.

- David Pine's excellent book, ***Learning Blazor*** (O'Reilly 2022): https://learning.oreilly.com/library/view/learning-blazor/9781098113230/
- The animal matching game, quiz game, and other code in this course were based on my book, ***Head First C#*** (O'Reilly 2021) – and there's more explanation of how the game works, if you want to delve deeper into it: https://learning.oreilly.com/library/view/head-first-c/9781491976692/
- Microsoft has excellent Blazor documentation. Here's a starting point: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0
- Finally, these slides, all the code I showed you, and links to other materials can be found on the GitHub page for the course: https://bit.ly/blazor-training

Thank you!

I hope you enjoyed my course! If you want to learn more, follow me on Twitter (@AndrewStellman), and check out *Head First C#* and my other books on O'Reilly Learning.

Thanks so much for attending!