If JS is loaded from a JavaScript module, then `[JSImport]` attributes must include the module name as the second parameter. For example, `[JSImport("globalThis.callAlert", "ExampleShim")]` indicates the imported method was declared in a JavaScript module named "`ExampleShim`."

Type mappings

Parameters and return types in the .NET method signature are automatically converted to or from appropriate JS types at runtime if a unique mapping is supported. This may result in values converted by value or references wrapped in a proxy type. This process is known as *type marshalling*. Use JSMarshalAsAttribute<T> to control how the imported method parameters and return types are marshalled.

Some types don't have a default type mapping. For example, a `long` can be marshalled as System.Runtime.InteropServices.JavaScript.JSType.Number or System.Runtime.InteropServices.JavaScript.JSType.BigInt, so the JSMarshalAsAttribute<T> is required to avoid a compile-time error.

The following type mapping scenarios are supported:

- Passing Action or Func<TResult> as parameters, which are are marshalled as callable JS methods. This allows .NET code to invoke listeners in response to JS callbacks or events.
- Passing JS references and .NET managed object references in either direction, which as marshaled as proxy objects and kept alive across the interop boundary until the proxy is garbage collected.
- Marshalling asynchronous JS methods or a JS Promise with a Task result, and vice versa.

Most of the marshalled types work in both directions, as parameters and as return values, on both imported and exported methods.

The following table indicates the supported type mappings.

Expand table

.NET	JavaScript	`Nullable`	`Task` ➔ `Promise`	`JSMarshalAs` optional	`Array of`
`Boolean`	`Boolean`	✓	✓	✓	
`Byte`	`Number`	✓	✓	✓	✓

.NET	JavaScript	`Nullable`	`Task` ➔ `Promise`	`JSMarshalAs` optional	`Array of`
`Char`	`String`	✓	✓	✓	
`Int16`	`Number`	✓	✓	✓	
`Int32`	`Number`	✓	✓	✓	✓
`Int64`	`Number`	✓	✓		
`Int64`	`BigInt`	✓	✓		
`Single`	`Number`	✓	✓	✓	
`Double`	`Number`	✓	✓	✓	✓
`IntPtr`	`Number`	✓	✓	✓	
`DateTime`	`Date`	✓	✓		
`DateTimeOffset`	`Date`	✓	✓		
`Exception`	`Error`		✓	✓	
`JSObject`	`Object`		✓	✓	✓
`String`	`String`		✓	✓	✓
`Object`	`Any`		✓		✓
`Span<Byte>`	`MemoryView`				
`Span<Int32>`	`MemoryView`				
`Span<Double>`	`MemoryView`				
`ArraySegment<Byte>`	`MemoryView`				
`ArraySegment<Int32>`	`MemoryView`				
`ArraySegment<Double>`	`MemoryView`				
`Task`	`Promise`			✓	
`Action`	`Function`				
`Action<T1>`	`Function`				
`Action<T1, T2>`	`Function`				
`Action<T1, T2, T3>`	`Function`				

.NET	JavaScript	`Nullable`	`Task` ➔ `Promise`	`JSMarshalAs` optional	`Array of`
`Func<TResult>`	`Function`				
`Func<T1, TResult>`	`Function`				
`Func<T1, T2, TResult>`	`Function`				
`Func<T1, T2, T3, TResult>`	`Function`				

The following conditions apply to type mapping and marshalled values:

- The `Array of` column indicates if the .NET type can be marshalled as a JS Array. Example: C# `int[]` (`Int32`) mapped to JS `Array` of `Number`s.
- When passing a JS value to C# with a value of the wrong type, the framework throws an exception in most cases. The framework doesn't perform compile-time type checking in JS.
- `JSObject`, `Exception`, `Task` and `ArraySegment` create `GCHandle` and a proxy. You can trigger disposal in developer code or allow .NET garbage collection (GC) to dispose of the objects later. These types carry significant performance overhead.
- `Array`: Marshaling an array creates a copy of the array in JS or .NET.
- `MemoryView`
  - `MemoryView` is a JS class for the .NET WebAssembly runtime to marshal `Span` and `ArraySegment`.
  - Unlike marshaling an array, marshaling a `Span` or `ArraySegment` doesn't create a copy of the underlying memory.
  - `MemoryView` can only be properly instantiated by the .NET WebAssembly runtime. Therefore, it isn't possible to import a JS method as a .NET method that has a parameter of `Span` or `ArraySegment`.
  - `MemoryView` created for a `Span` is only valid for the duration of the interop call. As `Span` is allocated on the call stack, which doesn't persist after the interop call, it isn't possible to export a .NET method that returns a `Span`.
  - `MemoryView` created for an `ArraySegment` survives after the interop call and is useful for sharing a buffer. Calling `dispose()` on a `MemoryView` created for an `ArraySegment` disposes the proxy and unpins the underlying .NET array. We recommend calling `dispose()` in a `try-finally` block for `MemoryView`.

Some combinations of type mappings that require nested generic types in JSMarshalAs aren't currently supported. For example, attempting to materialize an array from a Promise such as `[return: JSMarshalAs<JSType.Promise<JSType.Array<JSType.Number>>>`

`()]` generates a compile-time error. An appropriate workaround varies depending on the scenario, but this specific scenario is further explored in the Type mapping limitations section.

JS primitives

The following example demonstrates `[JSImport]` leveraging type mappings of several primitive JS types and the use of JSMarshalAs, where explicit mappings are required at compile time.

`PrimitivesShim.js`:

JavaScript

```javascript
globalThis.counter = 0;

// Takes no parameters and returns nothing.
export function incrementCounter() {
  globalThis.counter += 1;
};

// Returns an int.
export function getCounter() { return globalThis.counter; };

// Takes a parameter and returns nothing. JS doesn't restrict the parameter
type,
// but we can restrict it in the .NET proxy, if desired.
export function logValue(value) { console.log(value); };

// Called for various .NET types to demonstrate mapping to JS primitive
types.
export function logValueAndType(value) { console.log(typeof value, value);
};
```

`PrimitivesInterop.cs`:

C#

```csharp
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class PrimitivesInterop
{
    // Importing an existing JS method.
    [JSImport("globalThis.console.log")]
    public static partial void ConsoleLog([JSMarshalAs<JSType.Any>] object
value);
```

```
    // Importing static methods from a JS module.
    [JSImport("incrementCounter", "PrimitivesShim")]
    public static partial void IncrementCounter();

    [JSImport("getCounter", "PrimitivesShim")]
    public static partial int GetCounter();

    // The JS shim method name isn't required to match the C# method name.
    [JSImport("logValue", "PrimitivesShim")]
    public static partial void LogInt(int value);

    // A second mapping to the same JS method with compatible type.
    [JSImport("logValue", "PrimitivesShim")]
    public static partial void LogString(string value);

    // Accept any type as parameter. .NET types are mapped to JS types where
    // possible. Otherwise, they're marshalled as an untyped object
reference
    // to the .NET object proxy. The JS implementation logs to browser
console
    // the JS type and value to demonstrate results of marshalling.
    [JSImport("logValueAndType", "PrimitivesShim")]
    public static partial void LogValueAndType(
        [JSMarshalAs<JSType.Any>] object value);

    // Some types have multiple mappings and require explicit marshalling to
the
    // desired JS type. A long/Int64 can be mapped as either a Number or
BigInt.
    // Passing a long value to the above method generates an error at
runtime:
    // "ToJS for System.Int64 is not implemented." ("ToJS" means "to
JavaScript")
    // If the parameter declaration `Method(JSMarshalAs<JSType.Any>] long
value)`
    // is used, a compile-time error is generated:
    // "Type long is not supported by source-generated JS interop...."
    // Instead, explicitly map the long parameter to either a JSType.Number
or
    // JSType.BigInt. Note that runtime overflow errors are possible in JS
if the
    // C# value is too large.
    [JSImport("logValueAndType", "PrimitivesShim")]
    public static partial void LogValueAndTypeForNumber(
        [JSMarshalAs<JSType.Number>] long value);

    [JSImport("logValueAndType", "PrimitivesShim")]
    public static partial void LogValueAndTypeForBigInt(
        [JSMarshalAs<JSType.BigInt>] long value);
}

public static class PrimitivesUsage
{
    public static async Task Run()
    {
        // Ensure JS module loaded.
        await JSHost.ImportAsync("PrimitivesShim", "/PrimitivesShim.js");

        // Call a proxy to a static JS method, console.log().
        PrimitivesInterop.ConsoleLog("Printed from JSImport of 
console.log()");

        // Basic examples of JS interop with an integer.
        PrimitivesInterop.IncrementCounter();
        int counterValue = PrimitivesInterop.GetCounter();
        PrimitivesInterop.LogInt(counterValue);
        PrimitivesInterop.LogString("I'm a string from .NET in your 
browser!");

        // Mapping some other .NET types to JS primitives.
        PrimitivesInterop.LogValueAndType(true);
        PrimitivesInterop.LogValueAndType(0x3A); // Byte literal
        PrimitivesInterop.LogValueAndType('C');
        PrimitivesInterop.LogValueAndType((Int16)12);
        // JS Number has a lower max value and can generate overflow errors.
        PrimitivesInterop.LogValueAndTypeForNumber(9007199254740990L); // 
Int64/Long
        // Next line: Int64/Long, JS BigInt supports larger numbers.
        PrimitivesInterop.LogValueAndTypeForBigInt(1234567890123456789L);// 
        PrimitivesInterop.LogValueAndType(3.14f); // Single floating point 
literal
        PrimitivesInterop.LogValueAndType(3.14d); // Double floating point 
literal
        PrimitivesInterop.LogValueAndType("A string");
    }
}
```

In `Program.Main`:

```C#
await PrimitivesUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

```
Printed from JSImport of console.log()
1
I'm a string from .NET in your browser!
boolean true
number 58
number 67
number 12
number 9007199254740990
bigint 1234567890123456789n
```

```
number 3.140000104904175
number 3.14
string A string
```

JS `Date` objects

The example in this section demonstrates importing methods which have a JS Date object as its return or parameter. Dates are marshalled across interop by-value, meaning they are copied in much the same way as JS primitives.

A `Date` object is timezone agnostic. A .NET DateTime is adjusted relative to its DateTimeKind when marshalled to a `Date`, but timezone information isn't preserved. Consider initializing a DateTime with a DateTimeKind.Utc or DateTimeKind.Local consistent with the value it represents.

`DateShim.js`:

JavaScript

```javascript
export function incrementDay(date) {
  date.setDate(date.getDate() + 1);
  return date;
}

export function logValueAndType(value) {
  console.log("Date:", value)
}
```

`DateInterop.cs`:

C#

```csharp
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class DateInterop
{
    [JSImport("incrementDay", "DateShim")]
    [return: JSMarshalAs<JSType.Date>] // Explicit JSMarshalAs for a return
type
    public static partial DateTime IncrementDay(
        [JSMarshalAs<JSType.Date>] DateTime date);

    [JSImport("logValueAndType", "DateShim")]
    public static partial void LogValueAndType(
        [JSMarshalAs<JSType.Date>] DateTime value);
}

public static class DateUsage
{
    public static async Task Run()
    {
        // Ensure JS module loaded.
        await JSHost.ImportAsync("DateShim", "/DateShim.js");

        // Basic examples of interop with a C# DateTime and JS Date.
        DateTime date = new(1968, 12, 21, 12, 51, 0, DateTimeKind.Utc);
        DateInterop.LogValueAndType(date);
        date = DateInterop.IncrementDay(date);
        DateInterop.LogValueAndType(date);
    }
}
```

In `Program.Main`:

```C#
await DateUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> Date: Sat Dec 21 1968 07:51:00 GMT-0500 (Eastern Standard Time)
> Date: Sun Dec 22 1968 07:51:00 GMT-0500 (Eastern Standard Time)

The preceding timezone information (`GMT-0500 (Eastern Standard Time)`) depends on local timezone of your computer/browser.

JS object references

Whenever a JS method returns an object reference, it's represented in .NET as a JSObject. The original JS object continues its lifetime within the JS boundary, while .NET code can access and modify it by reference through the JSObject. While the type itself exposes a limited API, the ability to hold a JS object reference and return or pass it across the interop boundary enables support for several interop scenarios.

The JSObject provides methods to access properties, but it doesn't provide direct access to instance methods. As the following `Summarize` method demonstrates, instance methods can be accessed indirectly by implementing a static method that takes the instance as a parameter.

`JSObjectShim.js`:

JavaScript

```javascript
export function createObject() {
  return {
    name: "Example JS Object",
    answer: 41,
    question: null,
    summarize: function () {
      return `Question: "${this.question}" Answer: ${this.answer}`;
    }
  };
}

export function incrementAnswer(object) {
  object.answer += 1;
  // Don't return the modified object, since the reference is modified.
}

// Proxy an instance method call.
export function summarize(object) {
  return object.summarize();
}
```

`JSObjectInterop.cs`:

C#

```csharp
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class JSObjectInterop
{
    [JSImport("createObject", "JSObjectShim")]
    public static partial JSObject CreateObject();

    [JSImport("incrementAnswer", "JSObjectShim")]
    public static partial void IncrementAnswer(JSObject jsObject);

    [JSImport("summarize", "JSObjectShim")]
    public static partial string Summarize(JSObject jsObject);

    [JSImport("globalThis.console.log")]
    public static partial void ConsoleLog([JSMarshalAs<JSType.Any>] object
value);
}

public static class JSObjectUsage
{
    public static async Task Run()
    {
        await JSHost.ImportAsync("JSObjectShim", "/JSObjectShim.js");
        JSObject jsObject = JSObjectInterop.CreateObject();
        JSObjectInterop.ConsoleLog(jsObject);
        JSObjectInterop.IncrementAnswer(jsObject);
        // An updated object isn't retrieved. The change is reflected in the
        // existing instance.
        JSObjectInterop.ConsoleLog(jsObject);

        // JSObject exposes several methods for interacting with properties.
        jsObject.SetProperty("question", "What is the answer?");
        JSObjectInterop.ConsoleLog(jsObject);

        // We can't directly JSImport an instance method on the jsObject,
but we
        // can pass the object reference and have the JS shim call the
instance
        // method.
        string summary = JSObjectInterop.Summarize(jsObject);
        Console.WriteLine("Summary: " + summary);
    }
}
```

In `Program.Main`:

C#

```csharp
await JSObjectUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> {name: 'Example JS Object', answer: 41, question: null, Symbol(wasm cs_owned_js_handle): 5, summarize: ƒ}
> {name: 'Example JS Object', answer: 42, question: null, Symbol(wasm cs_owned_js_handle): 5, summarize: ƒ}
> {name: 'Example JS Object', answer: 42, question: 'What is the answer?', Symbol(wasm cs_owned_js_handle): 5, summarize: ƒ}
> Summary: Question: "What is the answer?" Answer: 42

Asynchronous interop

Many JS APIs are asynchronous and signal completion through either a callback, a Promise, or an async method. Ignoring asynchronous capabilities is often not an option, as subsequent code may depend upon the completion of the asynchronous operation and must be awaited.

JS methods using the `async` keyword or returning a Promise can be awaited in C# by a method returning a Task. As demonstrated below, the `async` keyword isn't used on the

C# method with the `[JSImport]` attribute because it doesn't use the `await` keyword within it. However, consuming code calling the method would typically use the `await` keyword and be marked as `async`, as demonstrated in the `PromisesUsage` example.

JS with a callback, such as a `setTimeout`, can be wrapped in a Promise before returning from JS. Wrapping a callback in a Promise, as demonstrated in the function assigned to `Wait2Seconds`, is only appropriate when the callback is called exactly once. Otherwise, a C# Action can be passed to listen for a callback that may be called zero or many times, which is demonstrated in the Subscribing to JS events section.

`PromisesShim.js`:

JavaScript

```javascript
export function wait2Seconds() {
  // This also demonstrates wrapping a callback-based API in a promise to
  // make it awaitable.
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(); // Resolve promise after 2 seconds
    }, 2000);
  });
}

// Return a value via resolve in a promise.
export function waitGetString() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("String From Resolve"); // Return a string via promise
    }, 500);
  });
}

export function waitGetDate() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(new Date('1988-11-24')); // Return a date via promise
    }, 500);
  });
}

// Demonstrates an awaitable fetch.
export function fetchCurrentUrl() {
  // This method returns the promise returned by .then(*.text())
  // and .NET awaits the returned promise.
  return fetch(globalThis.window.location, { method: 'GET' })
    .then(response => response.text());
}

// .NET can await JS methods using the async/await JS syntax.
export async function asyncFunction() {
  await wait2Seconds();
}

// A Promise.reject can be used to signal failure and is bubbled to .NET
code
// as a JSException.
export function conditionalSuccess(shouldSucceed) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldSucceed)
        resolve(); // Success
      else
        reject("Reject: ShouldSucceed == false"); // Failure
    }, 500);
  });
}
```

Don't use the `async` keyword in the C# method signature. Returning Task or Task<TResult> is sufficient.

When calling asynchronous JS methods, we often want to wait until the JS method completes execution. If loading a resource or making a request, we likely want the following code to assume the action is completed.

If the JS shim returns a Promise, then C# can treat it as an awaitable Task/Task<TResult>.

`PromisesInterop.cs`:

```C#
using System;
using System.Diagnostics;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class PromisesInterop
{
    // For a promise with void return type, declare a Task return type:
    [JSImport("wait2Seconds", "PromisesShim")]
    public static partial Task Wait2Seconds();

    [JSImport("waitGetString", "PromisesShim")]
    public static partial Task<string> WaitGetString();

    // Some return types require a [return: JSMarshalAs...] declaring the
    // Promise's return type corresponding to Task<T>.
    [JSImport("waitGetDate", "PromisesShim")]
    [return: JSMarshalAs<JSType.Promise<JSType.Date>>()]
    public static partial Task<DateTime> WaitGetDate();
    [JSImport("fetchCurrentUrl", "PromisesShim")]
    public static partial Task<string> FetchCurrentUrl();

    [JSImport("asyncFunction", "PromisesShim")]
    public static partial Task AsyncFunction();

    [JSImport("conditionalSuccess", "PromisesShim")]
    public static partial Task ConditionalSuccess(bool shouldSucceed);
}

public static class PromisesUsage
{
    public static async Task Run()
    {
        await JSHost.ImportAsync("PromisesShim", "/PromisesShim.js");

        Stopwatch sw = new();
        sw.Start();

        await PromisesInterop.Wait2Seconds(); // Await Promise
        Console.WriteLine($"Waited {sw.Elapsed.TotalSeconds:#.0}s.");

        sw.Restart();
        string str =
            await PromisesInterop.WaitGetString(); // Await promise (string
return)
        Console.WriteLine(
            $"Waited {sw.Elapsed.TotalSeconds:#.0}s for WaitGetString:
'{str}'");

        sw.Restart();
        // Await promise with string return.
        DateTime date = await PromisesInterop.WaitGetDate();
        Console.WriteLine(
            $"Waited {sw.Elapsed.TotalSeconds:#.0}s for WaitGetDate:
'{date}'");

        // Await a JS fetch.
        string responseText = await PromisesInterop.FetchCurrentUrl();
        Console.WriteLine($"responseText.Length: {responseText.Length}");

        sw.Restart();

        await PromisesInterop.AsyncFunction(); // Await an async JS method
        Console.WriteLine(
            $"Waited {sw.Elapsed.TotalSeconds:#.0}s for AsyncFunction.");

        try
        {
            // Handle a promise rejection. Await an async JS method.
            await PromisesInterop.ConditionalSuccess(shouldSucceed: false);
        }
        catch (JSException ex) // Catch JS exception
        {
            Console.WriteLine($"JS Exception Caught: '{ex.Message}'");
        }
    }
}
```

In `Program.Main`:

C#

```
await PromisesUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> Waited 2.0s.
> Waited .5s for WaitGetString: 'String From Resolve'
> Waited .5s for WaitGetDate: '11/24/1988 12:00:00 AM'
> responseText.Length: 582
> Waited 2.0s for AsyncFunction.
> JS Exception Caught: 'Reject: ShouldSucceed == false'

Type mapping limitations

Some type mappings requiring nested generic types in the JSMarshalAs definition aren't currently supported. For example, returning a Promise for an array such as `[return: JSMarshalAs<JSType.Promise<JSType.Array<JSType.Number>>>()]` generates a compile-time error. An appropriate workaround varies depending on the scenario, but one option is to represent the array as a JSObject reference. This may be sufficient if accessing individual elements within .NET isn't necessary and the reference can be passed to other JS methods that act on the array. Alternatively, a dedicated method can take the JSObject reference as a parameter and return the materialized array, as demonstrated by the following `UnwrapJSObjectAsIntArray` example. In this case, the JS method has no type checking, and the developer has the responsibility to ensure a JSObject wrapping the appropriate array type is passed.

JavaScript

```
export function waitGetIntArrayAsObject() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve([1, 2, 3, 4, 5]); // Return an array from the Promise
    }, 500);
  });
}

export function unwrapJSObjectAsIntArray(jsObject) {
  return jsObject;
}
```

```C#
// Not supported, generates compile-time error.
// [JSImport("waitGetArray", "PromisesShim")]
// [return: JSMarshalAs<JSType.Promise<JSType.Array<JSType.Number>>>()]
// public static partial Task<int[]> WaitGetIntArray();

// Workaround, take the return the call and pass it to
UnwrapJSObjectAsIntArray.
// Return a JSObject reference to a JS number array.
[JSImport("waitGetIntArrayAsObject", "PromisesShim")]
[return: JSMarshalAs<JSType.Promise<JSType.Object>>()]
public static partial Task<JSObject> WaitGetIntArrayAsObject();

// Takes a JSObject reference to a JS number array, and returns the array as
a C#
// int array.
[JSImport("unwrapJSObjectAsIntArray", "PromisesShim")]
[return: JSMarshalAs<JSType.Array<JSType.Number>>()]
public static partial int[] UnwrapJSObjectAsIntArray(JSObject intArray);
//...
```

In `Program.Main`:

```C#
JSObject arrayAsJSObject = await PromisesInterop.WaitGetIntArrayAsObject();
int[] intArray = PromisesInterop.UnwrapJSObjectAsIntArray(arrayAsJSObject);
```

Performance considerations

Marshalling of calls and the overhead of tracking objects across the interop boundary is more expensive than native .NET operations but should still demonstrate acceptable performance for a typical web app with moderate demand.

Object proxies, such as JSObject, which maintain references across the interop boundary, have additional memory overhead and impact how garbage collection affects these objects. Additionally, available memory might be exhausted without triggering garbage collection in some scenarios because memory pressure from JS and .NET isn't shared. This risk is significant when an excessive number of large objects are referenced across the interop boundary by relatively small JS objects, or vice versa where large .NET objects are referenced by JS proxies. In such cases, we recommend following deterministic disposal patterns with `using` scopes leveraging the IDisposable interface on JS objects.

The following benchmarks, which leverage earlier example code, demonstrate that interop operations are roughly an order of magnitude slower than those that remain within the .NET boundary, but the interop operations remain relatively fast. Additionally, consider that a user's device capabilities impact performance.

`JSObjectBenchmark.cs`:

```C#
using System;
using System.Diagnostics;

public static class JSObjectBenchmark
{
    public static void Run()
    {
        Stopwatch sw = new();
        var jsObject = JSObjectInterop.CreateObject();

        sw.Start();

        for (int i = 0; i < 1000000; i++)
        {
            JSObjectInterop.IncrementAnswer(jsObject);
        }

        sw.Stop();

        Console.WriteLine(
            $"JS interop elapsed time: {sw.Elapsed.TotalSeconds:#.0000}
seconds " +
            $"at {sw.Elapsed.TotalMilliseconds / 1000000d:#.000000} ms per "
+
            "operation");

        var pocoObject =
            new PocoObject { Question = "What is the answer?", Answer = 41
};
        sw.Restart();

        for (int i = 0; i < 1000000; i++)
        {
            pocoObject.IncrementAnswer();
        }

        sw.Stop();

        Console.WriteLine($".NET elapsed time:
{sw.Elapsed.TotalSeconds:#.0000} " +
            $"seconds at {sw.Elapsed.TotalMilliseconds / 1000000d:#.000000}
ms " +
            "per operation");

        Console.WriteLine($"Begin Object Creation");

        sw.Restart();

        for (int i = 0; i < 1000000; i++)
        {
            var jsObject2 = JSObjectInterop.CreateObject();
            JSObjectInterop.IncrementAnswer(jsObject2);
        }

        sw.Stop();

        Console.WriteLine(
            $"JS interop elapsed time: {sw.Elapsed.TotalSeconds:#.0000}
seconds " +
            $"at {sw.Elapsed.TotalMilliseconds / 1000000d:#.000000} ms per "
+
            "operation");

        sw.Restart();

        for (int i = 0; i < 1000000; i++)
        {
            var pocoObject2 =
                new PocoObject { Question = "What is the answer?", Answer =
0 };
            pocoObject2.IncrementAnswer();
        }

        sw.Stop();
        Console.WriteLine(
            $".NET elapsed time: {sw.Elapsed.TotalSeconds:#.0000} seconds at
" +
            $"{sw.Elapsed.TotalMilliseconds / 1000000d:#.000000} ms per
operation");
    }

    public class PocoObject // Plain old CLR object
    {
        public string Question { get; set; }
        public int Answer { get; set; }

        public void IncrementAnswer() => Answer += 1;
    }
}
```

In `Program.Main`:

C#

```
JSObjectBenchmark.Run();
```

The preceding example displays the following output in the browser's debug console:

> JS interop elapsed time: .2536 seconds at .000254 ms per operation
> .NET elapsed time: .0210 seconds at .000021 ms per operation
> Begin Object Creation
> JS interop elapsed time: 2.1686 seconds at .002169 ms per operation
> .NET elapsed time: .1089 seconds at .000109 ms per operation

Subscribing to JS events

.NET code can subscribe to JS events and handle JS events by passing a C# Action to a JS function to act as a handler. The JS shim code handles subscribing to the event.

> ⚠ **Warning**
>
> Interacting with individual properties of the DOM via JS interop, as the guidance in this section demonstrates, is relatively slow and may lead to the creation of many proxies that create high garbage collection pressure. The following pattern isn't generally recommended. Use the following pattern for no more than a few elements. For more information, see the Performance considerations section.

A nuance of `removeEventListener` is that it requires a reference to the function previously passed to `addEventListener`. When a C# Action is passed across the interop boundary, it's wrapped in a JS proxy object. Therefore, passing the same C# Action to both `addEventListener` and `removeEventListener` results in generating two different JS proxy objects wrapping the Action. These references are different, thus `removeEventListener` isn't able to find the event listener to remove. To address this problem, the following examples wrap the C# Action in a JS function and return the reference as a JSObject from the subscribe call to pass later to the unsubscribe call. Because the C# Action is returned and passed as a JSObject, the same reference is used for both calls, and the event listener can be removed.

`EventsShim.js`:

JavaScript

```javascript
export function subscribeEventById(elementId, eventName, listenerFunc) {
  const elementObj = document.getElementById(elementId);
  // Need to wrap the Managed C# action in JS func (only because it is being
  // returned).
  let handler = function (event) {
    listenerFunc(event.type, event.target.id); // Decompose object to
primitives
  }.bind(elementObj);

  elementObj.addEventListener(eventName, handler, false);
  // Return JSObject reference so it can be used for removeEventListener
later.
  return handler;
}

// Param listenerHandler must be the JSObject reference returned from the
prior
// SubscribeEvent call.
export function unsubscribeEventById(elementId, eventName, listenerHandler)
{
  const elementObj = document.getElementById(elementId);
  elementObj.removeEventListener(eventName, listenerHandler, false);
}

export function triggerClick(elementId) {
  const elementObj = document.getElementById(elementId);
  elementObj.click();
}

export function getElementById(elementId) {
  return document.getElementById(elementId);
}

export function subscribeEvent(elementObj, eventName, listenerFunc) {
  let handler = function (e) {
    listenerFunc(e);
  }.bind(elementObj);

  elementObj.addEventListener(eventName, handler, false);
  return handler;
}

export function unsubscribeEvent(elementObj, eventName, listenerHandler) {
  return elementObj.removeEventListener(eventName, listenerHandler, false);
}

export function subscribeEventFailure(elementObj, eventName, listenerFunc) {
  // It's not strictly required to wrap the C# action listenerFunc in a JS
  // function.
  elementObj.addEventListener(eventName, listenerFunc, false);
  // If you need to return the wrapped proxy object, you will receive an
error
  // when it tries to wrap the existing proxy in an additional proxy:
  // Error: "JSObject proxy of ManagedObject proxy is not supported."
  return listenerFunc;
}
```

`EventsInterop.cs`:

C#

```csharp
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class EventsInterop
{
    [JSImport("subscribeEventById", "EventsShim")]
    public static partial JSObject SubscribeEventById(string elementId,
        string eventName,
        [JSMarshalAs<JSType.Function<JSType.String, JSType.String>>]
        Action<string, string> listenerFunc);

    [JSImport("unsubscribeEventById", "EventsShim")]
    public static partial void UnsubscribeEventById(string elementId,
        string eventName, JSObject listenerHandler);

    [JSImport("triggerClick", "EventsShim")]
    public static partial void TriggerClick(string elementId);

    [JSImport("getElementById", "EventsShim")]
    public static partial JSObject GetElementById(string elementId);

    [JSImport("subscribeEvent", "EventsShim")]
    public static partial JSObject SubscribeEvent(JSObject htmlElement,
        string eventName,
        [JSMarshalAs<JSType.Function<JSType.Object>>]
        Action<JSObject> listenerFunc);

    [JSImport("unsubscribeEvent", "EventsShim")]
    public static partial void UnsubscribeEvent(JSObject htmlElement,
        string eventName, JSObject listenerHandler);
}

public static class EventsUsage
{
    public static async Task Run()
    {
        await JSHost.ImportAsync("EventsShim", "/EventsShim.js");

        Action<string, string> listenerFunc = (eventName, elementId) =>
            Console.WriteLine(
                $"In C# event listener: Event {eventName} from ID
{elementId}");

        // Assumes two buttons exist on the page with ids of "btn1" and
"btn2"
        JSObject listenerHandler1 =
            EventsInterop.SubscribeEventById("btn1", "click", listenerFunc);
        JSObject listenerHandler2 =
            EventsInterop.SubscribeEventById("btn2", "click", listenerFunc);
        Console.WriteLine("Subscribed to btn1 & 2.");
        EventsInterop.TriggerClick("btn1");
        EventsInterop.TriggerClick("btn2");

        EventsInterop.UnsubscribeEventById("btn2", "click",
listenerHandler2);
        Console.WriteLine("Unsubscribed btn2.");
        EventsInterop.TriggerClick("btn1");
        EventsInterop.TriggerClick("btn2"); // Doesn't trigger because
unsubscribed
        EventsInterop.UnsubscribeEventById("btn1", "click",
listenerHandler1);
        // Pitfall: Using a different handler for unsubscribe silently
fails.
        // EventsInterop.UnsubscribeEventById("btn1", "click",
listenerHandler2);

        // With JSObject as event target and event object.
        Action<JSObject> listenerFuncForElement = (eventObj) =>
        {
            string eventType = eventObj.GetPropertyAsString("type");
            JSObject target = eventObj.GetPropertyAsJSObject("target");
            Console.WriteLine(
                $"In C# event listener: Event {eventType} from " +
                $"ID {target.GetPropertyAsString("id")}");
        };

        JSObject htmlElement = EventsInterop.GetElementById("btn1");
        JSObject listenerHandler3 = EventsInterop.SubscribeEvent(
            htmlElement, "click", listenerFuncForElement);
        Console.WriteLine("Subscribed to btn1.");
        EventsInterop.TriggerClick("btn1");
        EventsInterop.UnsubscribeEvent(htmlElement, "click",
listenerHandler3);
        Console.WriteLine("Unsubscribed btn1.");
        EventsInterop.TriggerClick("btn1");
    }
}
```

In `Program.Main`:

C#

```
await EventsUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> Subscribed to btn1 & 2.
> In C# event listener: Event click from ID btn1
> In C# event listener: Event click from ID btn2
> Unsubscribed btn2.

> In C# event listener: Event click from ID btn1
> Subscribed to btn1.
> In C# event listener: Event click from ID btn1
> Unsubscribed btn1.

JS `[JSImport]`/`[JSExport]` interop scenarios

The following articles focus on running a .NET WebAssembly module in a JS host, such as a browser:

- JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project
- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor

JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project

Article • 08/22/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

This article explains how to setup a WebAssembly Browser App project to run .NET from JavaScript (JS) using JS `[JSImport]`/`[JSExport]` interop. For additional information and examples, see JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly.

For additional guidance, see the Configuring and hosting .NET WebAssembly applications guidance in the .NET Runtime (`dotnet/runtime`) GitHub repository.

Existing JS apps can use the expanded client-side WebAssembly support to reuse .NET libraries from JS or to build novel .NET-based apps and frameworks.

ⓘ **Note**

This article focuses on running .NET from JS apps without any dependency on Blazor. For guidance on using `[JSImport]`/`[JSExport]` interop in Blazor WebAssembly apps, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

These approaches are appropriate when you only expect the Blazor app to run on WebAssembly (WASM). Libraries can make a runtime check to determine if the app is running on WASM by calling OperatingSystem.IsBrowser.

Prerequisites

.NET SDK (latest version)

Install the `wasm-tools` workload in an administrative command shell, which brings in the related MSBuild targets:

.NET CLI

```
dotnet workload install wasm-tools
```

The tools can also be installed via Visual Studio's installer under the **ASP.NET and web development** workload in the Visual Studio installer. Select the **.NET WebAssembly build tools** option from the list of optional components.

Optionally, install the `wasm-experimental` workload, which adds the following experimental project templates:

- *WebAssembly Browser App* for getting started with .NET on WebAssembly in a browser app.
- *WebAssembly Console App* for getting started in a Node.js-based console app.

After installing the workload, these new templates can be selected when creating a new project. This workload isn't required if you plan to integrate JS `[JSImport]`/`[JSExport]` interop into an existing JS app.

.NET CLI

```
dotnet workload install wasm-experimental
```

The templates can also be installed from the Microsoft.NET.Runtime.WebAssembly.Templates NuGet package with the following command:

.NET CLI

```
dotnet new install Microsoft.NET.Runtime.WebAssembly.Templates
```

For more information, see the Experimental workload and project templates section.

Namespace

The JS interop API described in this article is controlled by attributes in the System.Runtime.InteropServices.JavaScript namespace.

Project configuration

To configure a project (`.csproj`) to enable JS interop:

- Set the target framework moniker (`{TARGET FRAMEWORK}` placeholder):

  XML

  ```xml
  <TargetFramework>{TARGET FRAMEWORK}</TargetFramework>
  ```

  .NET 7 (`net7.0`) or later is supported.

- Enable the AllowUnsafeBlocks property, which permits the code generator in the Roslyn compiler to use pointers for JS interop:

  XML

  ```xml
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  ```

  ⚠ **Warning**

  The JS interop API requires enabling AllowUnsafeBlocks. Be careful when implementing your own unsafe code in .NET apps, which can introduce security and stability risks. For more information, see Unsafe code, pointer types, and function pointers.

The following is an example project file (`.csproj`) after configuration. The `{TARGET FRAMEWORK}` placeholder is the target framework:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.WebAssembly">

  <PropertyGroup>
    <TargetFramework>{TARGET FRAMEWORK}</TargetFramework>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  </PropertyGroup>

</Project>
```

JavaScript interop on WASM

APIs in the following example are imported from `dotnet.js`. These APIs enable you to set up named modules that can be imported into your C# code and call into methods exposed by your .NET code, including `Program.Main`.

> ⓘ **Important**
>
> "Import" and "export" throughout this article are defined from the perspective of .NET:
>
> - An app imports JS methods so that they can be called from .NET.
> - The app exports .NET methods so that they can be called from JS.

In the following example:

- The `dotnet.js` file is used to create and start the .NET WebAssembly runtime. `dotnet.js` is generated as part of the app's build output.

  > ⓘ **Important**
  >
  > To integrate with an existing app, copy the contents of the publish output folder† to the existing app's deployment assets so that it can be served along with the rest of the app. For production deployments, publish the app with the `dotnet publish -c Release` command in a command shell and deploy the output folder's contents with the app.
  >
  > †The publish output folder is the target location of your publish profile. The default for a **Release** profile in .NET 8 or later is `bin/Release/{TARGET FRAMEWORK}/publish`, where the `{TARGET FRAMEWORK}` placeholder is the target framework (for example, `net8.0`).

- `dotnet.create()` sets up the .NET WebAssembly runtime.

- `setModuleImports` associates a name with a module of JS functions for import into .NET. The JS module contains a `dom.setInnerText` function, which accepts and element selector and time to display the current stopwatch time in the UI. The name of the module can be any string (it doesn't need to be a file name), but it must match the name used with the `JSImportAttribute` (explained later in this article). The `dom.setInnerText` function is imported into C# and called by the C# method `SetInnerText`. The `SetInnerText` method is shown later in this section.

- `exports.StopwatchSample.Reset()` calls into .NET (`StopwatchSample.Reset`) from JS. The `Reset` C# method restarts the stopwatch if it's running or resets it if it isn't running. The `Reset` method is shown later in this section.

- `exports.StopwatchSample.Toggle()` calls into .NET (`StopwatchSample.Toggle`) from JS. The `Toggle` C# method starts or stops the stopwatch depending on if it's currently running or not. The `Toggle` method is shown later in this section.
- `runMain()` runs `Program.Main`.

JS module:

```JavaScript
import { dotnet } from './_framework/dotnet.js'

const { setModuleImports, getAssemblyExports, getConfig, runMain } = await
dotnet
  .withApplicationArguments("start")
  .create();

setModuleImports('main.js', {
  dom: {
    setInnerText: (selector, time) =>
      document.querySelector(selector).innerText = time
  }
});

const config = getConfig();
const exports = await getAssemblyExports(config.mainAssemblyName);

document.getElementById('reset').addEventListener('click', e => {
  exports.StopwatchSample.Reset();
  e.preventDefault();
});

const pauseButton = document.getElementById('pause');
pauseButton.addEventListener('click', e => {
  const isRunning = exports.StopwatchSample.Toggle();
  pauseButton.innerText = isRunning ? 'Pause' : 'Start';
  e.preventDefault();
});

await runMain();
```

To import a JS function so it can be called from C#, use the new JSImportAttribute on a matching method signature. The first parameter to the JSImportAttribute is the name of the JS function to import and the second parameter is the name of the module.

In the following example, the `dom.setInnerText` function is called from the `main.js` module when `SetInnerText` method is called:

C#

```csharp
[JSImport("dom.setInnerText", "main.js")]
internal static partial void SetInnerText(string selector, string content);
```

In the imported method signature, you can use .NET types for parameters and return values, which are marshalled automatically by the runtime. Use JSMarshalAsAttribute<T> to control how the imported method parameters are marshalled. For example, you might choose to marshal a `long` as System.Runtime.InteropServices.JavaScript.JSType.Number or System.Runtime.InteropServices.JavaScript.JSType.BigInt. You can pass Action/Func<TResult> callbacks as parameters, which are marshalled as callable JS functions. You can pass both JS and managed object references, and they are marshaled as proxy objects, keeping the object alive across the boundary until the proxy is garbage collected. You can also import and export asynchronous methods with a Task result, which are marshaled as JS promises. Most of the marshalled types work in both directions, as parameters and as return values, on both imported and exported methods.

For additional type mapping information and examples, see JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly.

Functions accessible on the global namespace can be imported by using the globalThis prefix in the function name and by using the `[JSImport]` attribute without providing a module name. In the following example, console.log is prefixed with `globalThis`. The imported function is called by the C# `Log` method, which accepts a C# string message (`message`) and marshalls the C# string to a JS String for `console.log`:

C#

```csharp
[JSImport("globalThis.console.log")]
internal static partial void Log([JSMarshalAs<JSType.String>] string
message);
```

To export a .NET method so it can be called from JS, use the JSExportAttribute.

In the following example, each method is exported to JS and can be called from JS functions:

- The `Toggle` method starts or stops the stopwatch depending on its running state.
- The `Reset` method restarts the stopwatch if it's running or resets it if it isn't running.
- The `IsRunning` method indicates if the stopwatch is running.

C#

```csharp
[JSExport]
internal static bool Toggle()
{
    if (stopwatch.IsRunning)
    {
        stopwatch.Stop();
        return false;
    }
    else
    {
        stopwatch.Start();
        return true;
    }
}

[JSExport]
internal static void Reset()
{
    if (stopwatch.IsRunning)
        stopwatch.Restart();
    else
        stopwatch.Reset();

    Render();
}

[JSExport]
internal static bool IsRunning() => stopwatch.IsRunning;
```

Experimental workload and project templates

To demonstrate the JS interop functionality and obtain JS interop project templates, install the `wasm-experimental` workload:

.NET CLI

```
dotnet workload install wasm-experimental
```

The `wasm-experimental` workload contains two project templates: `wasmbrowser` and `wasmconsole`. These templates are experimental at this time, which means the developer workflow for the templates is evolving. However, the .NET and JS APIs used in the templates are supported in .NET 8 and provide a foundation for using .NET on WASM from JS.

The templates can also be installed from the Microsoft.NET.Runtime.WebAssembly.Templates NuGet package with the following command:

.NET CLI

```
dotnet new install Microsoft.NET.Runtime.WebAssembly.Templates
```

Browser app

You can create a browser app with the `wasmbrowser` template from the command line, which creates a web app that demonstrates using .NET and JS together in a browser:

.NET CLI

```
dotnet new wasmbrowser
```

Alternatively in Visual Studio, you can create the app using the **WebAssembly Browser App** project template.

Build the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet build
```

Build and run the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet run
```

Alternatively, install and use the dotnet serve command:

.NET CLI

```
dotnet serve -d:bin/$(Configuration)/{TARGET FRAMEWORK}/publish
```

In the preceding example, the `{TARGET FRAMEWORK}` placeholder is the target framework moniker.

Node.js console app

You can create a console app with the `wasmconsole` template, which creates an app that runs under WASM as a Node.js or V8 console app:

.NET CLI

```
dotnet new wasmconsole
```

Alternatively in Visual Studio, you can create the app using the **WebAssembly Console App** project template.

Build the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet build
```

Build and run the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet run
```

Alternatively, start any static file server from the publish output directory that contains the `main.mjs` file:

```
node bin/$(Configuration)/{TARGET FRAMEWORK}/{PATH}/main.mjs
```

In the preceding example, the `{TARGET FRAMEWORK}` placeholder is the target framework moniker, and the `{PATH}` placeholder is the path to the `main.mjs` file.

Additional resources

- JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly
- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor
- API documentation
  - [JSImport] attribute
  - [JSExport] attribute
- In the `dotnet/runtime` GitHub repository:
  - Configuring and hosting .NET WebAssembly applications
  - .NET WebAssembly runtime

  - dotnet.d.ts file (.NET WebAssembly runtime configuration)
- Use .NET from any JavaScript app in .NET 7

Use Grunt in ASP.NET Core

Article • 09/27/2024

Grunt is a JavaScript task runner that automates script minification, TypeScript compilation, code quality "lint" tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is fully supported in Visual Studio.

This example uses an empty ASP.NET Core project as its starting point, to show how to automate the client build process from scratch.

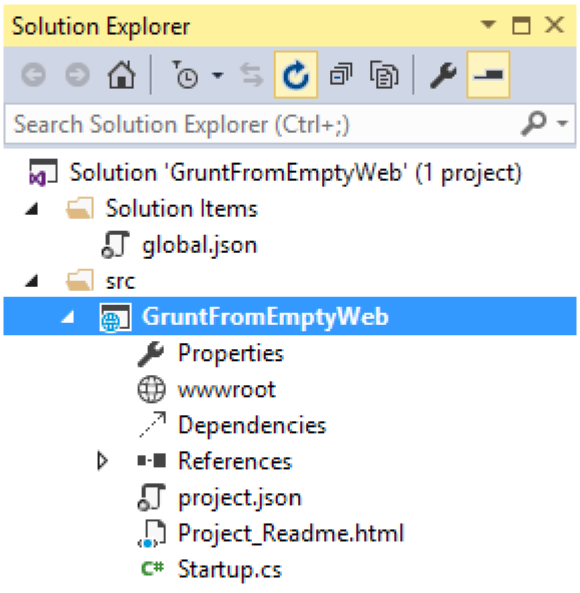
The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

- **grunt**: The Grunt task runner package.
- **grunt-contrib-clean**: A plugin that removes files or directories.
- **grunt-contrib-jshint**: A plugin that reviews JavaScript code quality.
- **grunt-contrib-concat**: A plugin that joins files into a single file.
- **grunt-contrib-uglify**: A plugin that minifies JavaScript to reduce size.
- **grunt-contrib-watch**: A plugin that watches file activity.

Preparing the application

To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio settings and will be our raw material to process using Grunt.

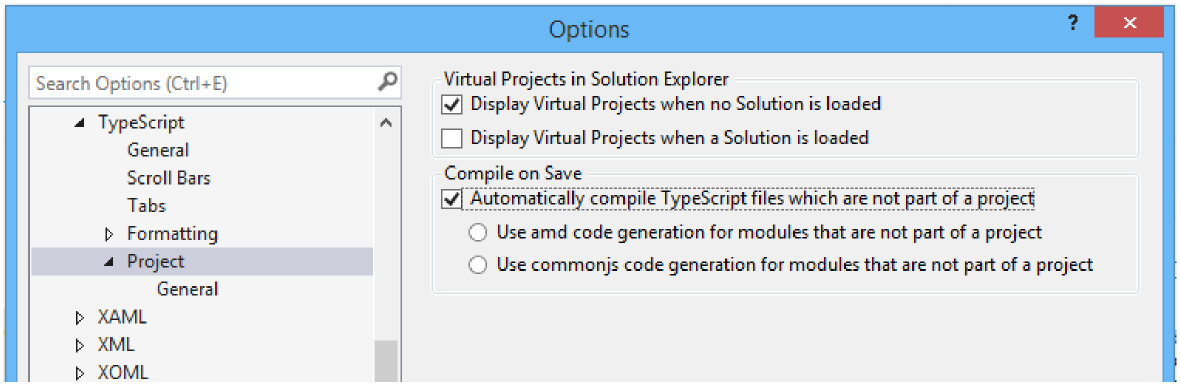
1. In Visual Studio, create a new `ASP.NET Web Application`.
2. In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template and click the OK button.
3. In the Solution Explorer, review the project structure. The `\src` folder includes empty `wwwroot` and `Dependencies` nodes.



Solution 'GruntFromEmptyWeb' (1 project)
- Solution Items
  - global.json
- src
  - GruntFromEmptyWeb
    - Properties
    - wwwroot
    - Dependencies
    - References
    - project.json
    - Project_Readme.html
    - Startup.cs

4. Add a new folder named `TypeScript` to your project directory.

5. Before adding any files, make sure that Visual Studio has the option 'compile on save' for TypeScript files checked. Navigate to **Tools** > **Options** > **Text Editor** > **Typescript** > **Project**:



Virtual Projects in Solution Explorer

[✓] Display Virtual Projects when no Solution is loaded

[ ] Display Virtual Projects when a Solution is loaded

Compile on Save

[✓] Automatically compile TypeScript files which are not part of a project

( ) Use amd code generation for modules that are not part of a project

( ) Use commonjs code generation for modules that are not part of a project

6. Right-click the `TypeScript` directory and select **Add** > **New Item** from the context menu. Select the **JavaScript file** item and name the file `Tastes.ts` (note the *.ts extension). Copy the line of TypeScript code below into the file (when you save, a new `Tastes.js` file will appear with the JavaScript source).

TypeScript

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. Add a second file to the **TypeScript** directory and name it `Food.ts`. Copy the code below into the file.

TypeScript

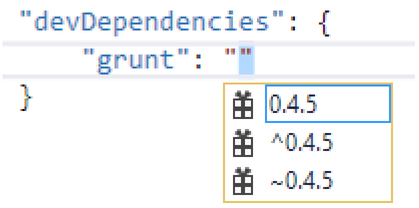
```
class Food {
  constructor(name: string, calories: number) {
    this._name = name;
    this._calories = calories;
  }

  private _name: string;
  get Name() {
    return this._name;
  }

  private _calories: number;
  get Calories() {
    return this._calories;
  }

  private _taste: Tastes;
  get Taste(): Tastes { return this._taste }
  set Taste(value: Tastes) {
    this._taste = value;
  }
}
```

Configuring NPM

Next, configure NPM to download grunt and grunt-tasks.

1. In the Solution Explorer, right-click the project and select **Add** > **New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, `package.json`, and click the **Add** button.

2. In the `package.json` file, inside the `devDependencies` object braces, enter "grunt". Select `grunt` from the Intellisense list and press the Enter key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press `Ctrl-Space` if Intellisense doesn't appear).



```
"devDependencies": {
    "grunt": ""
}
```

ⓘ Note

NPM uses semantic versioning to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme <major>.<minor>.<patch>. Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (0.4.5 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version. See the NPM semver version parser reference as a guide to the full expressivity that SemVer provides.

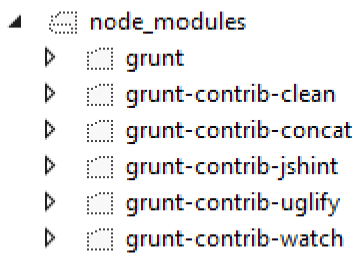
3. Add more dependencies to load grunt-contrib-* packages for *clean*, *jshint*, *concat*, *uglify*, and *watch* as shown in the example below. The versions don't need to match the example.

JSON

```json
"devDependencies": {
  "grunt": "0.4.5",
  "grunt-contrib-clean": "0.6.0",
  "grunt-contrib-jshint": "0.11.0",
  "grunt-contrib-concat": "0.5.1",
  "grunt-contrib-uglify": "0.8.0",
  "grunt-contrib-watch": "0.6.1"
}
```

4. Save the `package.json` file.

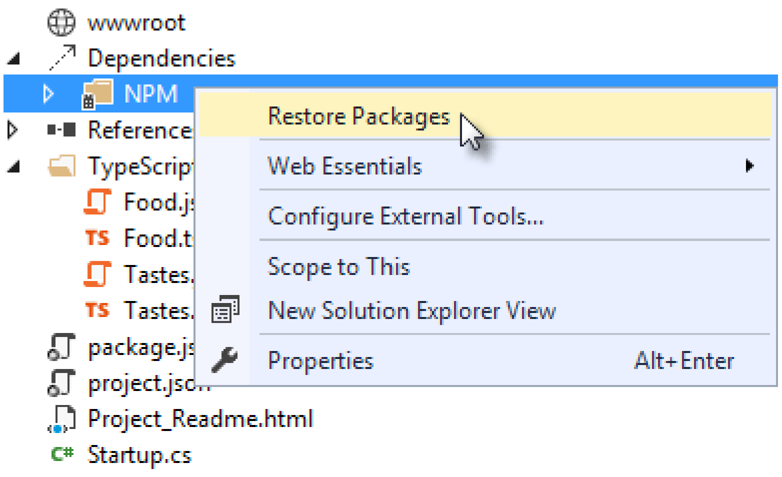
The packages for each `devDependencies` item will download, along with any files that each package requires. You can find the package files in the *node_modules* directory by enabling the **Show All Files** button in **Solution Explorer**.



```
◢ node_modules
   ▷ grunt
   ▷ grunt-contrib-clean
   ▷ grunt-contrib-concat
   ▷ grunt-contrib-jshint
   ▷ grunt-contrib-uglify
   ▷ grunt-contrib-watch
```

ⓘ Note

If you need to, you can manually restore dependencies in **Solution Explorer** by right-clicking on `Dependencies\NPM` and selecting the **Restore Packages** menu option.



Configuring Grunt

Grunt is configured using a manifest named `Gruntfile.js` that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add** > **New Item**. Select the **JavaScript File** item template, change the name to `Gruntfile.js`, and click the **Add** button.

2. Add the following code to `Gruntfile.js`. The `initConfig` function sets options for each package, and the remainder of the module loads and register tasks.

JavaScript

```javascript
module.exports = function (grunt) {
  grunt.initConfig({
  });
};
```

3. Inside the `initConfig` function, add options for the `clean` task as shown in the example `Gruntfile.js` below. The `clean` task accepts an array of directory strings. This task removes files from *wwwroot/lib* and removes the entire */temp* directory.

JavaScript

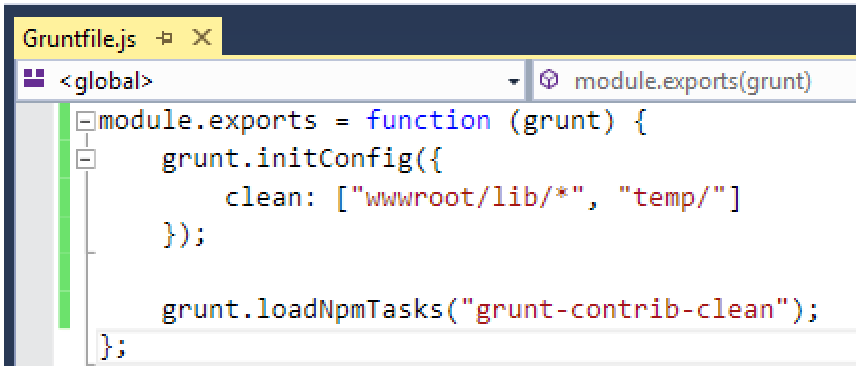
```javascript
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"],
  });
};
```

4. Below the `initConfig` function, add a call to `grunt.loadNpmTasks`. This will make the task runnable from Visual Studio.

JavaScript

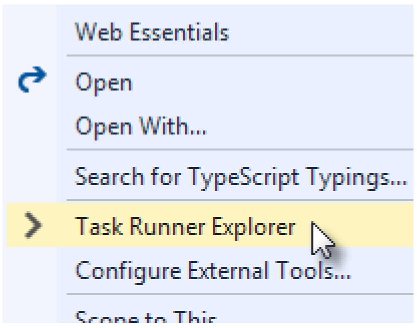
```
grunt.loadNpmTasks("grunt-contrib-clean");
```

5. Save `Gruntfile.js`. The file should look something like the screenshot below.

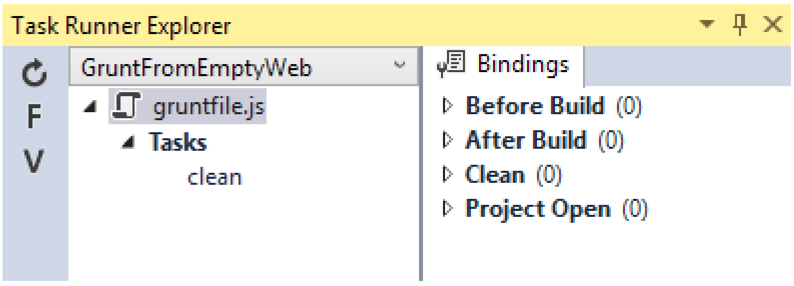


```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"]
    });

    grunt.loadNpmTasks("grunt-contrib-clean");
};
```

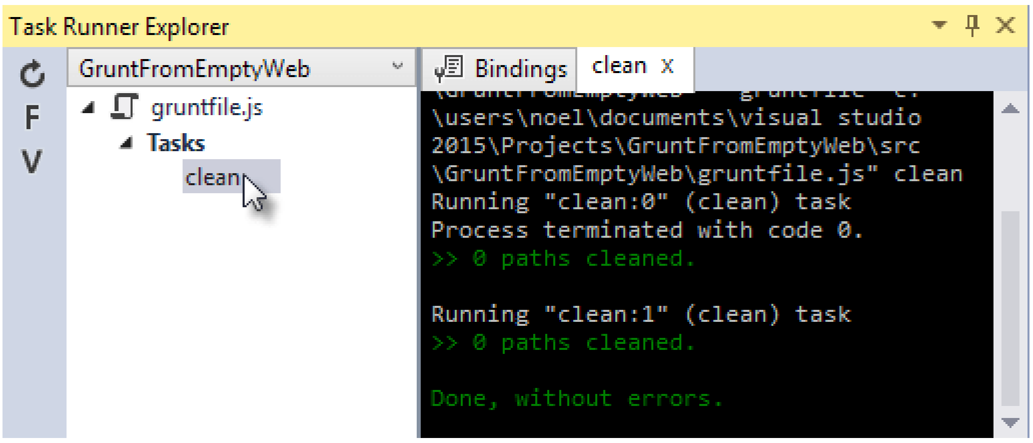
6. Right-click `Gruntfile.js` and select **Task Runner Explorer** from the context menu. The **Task Runner Explorer** window will open.



7. Verify that `clean` shows under **Tasks** in the **Task Runner Explorer**.



8. Right-click the clean task and select **Run** from the context menu. A command window displays progress of the task.



```
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" clean
Running "clean:0" (clean) task
Process terminated with code 0.
>> 0 paths cleaned.

Running "clean:1" (clean) task
>> 0 paths cleaned.

Done, without errors.
```

> ⓘ **Note**
>
> There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

9. In the `initConfig` function, add an entry for `concat` using the code below.

   The `src` property array lists files to combine, in the order that they should be combined. The `dest` property assigns the path to the combined file that's produced.

   JavaScript

   ```javascript
   concat: {
     all: {
       src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
       dest: 'temp/combined.js'
     }
   },
   ```

   > ⓘ **Note**
   >
   > The `all` property in the code above is the name of a target. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using IntelliSense or assign your own.

10. Add the `jshint` task using the code below.

    The jshint `code-quality` utility is run against every JavaScript file found in the *temp* directory.

    JavaScript

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

Note

The option "-W069" is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, `Tastes["Sweet"]` instead of `Tastes.Sweet`. The option turns off the warning to allow the rest of the process to continue.

11. Add the `uglify` task using the code below.

    The task minifies the `combined.js` file found in the temp directory and creates the result file in wwwroot/lib following the standard naming convention *<file name>.min.js*.

    JavaScript

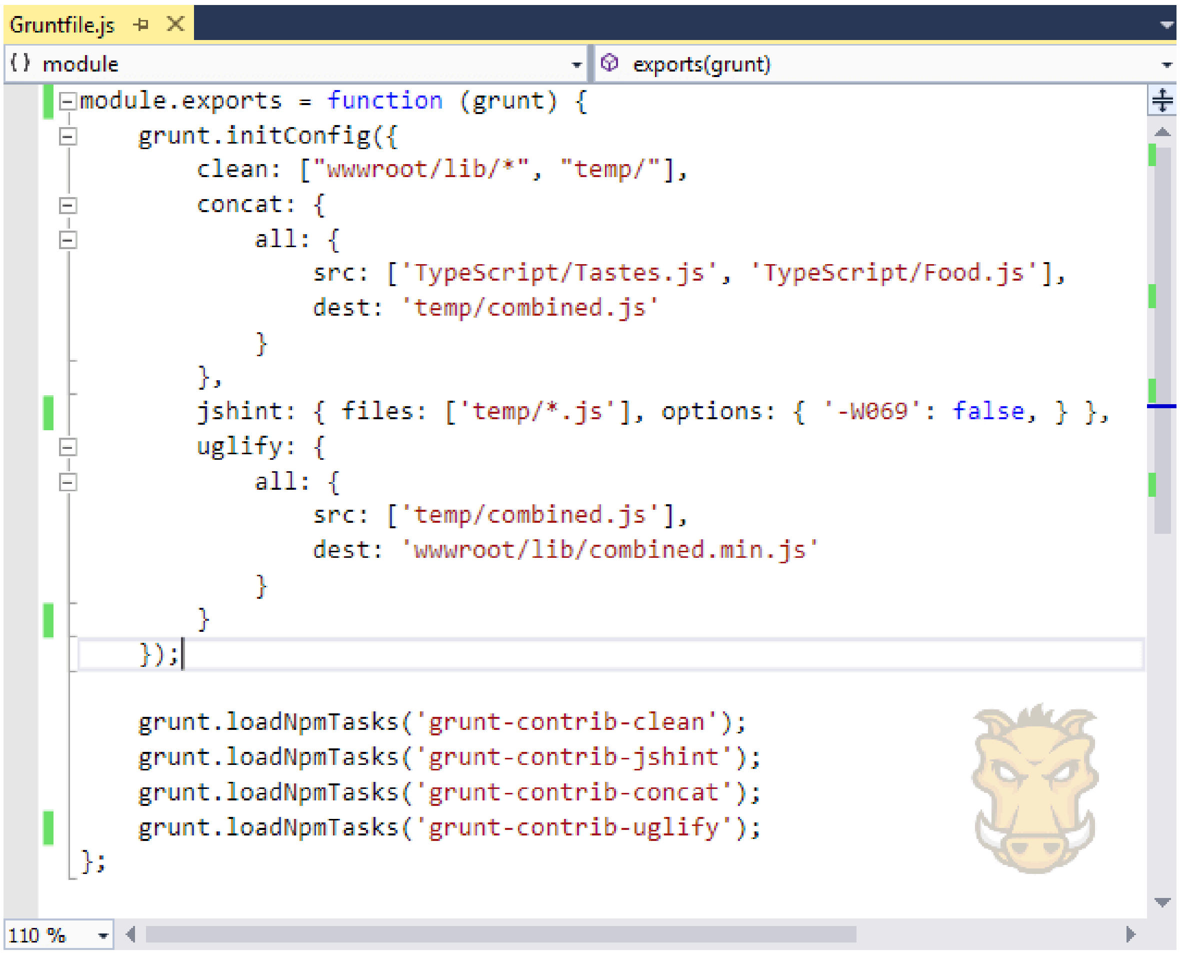
    ```
    uglify: {
     all: {
       src: ['temp/combined.js'],
       dest: 'wwwroot/lib/combined.min.js'
     }
    },
    ```

12. Under the call to `grunt.loadNpmTasks` that loads `grunt-contrib-clean`, include the same call for jshint, concat, and uglify using the code below.

    JavaScript

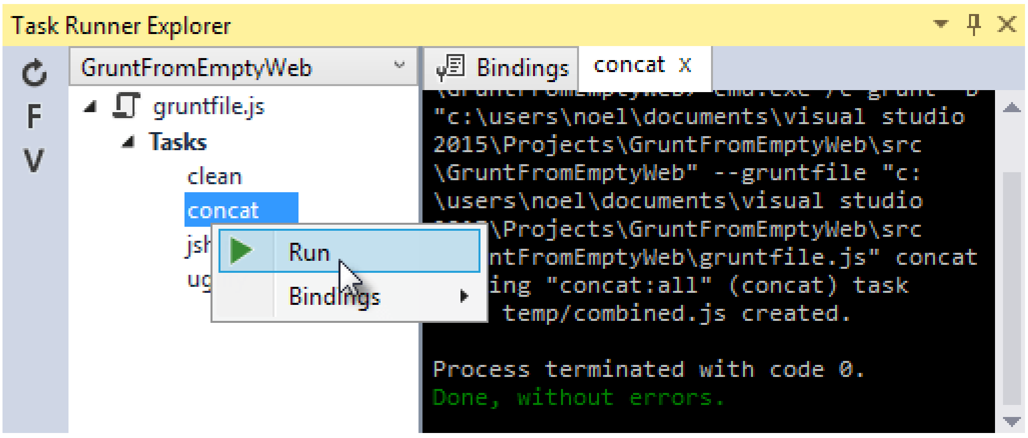
    ```
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    ```

13. Save `Gruntfile.js`. The file should look something like the example below.



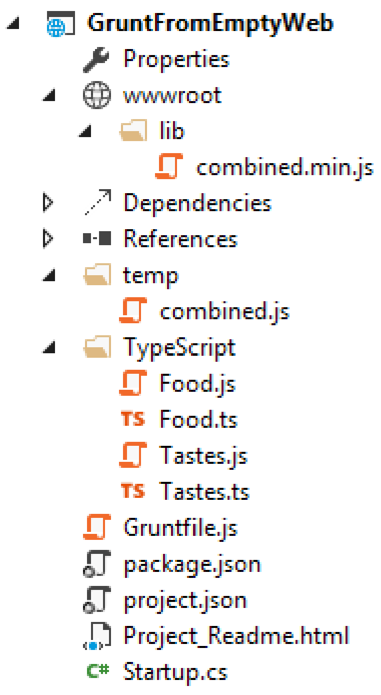
```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"],
        concat: {
            all: {
                src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
                dest: 'temp/combined.js'
            }
        },
        jshint: { files: ['temp/*.js'], options: { '-W069': false, } },
        uglify: {
            all: {
                src: ['temp/combined.js'],
                dest: 'wwwroot/lib/combined.min.js'
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-clean');
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
};
```

14. Notice that the **Task Runner Explorer** Tasks list includes `clean`, `concat`, `jshint` and `uglify` tasks. Run each task in order and observe the results in **Solution Explorer**. Each task should run without errors.



```
"c:\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb" --gruntfile "c:
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" concat
Running "concat:all" (concat) task
File temp/combined.js created.

Process terminated with code 0.
Done, without errors.
```

The concat task creates a new `combined.js` file and places it into the temp directory. The `jshint` task simply runs and doesn't produce output. The `uglify` task creates a new `combined.min.js` file and places it into *wwwroot/lib*. On completion, the solution should look something like the screenshot below:



ⓘ **Note**

For more information on the options for each package, visit https://www.npmjs.com/ and lookup the package name in the search box on the main page. For example, you can look up the grunt-contrib-clean package to get a documentation link that explains all of its parameters.

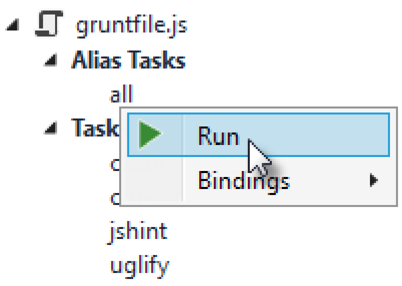
All together now

Use the Grunt `registerTask()` method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean -> concat -> jshint -> uglify, add the code below to the module. The code should be added to the same level as the loadNpmTasks() calls, outside initConfig.

JavaScript

```javascript
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The `all` task will run `clean`, `concat`, `jshint` and `uglify`, in order.



Watching for changes

A `watch` task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to initConfig to watch for changes to *.js files in the TypeScript directory. If a JavaScript file is changed, `watch` will run the `all` task.

JavaScript

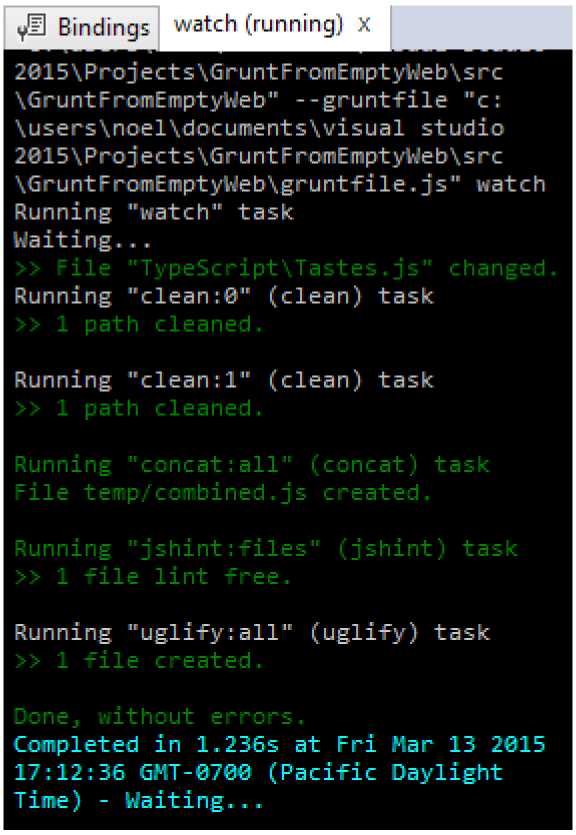
```javascript
watch: {
  files: ["TypeScript/*.js"],
  tasks: ["all"]
}
```

Add a call to `loadNpmTasks()` to show the `watch` task in Task Runner Explorer.

JavaScript

```javascript
grunt.loadNpmTasks('grunt-contrib-watch');
```

Right-click the watch task in Task Runner Explorer and select Run from the context menu. The command window that shows the watch task running will display a "Waiting..." message. Open one of the TypeScript files, add a space, and then save the file. This will trigger the watch task and trigger the other tasks to run in order. The screenshot below shows a sample run.

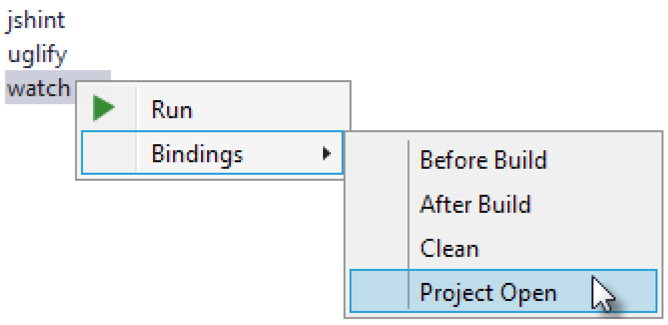


```
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb" --gruntfile "c:
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" watch
Running "watch" task
Waiting...
>> File "TypeScript\Tastes.js" changed.
Running "clean:0" (clean) task
>> 1 path cleaned.

Running "clean:1" (clean) task
>> 1 path cleaned.

Running "concat:all" (concat) task
File temp/combined.js created.

Running "jshint:files" (jshint) task
>> 1 file lint free.

Running "uglify:all" (uglify) task
>> 1 file created.

Done, without errors.
Completed in 1.236s at Fri Mar 13 2015
17:12:36 GMT-0700 (Pacific Daylight
Time) - Waiting...
```

Binding to Visual Studio events

Unless you want to manually start your tasks every time you work in Visual Studio, bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Bind `watch` so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the watch task and select **Bindings** > **Project Open** from the context menu.



Unload and reload the project. When the project loads again, the watch task starts running automatically.

Summary

Grunt is a powerful task runner that can be used to automate most client-build tasks. Grunt leverages NPM to deliver its packages, and features tooling integration with Visual Studio. Visual Studio's Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

Bundle and minify static assets in ASP.NET Core

Article • 08/08/2024

By Scott Addie and David Pine

This article explains the benefits of applying bundling and minification, including how these features can be used with ASP.NET Core web apps.

What is bundling and minification

Bundling and minification are two distinct performance optimizations you can apply in a web app. Used together, bundling and minification improve performance by reducing the number of server requests and reducing the size of the requested static assets.

Bundling and minification primarily improve the first page request load time. Once a web page has been requested, the browser caches the static assets (JavaScript, CSS, and images). So, bundling and minification don't improve performance when requesting the same page, or pages, on the same site requesting the same assets. If the expires header isn't set correctly on the assets and if bundling and minification aren't used, the browser's freshness heuristics mark the assets stale after a few days. Additionally, the browser requires a validation request for each asset. In this case, bundling and minification provide a performance improvement even after the first page request.

Bundling

Bundling combines multiple files into a single file. Bundling reduces the number of server requests that are necessary to render a web asset, such as a web page. You can create any number of individual bundles specifically for CSS, JavaScript, etc. Fewer files mean fewer HTTP requests from the browser to the server or from the service providing your application. This results in improved first page load performance.

Minification

Minification removes unnecessary characters from code without altering functionality. The result is a significant size reduction in requested assets (such as CSS, images, and JavaScript files). Common side effects of minification include shortening variable names to one character and removing comments and unnecessary whitespace.

Consider the following JavaScript function:

```javascript
AddAltToImg = function (imageTagAndImageID, imageContext) {
  ///<signature>
  ///<summary> Adds an alt tab to the image
  // </summary>
  //<param name="imgElement" type="String">The image selector.</param>
  //<param name="ContextForImage" type="String">The image context.</param>
  ///</signature>
  var imageElement = $(imageTagAndImageID, imageContext);
  imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

Minification reduces the function to the following:

```javascript
AddAltToImg=function(t,a){var
r=$(t,a);r.attr("alt",r.attr("id").replace(/ID/,""))};
```

In addition to removing the comments and unnecessary whitespace, the following parameter and variable names were renamed as follows:

Expand table

Original	Renamed
`imageTagAndImageID`	`t`
`imageContext`	`a`
`imageElement`	`r`

Impact of bundling and minification

The following table outlines differences between individually loading assets and using bundling and minification for a typical web app.

Expand table

Action	Without B/M	With B/M	Reduction
File Requests	18	7	61%

Action	Without B/M	With B/M	Reduction
Bytes Transferred (KB)	265	156	41%
Load Time (ms)	2360	885	63%

The load time improved, but this example ran locally. Greater performance gains are realized when using bundling and minification with assets transferred over a network.

The test app used to generate the figures in the preceding table demonstrates typical improvements that might not apply to a given app. We recommend testing an app to determine if bundling and minification yields an improved load time.

Choose a bundling and minification strategy

ASP.NET Core is compatible with WebOptimizer, an open-source bundling and minification solution. For set up instructions and sample projects, see WebOptimizer.
ASP.NET Core doesn't provide a native bundling and minification solution.

Third-party tools, such as Gulp and Webpack, provide workflow automation for bundling and minification, as well as linting and image optimization. By using bundling and minification, the minified files are created prior to the app's deployment. Bundling and minifying before deployment provides the advantage of reduced server load. However, it's important to recognize that bundling and minification increases build complexity and only works with static files.

Environment-based bundling and minification

As a best practice, the bundled and minified files of your app should be used in a production environment. During development, the original files make for easier debugging of the app.

Specify which files to include in your pages by using the Environment Tag Helper in your views. The Environment Tag Helper only renders its contents when running in specific environments.

The following `environment` tag renders the unprocessed CSS files when running in the `Development` environment:

CSHTML

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

The following `environment` tag renders the bundled and minified CSS files when running in an environment other than `Development`. For example, running in `Production` or `Staging` triggers the rendering of these stylesheets:

CSHTML

```cshtml
<environment exclude="Development">
    <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-
property="position" asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-
version="true" />
</environment>
```

Additional resources

- Use multiple environments
- Tag Helpers

Browser link in ASP.NET Core

Article • 06/17/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Nicolò Carandini and Tom Dykstra

Browser link is a Visual Studio feature. It creates a communication channel between the development environment and one or more web browsers. Use browser link to:

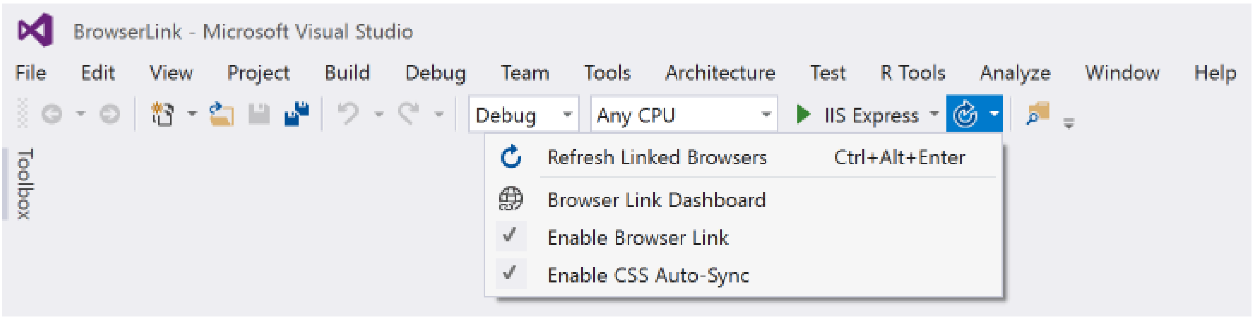
- Refresh your web app in several browsers at once.
- Test across multiple browsers with specific settings such as screen sizes.
- Select UI elements in browsers in real-time, see what markup and source it's correlated to in Visual Studio.
- Conduct real-time browser test automation.

Runtime compilation vs. hot reload

Use browser Link with runtime compilation or hot reload to see the effect of run-time changes in Razor (`.cshtml`) files. We recommend hot reload.

How to use browser link

When you have an ASP.NET Core project open, Visual Studio shows the browser link toolbar control next to the **Debug Type** toolbar control:

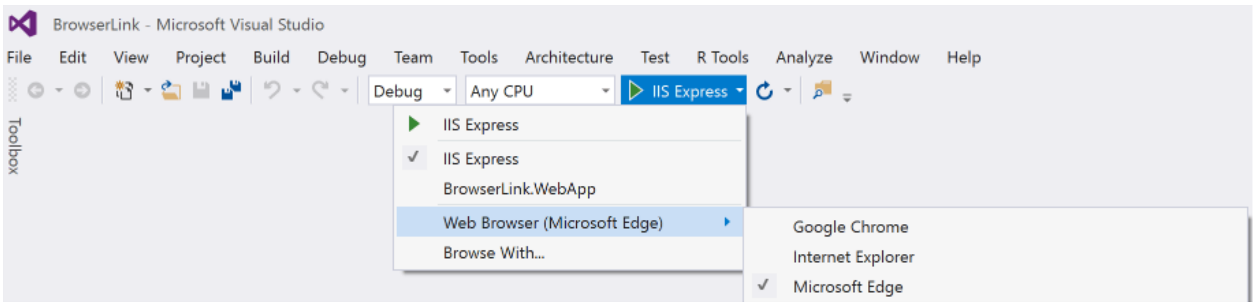


From the browser link toolbar control, you can:

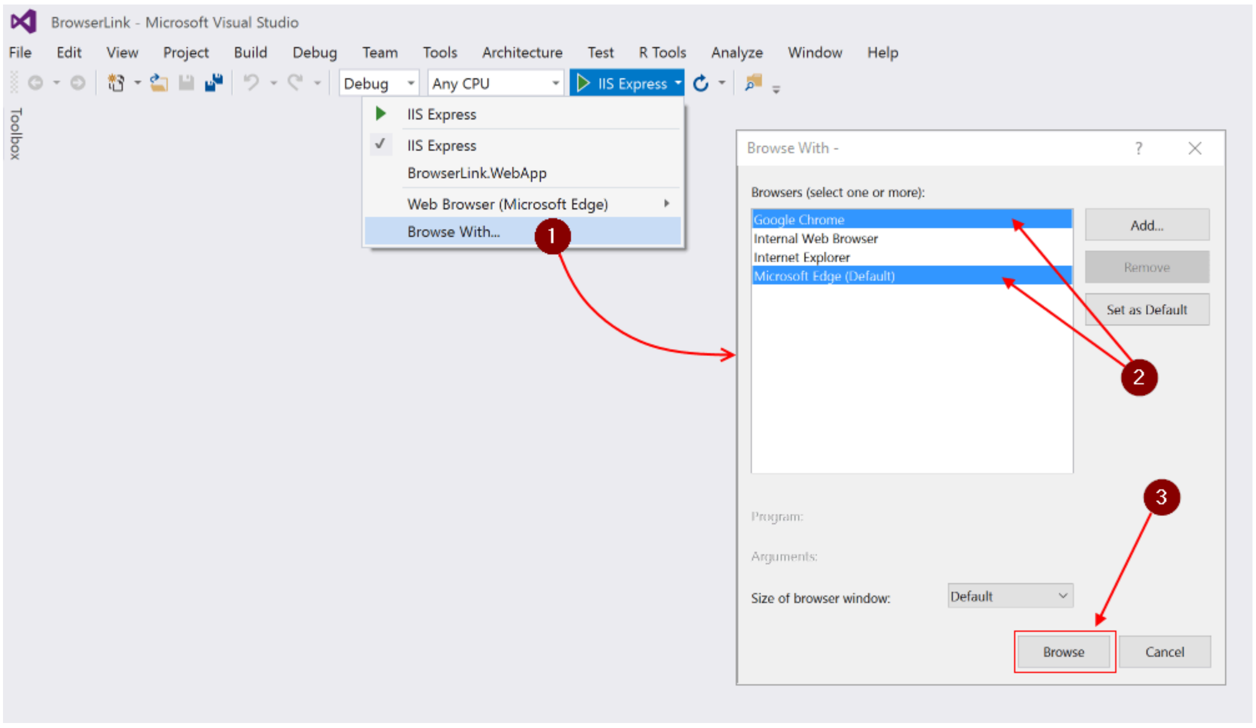
- Refresh the web app in several browsers at once.
- Open the **Browser Link Dashboard**.
- Enable or disable **Browser Link**.
- Enable or disable **CSS Hot Reload**.

Refresh the web app in several browsers at once

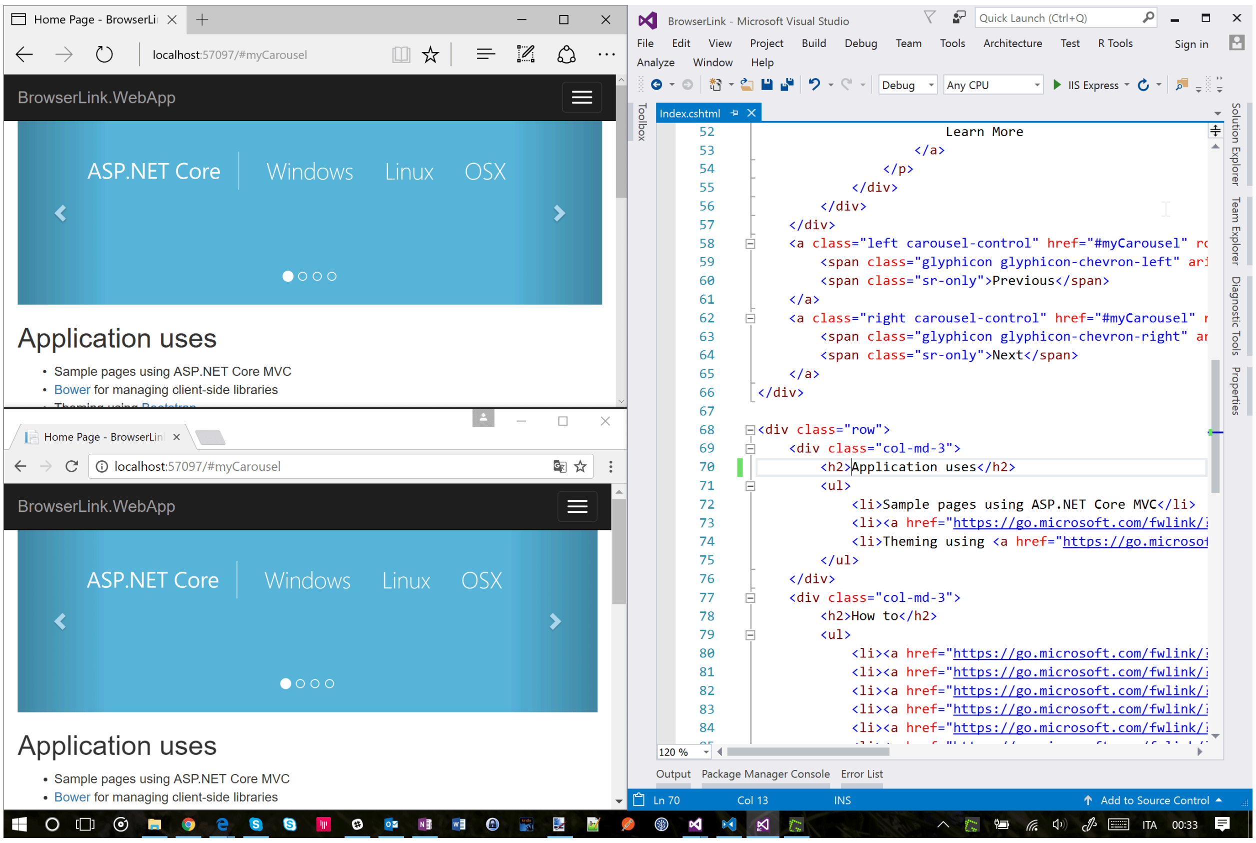
To choose a single web browser to launch when starting the project, use the drop-down menu in the **Debug Target** toolbar control:



To open multiple browsers at once, choose **Browse with...** from the same drop-down. Hold down the `Ctrl` key to select the browsers you want, and then click **Browse**:

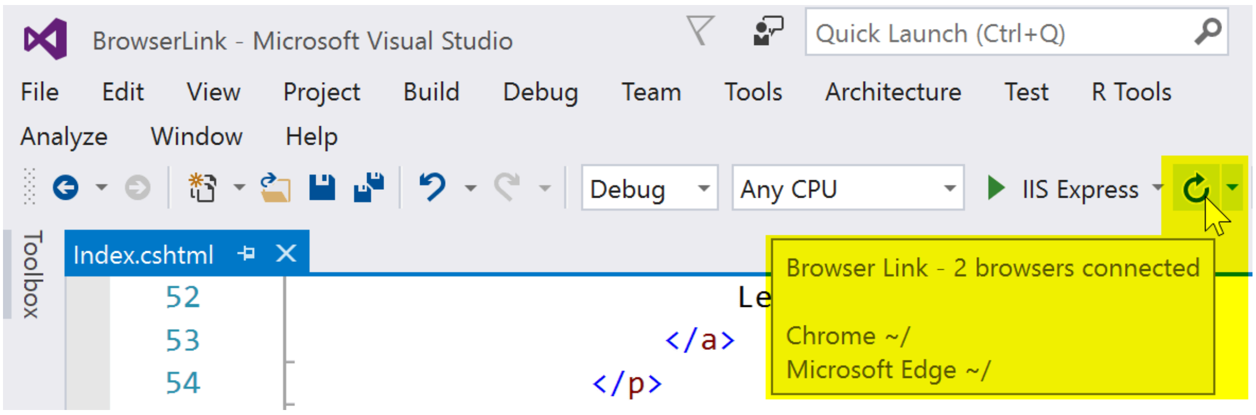


The following screenshot shows Visual Studio with the Index view open and two open browsers:



```
                    Learn More
                </a>
            </p>
        </div>
    </div>
</div>
<a class="left carousel-control" href="#myCarousel" ro
    <span class="glyphicon glyphicon-chevron-left" ari
    <span class="sr-only">Previous</span>
</a>
<a class="right carousel-control" href="#myCarousel" r
    <span class="glyphicon glyphicon-chevron-right" ar
    <span class="sr-only">Next</span>
</a>
</div>

<div class="row">
    <div class="col-md-3">
        <h2>Application uses</h2>
        <ul>
            <li>Sample pages using ASP.NET Core MVC</li>
            <li><a href="https://go.microsoft.com/fwlink/i
            <li>Theming using <a href="https://go.microsof
        </ul>
    </div>
    <div class="col-md-3">
        <h2>How to</h2>
        <ul>
            <li><a href="https://go.microsoft.com/fwlink/i
            <li><a href="https://go.microsoft.com/fwlink/i
            <li><a href="https://go.microsoft.com/fwlink/i
            <li><a href="https://go.microsoft.com/fwlink/i
            <li><a href="https://go.microsoft.com/fwlink/i
```

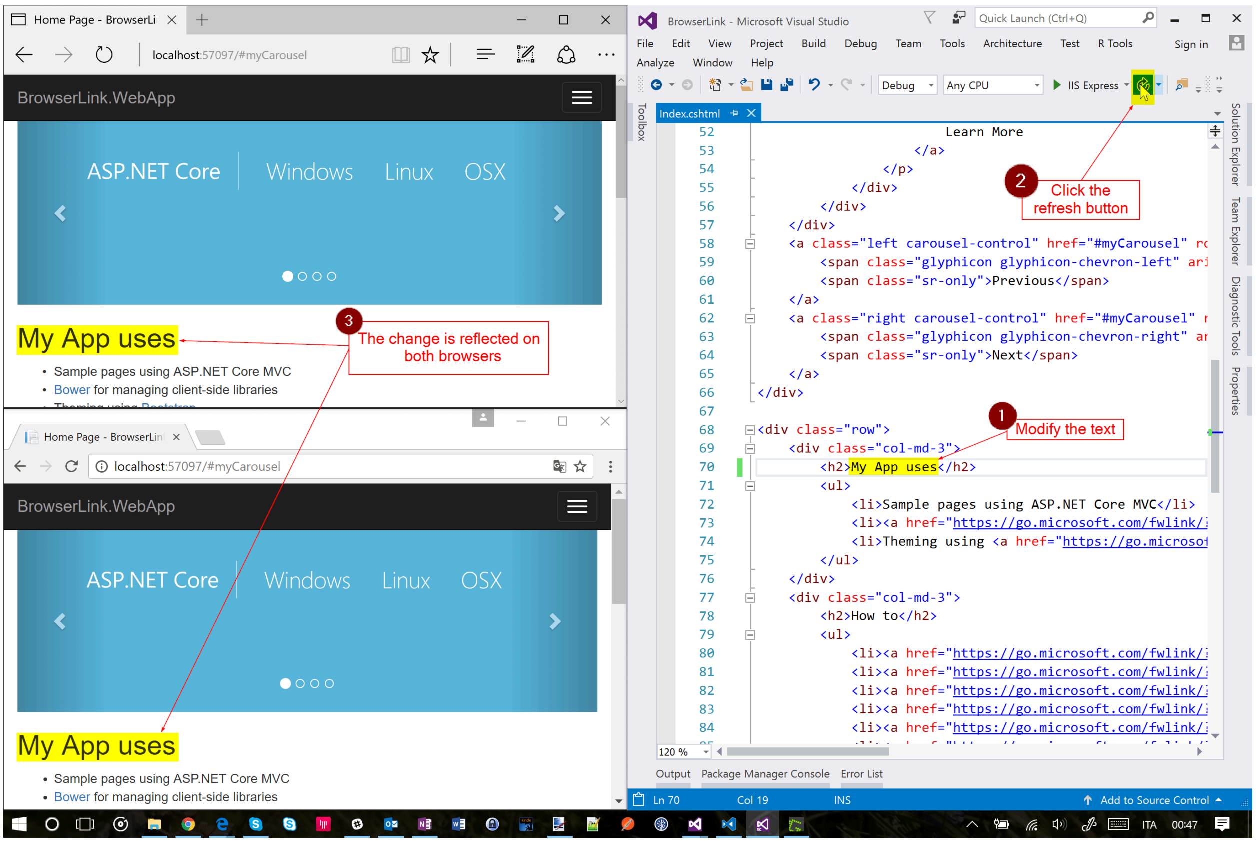
Hover over the browser link toolbar control to see the browsers that are connected to the project:


BrowserLink - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Architecture Test R Tools
Analyze Window Help
Debug
Any CPU
IIS Express
Toolbox
Index.cshtml

Browser Link - 2 browsers connected

Chrome ~/
Microsoft Edge ~/

Change the Index view, and all connected browsers are updated when you click the browser link refresh button:

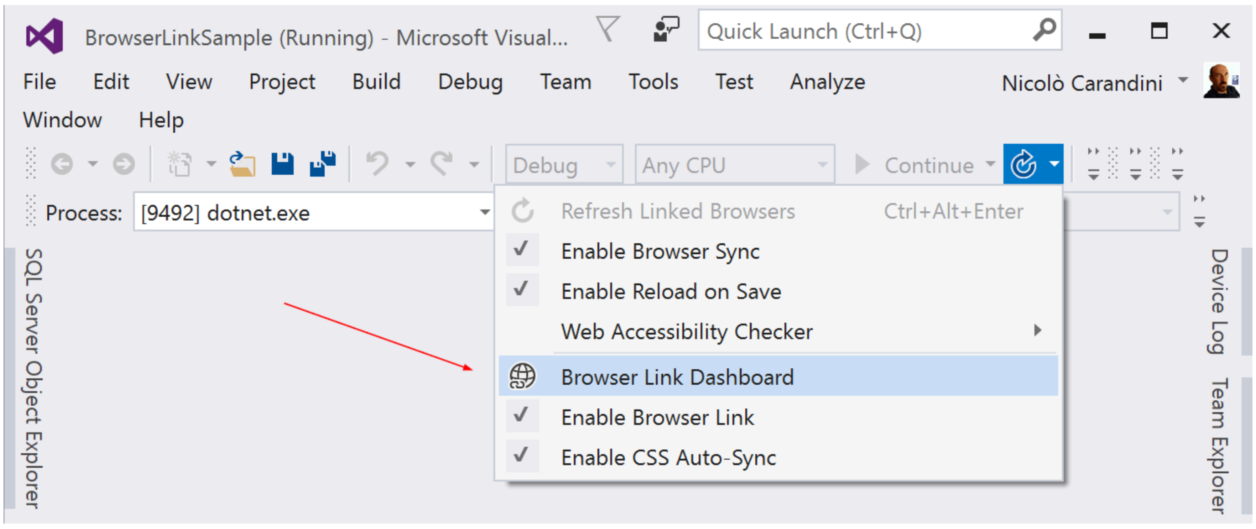

Home Page - BrowserLi
localhost:57097/#myCarousel
BrowserLink.WebApp
ASP.NET Core
Windows
Linux
OSX
My App uses
Sample pages using ASP.NET Core MVC
Bower for managing client-side libraries
3
The change is reflected on both browsers
2
Click the refresh button
1
Modify the text
BrowserLink - Microsoft Visual Studio
Index.cshtml
Ln 70
Col 19
INS
Add to Source Control

```
                    Learn More
                </a>
            </p>
        </div>
    </div>
</div>
<a class="left carousel-control" href="#myCarousel" ro
    <span class="glyphicon glyphicon-chevron-left" ari
    <span class="sr-only">Previous</span>
</a>
<a class="right carousel-control" href="#myCarousel" r
    <span class="glyphicon glyphicon-chevron-right" ar
    <span class="sr-only">Next</span>
</a>
</div>

<div class="row">
    <div class="col-md-3">
        <h2>My App uses</h2>
        <ul>
            <li>Sample pages using ASP.NET Core MVC</li>
            <li><a href="https://go.microsoft.com/fwlink/?
            <li>Theming using <a href="https://go.microsof
        </ul>
    </div>
    <div class="col-md-3">
        <h2>How to</h2>
        <ul>
            <li><a href="https://go.microsoft.com/fwlink/?
            <li><a href="https://go.microsoft.com/fwlink/?
            <li><a href="https://go.microsoft.com/fwlink/?
            <li><a href="https://go.microsoft.com/fwlink/?
            <li><a href="https://go.microsoft.com/fwlink/?
```

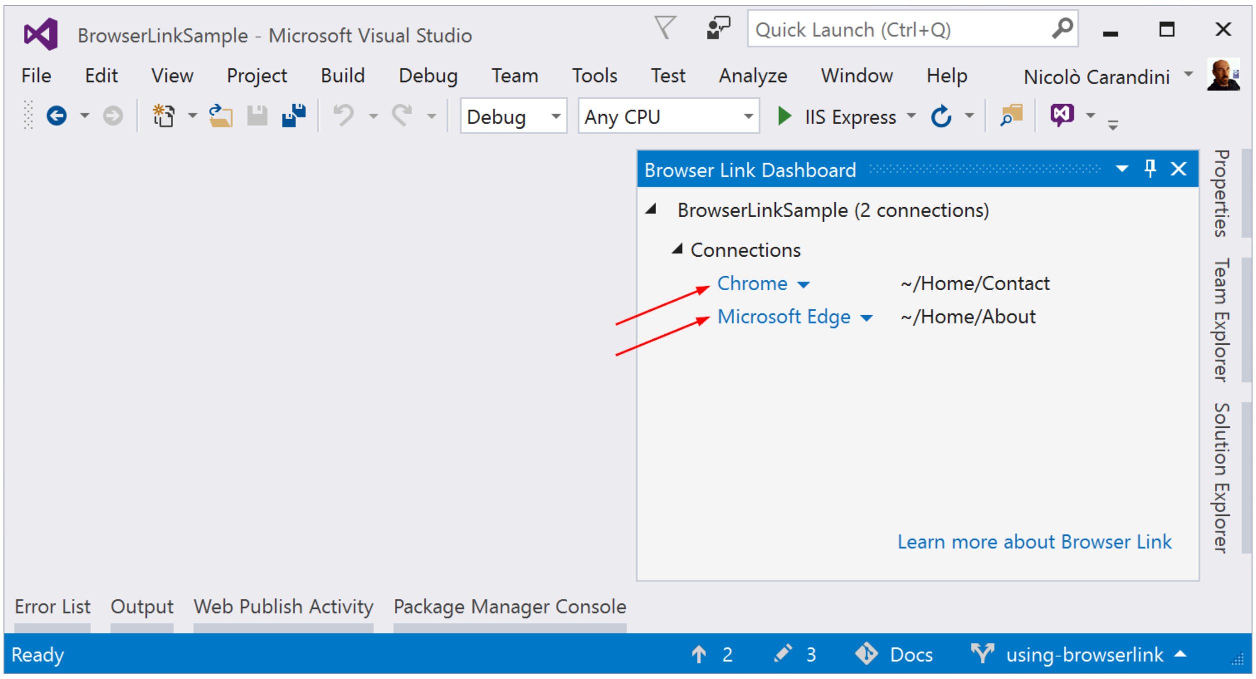
browser link also works with browsers that you launch from outside Visual Studio and navigate to the app URL.

The browser link dashboard

Open the **browser link dashboard** window from the browser link drop down menu to manage the connection with open browsers:


BrowserLinkSample (Running) - Microsoft Visual...
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze
Window Help
Nicolò Carandini
Debug
Any CPU
Continue
Process: [9492] dotnet.exe
Refresh Linked Browsers Ctrl+Alt+Enter
✓ Enable Browser Sync
✓ Enable Reload on Save
Web Accessibility Checker
Browser Link Dashboard
✓ Enable Browser Link
✓ Enable CSS Auto-Sync
SQL Server Object Explorer
Device Log
Team Explorer

The connected browsers are shown with the path to the page that each browser is showing:


BrowserLinkSample - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Nicolò Carandini
Debug
Any CPU
IIS Express
Browser Link Dashboard
Properties
Team Explorer
Solution Explorer
Error List Output Web Publish Activity Package Manager Console
Ready
2
3
Docs
using-browserlink

BrowserLinkSample (2 connections)
Connections
Chrome ~/Home/Contact
Microsoft Edge ~/Home/About

Learn more about Browser Link

You can also click on an individual browser name to refresh only that browser.

Enable or disable browser link

When you re-enable browser link after disabling it, you must refresh the browsers to reconnect them.

Enable or disable CSS hot reload

When CSS hot reload is enabled, connected browsers are automatically refreshed when you make any change to CSS files.

How it works

browser link uses SignalR to create a communication channel between Visual Studio and the browser. When browser link is enabled, Visual Studio acts as a SignalR server that multiple clients (browsers) can connect to. browser link also registers a middleware component in the ASP.NET Core request pipeline. This component injects special `<script>` references into every page request from the server. You can see the script references by selecting **View source** in the browser and scrolling to the end of the `<body>` tag content:

HTML

```html
<!-- Visual Studio browser link -->
<script type="application/json" id="__browserLink_initializationData">
```

```
{"requestId":"a717d5a07c1741949a7cefd6fa2bad08","requestMappingFromServer":f
alse}
    </script>
    <script type="text/javascript"
src="http://localhost:54139/b6e36e429d034f578ebccd6a79bf19bf/browserLink"
async="async"></script>
    <!-- End browser link -->
</body>
```

Your source files aren't modified. The middleware component injects the script references dynamically.

Because the browser-side code is all JavaScript, it works on all browsers that SignalR supports without requiring a browser plug-in.

Session and state management in ASP.NET Core

Article • 09/18/2024

By Rick Anderson, Kirk Larkin, and Diana LaRose

HTTP is a stateless protocol. By default, HTTP requests are independent messages that don't retain user values. This article describes several approaches to preserve user data between requests.

For Blazor state management guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor state management.

State management

State can be stored using several approaches. Each approach is described later in this article.

Expand table

Storage approach	Storage mechanism
Cookies	HTTP cookies. May include data stored using server-side app code.
Session state	HTTP cookies and server-side app code
TempData	HTTP cookies or session state
Query strings	HTTP query strings
Hidden fields	HTTP form fields
HttpContext.Items	Server-side app code
Cache	Server-side app code

SignalR/Blazor Server and HTTP context-based state management

SignalR apps shouldn't use session state and other state management approaches that rely upon a stable HTTP context to store information. SignalR apps can store per-connection state in Context.Items in the hub. For more information and alternative state management approaches for Blazor Server apps, see ASP.NET Core Blazor state management.

Cookies

Cookies store data across requests. Because cookies are sent with every request, their size should be kept to a minimum. Ideally, only an identifier should be stored in a cookie with the data stored by the app. Most browsers restrict cookie size to 4096 bytes. Only a limited number of cookies are available for each domain.

Because cookies are subject to tampering, they must be validated by the app. Cookies can be deleted by users and expire on clients. However, cookies are generally the most durable form of data persistence on the client.

Cookies are often used for personalization, where content is customized for a known user. The user is only identified and not authenticated in most cases. The cookie can store the user's name, account name, or unique user ID such as a GUID. The cookie can be used to access the user's personalized settings, such as their preferred website background color.

See the European Union General Data Protection Regulations (GDPR) when issuing cookies and dealing with privacy concerns. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

Session state

Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app. Session state uses a store maintained by the app to persist data across requests from a client. The session data is backed by a cache and considered ephemeral data. The site should continue to function without the session data. Critical application data should be stored in the user database and cached in session only as a performance optimization.

Session isn't supported in SignalR apps because a SignalR Hub may execute independent of an HTTP context. For example, this can occur when a long polling request is held open by a hub beyond the lifetime of the request's HTTP context.

ASP.NET Core maintains session state by providing a cookie to the client that contains a session ID. The cookie session ID:

- Is sent to the app with each request.
- Is used by the app to fetch the session data.

Session state exhibits the following behaviors:

- The session cookie is specific to the browser. Sessions aren't shared across browsers.
- Session cookies are deleted when the browser session ends.
- If a cookie is received for an expired session, a new session is created that uses the same session cookie.
- Empty sessions aren't retained. The session must have at least one value set to persist the session across requests. When a session isn't retained, a new session ID is generated for each new request.
- The app retains a session for a limited time after the last request. The app either sets the session timeout or uses the default value of 20 minutes. Session state is ideal for storing user data:
  - That's specific to a particular session.
  - Where the data doesn't require permanent storage across sessions.
- Session data is deleted either when the ISession.Clear implementation is called or when the session expires.
- There's no default mechanism to inform app code that a client browser has been closed or when the session cookie is deleted or expired on the client.
- Session state cookies aren't marked essential by default. Session state isn't functional unless tracking is permitted by the site visitor. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.
- **Note**: There is no replacement for the cookieless session feature from the ASP.NET Framework because it's considered insecure and can lead to session fixation attacks.

⚠ **Warning**

Don't store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows. A session might not be restricted to a single user. The next user might continue to browse the app with the same session cookie.

The in-memory cache provider stores session data in the memory of the server where the app resides. In a server farm scenario:

- Use *sticky sessions* to tie each session to a specific app instance on an individual server. Azure App Service uses Application Request Routing (ARR) to enforce sticky sessions by default. However, sticky sessions can affect scalability and complicate web app updates. A better approach is to use a Redis or SQL Server

distributed cache, which doesn't require sticky sessions. For more information, see Distributed caching in ASP.NET Core.
- The session cookie is encrypted via IDataProtector. Data Protection must be properly configured to read session cookies on each machine. For more information, see ASP.NET Core Data Protection Overview and Key storage providers.

Configure session state

Middleware for managing session state is included in the framework. To enable the session middleware, `Program.cs` must contain:

- Any of the IDistributedCache memory caches. The `IDistributedCache` implementation is used as a backing store for session. For more information, see Distributed caching in ASP.NET Core.
- A call to AddSession
- A call to UseSession

The following code shows how to set up the in-memory session provider with a default in-memory implementation of `IDistributedCache`:

C#

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthorization();

app.UseSession();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

The preceding code sets a short timeout to simplify testing.

The order of middleware is important. Call `UseSession` after `UseRouting` and before `MapRazorPages` and `MapDefaultControllerRoute` . See Middleware Ordering.

HttpContext.Session is available after session state is configured.

`HttpContext.Session` can't be accessed before `UseSession` has been called.

A new session with a new session cookie can't be created after the app has begun writing to the response stream. The exception is recorded in the web server log and not displayed in the browser.

Load session state asynchronously

The default session provider in ASP.NET Core loads session records from the underlying IDistributedCache backing store asynchronously only if the ISession.LoadAsync method is explicitly called before the TryGetValue, Set, or Remove methods. If `LoadAsync` isn't called first, the underlying session record is loaded synchronously, which can incur a performance penalty at scale.

To have apps enforce this pattern, wrap the DistributedSessionStore and DistributedSession implementations with versions that throw an exception if the `LoadAsync` method isn't called before `TryGetValue`, `Set`, or `Remove`. Register the wrapped versions in the services container.

Session options

To override session defaults, use SessionOptions.

Expand table

Option	Description
Cookie	Determines the settings used to create the cookie. Name defaults to SessionDefaults.CookieName (`.AspNetCore.Session`). Path defaults to SessionDefaults.CookiePath (`/`). SameSite defaults to SameSiteMode.Lax (`1`). HttpOnly defaults to `true`. IsEssential defaults to `false`.
IdleTimeout	The `IdleTimeout` indicates how long the session can be idle before its contents are abandoned. Each session access resets the timeout. This setting only applies to the content of the session, not the cookie. The default is 20 minutes.
IOTimeout	The maximum amount of time allowed to load a session from the store or to commit it back to the store. This setting may only apply to asynchronous operations. This timeout can be disabled using InfiniteTimeSpan. The default is 1 minute.

Session uses a cookie to track and identify requests from a single browser. By default, this cookie is named `.AspNetCore.Session`, and it uses a path of `/`. Because the cookie default doesn't specify a domain, it isn't made available to the client-side script on the page (because HttpOnly defaults to `true`).

To override cookie session defaults, use SessionOptions:

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

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>
{
    options.Cookie.Name = ".AdventureWorks.Session";
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.IsEssential = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
app.UseAuthorization();

app.UseSession();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

The app uses the IdleTimeout property to determine how long a session can be idle before its contents in the server's cache are abandoned. This property is independent of the cookie expiration. Each request that passes through the Session Middleware resets the timeout.

Session state is *non-locking*. If two requests simultaneously attempt to modify the contents of a session, the last request overrides the first. `Session` is implemented as a *coherent session*, which means that all the contents are stored together. When two requests seek to modify different session values, the last request may override session changes made by the first.

Set and get Session values

Session state is accessed from a Razor Pages PageModel class or MVC Controller class with HttpContext.Session. This property is an ISession implementation.

The `ISession` implementation provides several extension methods to set and retrieve integer and string values. The extension methods are in the Microsoft.AspNetCore.Http namespace.

`ISession` extension methods:

- Get(ISession, String)
- GetInt32(ISession, String)
- GetString(ISession, String)
- SetInt32(ISession, String, Int32)
- SetString(ISession, String, String)

The following example retrieves the session value for the `IndexModel.SessionKeyName` key (`_Name` in the sample app) in a Razor Pages page:

C#

```
@page
@using Microsoft.AspNetCore.Http
```

```
@model IndexModel

...

Name: @HttpContext.Session.GetString(IndexModel.SessionKeyName)
```

The following example shows how to set and get an integer and a string:

C#

```csharp
public class IndexModel : PageModel
{
    public const string SessionKeyName = "_Name";
    public const string SessionKeyAge = "_Age";

    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        if
(string.IsNullOrEmpty(HttpContext.Session.GetString(SessionKeyName)))
        {
            HttpContext.Session.SetString(SessionKeyName, "The Doctor");
            HttpContext.Session.SetInt32(SessionKeyAge, 73);
        }
        var name = HttpContext.Session.GetString(SessionKeyName);
        var age = HttpContext.Session.GetInt32(SessionKeyAge).ToString();

        _logger.LogInformation("Session Name: {Name}", name);
        _logger.LogInformation("Session Age: {Age}", age);
    }
}
```

The following markup displays the session values on a Razor Page:

CSHTML

```cshtml
@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<div class="text-center">
<p><b>Name:</b> @HttpContext.Session.GetString("_Name");<b>Age:
```

```
</b> @HttpContext.Session.GetInt32("_Age").ToString()</p>
</div>
```

All session data must be serialized to enable a distributed cache scenario, even when using the in-memory cache. String and integer serializers are provided by the extension methods of ISession. Complex types must be serialized by the user using another mechanism, such as JSON.

Use the following sample code to serialize objects:

```C#
public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonSerializer.Serialize(value));
    }

    public static T? Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return value == null ? default : JsonSerializer.Deserialize<T>
(value);
    }
}
```

The following example shows how to set and get a serializable object with the `SessionExtensions` class:

```C#
using Microsoft.AspNetCore.Mvc.RazorPages;
using Web.Extensions;    // SessionExtensions

namespace SessionSample.Pages
{
    public class Index6Model : PageModel
    {
        const string SessionKeyTime = "_Time";
        public string? SessionInfo_SessionTime { get; private set; }
        private readonly ILogger<Index6Model> _logger;

        public Index6Model(ILogger<Index6Model> logger)
        {
            _logger = logger;
        }
        public void OnGet()
        {
            var currentTime = DateTime.Now;

            // Requires SessionExtensions from sample.
            if (HttpContext.Session.Get<DateTime>(SessionKeyTime) ==
default)
            {
                HttpContext.Session.Set<DateTime>(SessionKeyTime,
currentTime);
            }
            _logger.LogInformation("Current Time: {Time}", currentTime);
            _logger.LogInformation("Session Time: {Time}",
                           HttpContext.Session.Get<DateTime>
(SessionKeyTime));

        }
    }
}
```

⚠ **Warning**

Storing a live object in the session should be used with caution, as there are many problems that can occur with serialized objects. For more information, see Sessions should be allowed to store objects (dotnet/aspnetcore #18159).

TempData

ASP.NET Core exposes the Razor Pages TempData or Controller TempData. This property stores data until it's read in another request. The Keep(String) and Peek(string) methods can be used to examine the data without deletion at the end of the request. Keep marks all items in the dictionary for retention. `TempData` is:

- Useful for redirection when data is required for more than a single request.
- Implemented by `TempData` providers using either cookies or session state.

TempData samples

Consider the following page that creates a customer:

C#

```csharp
public class CreateModel : PageModel
{
    private readonly RazorPagesContactsContext _context;
    public CreateModel(RazorPagesContactsContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";

        return RedirectToPage("./IndexPeek");
    }
}
```

The following page displays `TempData["Message"]`:

CSHTML

```
@page
@model IndexModel

<h1>Peek Contacts</h1>

@{
    if (TempData.Peek("Message") != null)
    {
        <h3>Message: @TempData.Peek("Message")</h3>
    }
}

@*Content removed for brevity.*@
```

In the preceding markup, at the end of the request, `TempData["Message"]` is **not** deleted because `Peek` is used. Refreshing the page displays the contents of `TempData["Message"]`.

The following markup is similar to the preceding code, but uses `Keep` to preserve the data at the end of the request:

CSHTML

```cshtml
@page
@model IndexModel

<h1>Contacts Keep</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
    TempData.Keep("Message");
}

@*Content removed for brevity.*@
```

Navigating between the *IndexPeek* and *IndexKeep* pages won't delete `TempData["Message"]`.

The following code displays `TempData["Message"]`, but at the end of the request, `TempData["Message"]` is deleted:

CSHTML

```cshtml
@page
@model IndexModel

<h1>Index no Keep or Peek</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
}

@*Content removed for brevity.*@
```

TempData providers

The cookie-based TempData provider is used by default to store TempData in cookies.

The cookie data is encrypted using IDataProtector, encoded with Base64UrlTextEncoder, then chunked. The maximum cookie size is less than 4096 bytes due to encryption and chunking. The cookie data isn't compressed because compressing encrypted data can lead to security problems such as the CRIME and BREACH attacks. For more information on the cookie-based TempData provider, see CookieTempDataProvider.

Choose a TempData provider

Choosing a TempData provider involves several considerations, such as:

- Does the app already use session state? If so, using the session state TempData provider has no additional cost to the app beyond the size of the data.
- Does the app use TempData only sparingly for relatively small amounts of data, up to 500 bytes? If so, the cookie TempData provider adds a small cost to each request that carries TempData. If not, the session state TempData provider can be beneficial to avoid round-tripping a large amount of data in each request until the TempData is consumed.
- Does the app run in a server farm on multiple servers? If so, there's no additional configuration required to use the cookie TempData provider outside of Data Protection. For more information, see ASP.NET Core Data Protection Overview and Key storage providers.

Most web clients such as web browsers enforce limits on the maximum size of each cookie and the total number of cookies. When using the cookie TempData provider, verify the app won't exceed these limits . Consider the total size of the data. Account for increases in cookie size due to encryption and chunking.

Configure the TempData provider

The cookie-based TempData provider is enabled by default.

To enable the session-based TempData provider, use the AddSessionStateTempDataProvider extension method. Only one call to `AddSessionStateTempDataProvider` is required:

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

```
builder.Services.AddRazorPages()
                    .AddSessionStateTempDataProvider();
builder.Services.AddControllersWithViews()
                    .AddSessionStateTempDataProvider();

builder.Services.AddSession();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseSession();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Query strings

A limited amount of data can be passed from one request to another by adding it to the new request's query string. This is useful for capturing state in a persistent manner that allows links with embedded state to be shared through email or social networks. Because URL query strings are public, never use query strings for sensitive data.

In addition to unintended sharing, including data in query strings can expose the app to Cross-Site Request Forgery (CSRF) attacks. Any preserved session state must protect against CSRF attacks. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Hidden fields

Data can be saved in hidden form fields and posted back on the next request. This is common in multi-page forms. Because the client can potentially tamper with the data, the app must always revalidate the data stored in hidden fields.

`HttpContext.Items`

The HttpContext.Items collection is used to store data while processing a single request. The collection's contents are discarded after a request is processed. The `Items` collection is often used to allow components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters.

In the following example, middleware adds `isVerified` to the `Items` collection:

C#

```csharp
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

ILogger logger = app.Logger;

app.Use(async (context, next) =>
{
    // context.Items["isVerified"] is null
    logger.LogInformation($"Before setting: Verified:
{context.Items["isVerified"]}");
    context.Items["isVerified"] = true;
    await next.Invoke();
});

app.Use(async (context, next) =>
{
    // context.Items["isVerified"] is true
    logger.LogInformation($"Next: Verified: {context.Items["isVerified"]}");
    await next.Invoke();
});

app.MapGet("/", async context =>
{
    await context.Response.WriteAsync($"Verified:
{context.Items["isVerified"]}");
});

app.Run();
```

For middleware that's only used in a single app, it's unlikely that using a fixed `string` key would cause a key collision. However, to avoid the possibility of a key collision altogether, an `object` can be used as an item key. This approach is particularly useful for middleware that's shared between apps and also has the advantage of eliminating the use of key strings in the code. The following example shows how to use an `object` key defined in a middleware class:

C#

```csharp
public class HttpContextItemsMiddleware
{
    private readonly RequestDelegate _next;
    public static readonly object HttpContextItemsMiddlewareKey = new();

    public HttpContextItemsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";

        await _next(httpContext);
    }
}

public static class HttpContextItemsMiddlewareExtensions
{
    public static IApplicationBuilder
        UseHttpContextItemsMiddleware(this IApplicationBuilder app)
    {
        return app.UseMiddleware<HttpContextItemsMiddleware>();
    }
}
```

Other code can access the value stored in `HttpContext.Items` using the key exposed by the middleware class:

C#

```csharp
public class Index2Model : PageModel
{
    private readonly ILogger<Index2Model> _logger;

    public Index2Model(ILogger<Index2Model> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        HttpContext.Items

.TryGetValue(HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey,
                out var middlewareSetValue);

        _logger.LogInformation("Middleware value {MV}",
            middlewareSetValue?.ToString() ?? "Middleware value not set!");
    }
}
```

Cache

Caching is an efficient way to store and retrieve data. The app can control the lifetime of cached items. For more information, see Response caching in ASP.NET Core.

Cached data isn't associated with a specific request, user, or session. **Do not cache user-specific data that may be retrieved by other user requests.**

To cache application wide data, see Cache in-memory in ASP.NET Core.

Checking session state

ISession.IsAvailable is intended to check for transient failures. Calling `IsAvailable` before the session middleware runs throws an `InvalidOperationException`.

Libraries that need to test session availability can use `HttpContext.Features.Get<ISessionFeature>()?.Session != null`.

Common errors

- "Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

  This is typically caused by failing to configure at least one `IDistributedCache` implementation. For more information, see Distributed caching in ASP.NET Core and Cache in-memory in ASP.NET Core.

If the session middleware fails to persist a session:

- The middleware logs the exception and the request continues normally.
- This leads to unpredictable behavior.

The session middleware can fail to persist a session if the backing store isn't available. For example, a user stores a shopping cart in session. The user adds an item to the cart but the commit fails. The app doesn't know about the failure so it reports to the user that the item was added to their cart, which isn't true.

The recommended approach to check for errors is to call `await feature.Session.CommitAsync` when the app is done writing to the session. CommitAsync throws an exception if the backing store is unavailable. If `CommitAsync` fails, the app can process the exception. LoadAsync throws under the same conditions when the data store is unavailable.

Additional resources

View or download sample code (how to download)

Host ASP.NET Core in a web farm

Layout in ASP.NET Core

Article • 06/03/2022

By Steve Smith and Dave Brock

Pages and views frequently share visual and programmatic elements. This article demonstrates how to:

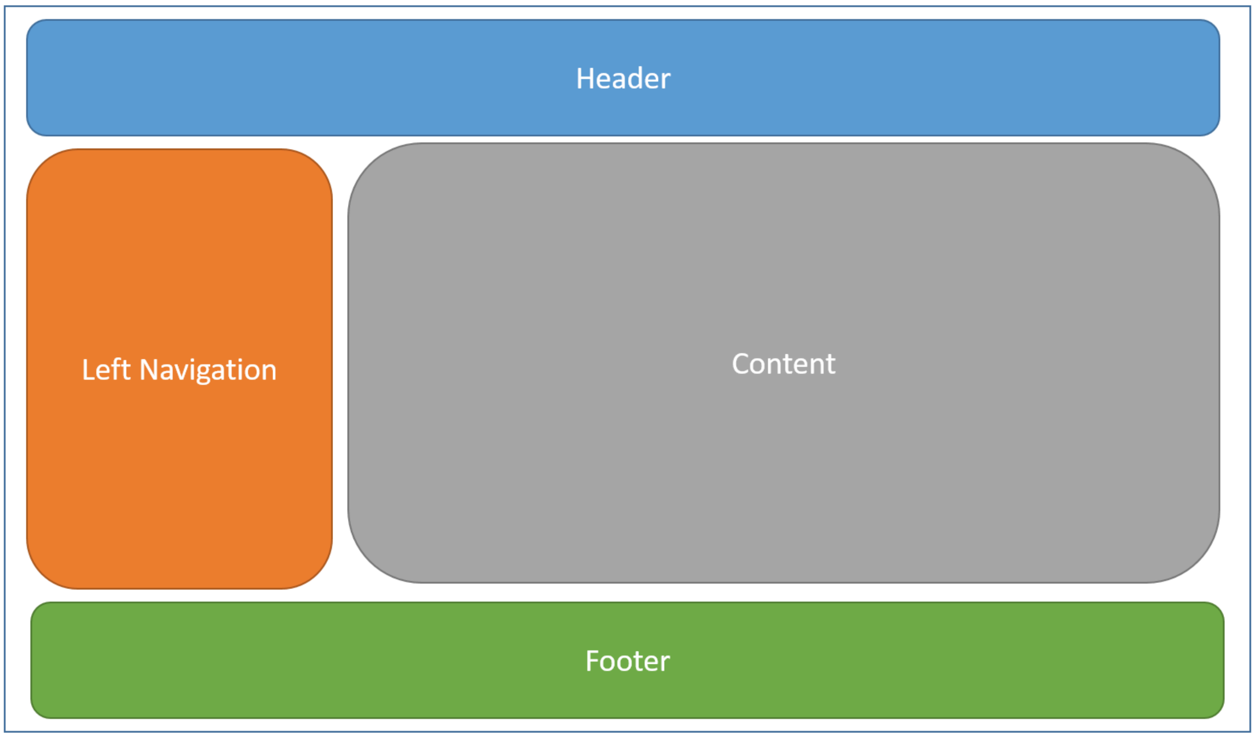
- Use common layouts.
- Share directives.
- Run common code before rendering pages or views.

This document discusses layouts for the two different approaches to ASP.NET Core MVC: Razor Pages and controllers with views. For this topic, the differences are minimal:

- Razor Pages are in the *Pages* folder.
- Controllers with views uses a *Views* folder for views.

What is a Layout

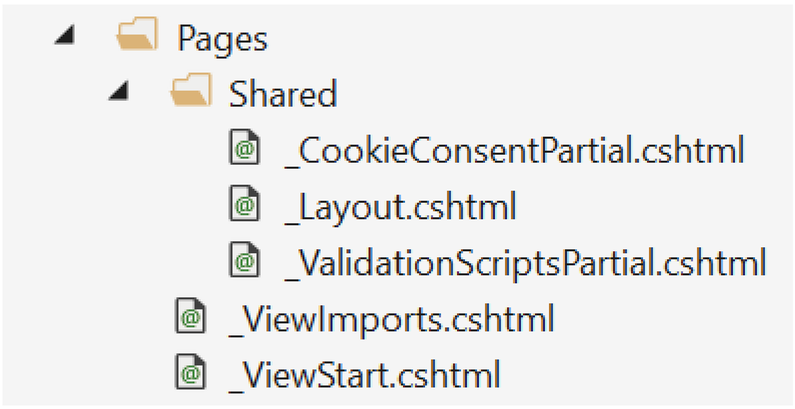
Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.



Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a *layout* file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views.

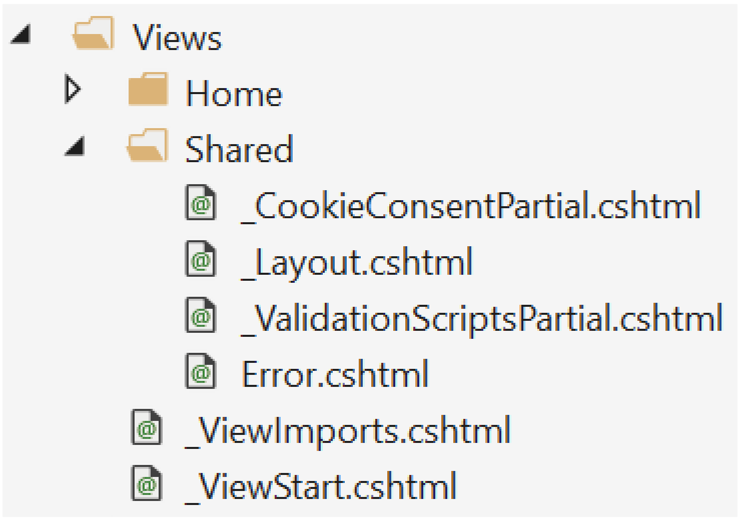
By convention, the default layout for an ASP.NET Core app is named `_Layout.cshtml`. The layout files for new ASP.NET Core projects created with the templates are:

- Razor Pages: `Pages/Shared/_Layout.cshtml`



Pages
Shared
_CookieConsentPartial.cshtml
_Layout.cshtml
_ValidationScriptsPartial.cshtml
_ViewImports.cshtml
_ViewStart.cshtml

- Controller with views: `Views/Shared/_Layout.cshtml`



Views
Home
Shared
_CookieConsentPartial.cshtml
_Layout.cshtml
_ValidationScriptsPartial.cshtml
Error.cshtml
_ViewImports.cshtml
_ViewStart.cshtml

The layout defines a top level template for views in the app. Apps don't require a layout. Apps can define more than one layout, with different views specifying different layouts.

The following code shows the layout file for a template created project with a controller and views:

CSHTML

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
/>
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-
property="position" asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-
version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-
brand">WebApplication1</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Contact">Contact</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - WebApplication1</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-
3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-
tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">
        </script>
        <script
src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn &&
window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-
Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

Specifying a Layout

Razor views have a `Layout` property. Individual views specify a layout by setting this property:

```cshtml
@{
    Layout = "_Layout";
}
```

The layout specified can use a full path (for example, `/Pages/Shared/_Layout.cshtml` or `/Views/Shared/_Layout.cshtml`) or a partial name (example: `_Layout`). When a partial name is provided, the Razor view engine searches for the layout file using its standard discovery process. The folder where the handler method (or controller) exists is searched first, followed by the *Shared* folder. This discovery process is identical to the process used to discover partial views.

By default, every layout must call `RenderBody`. Wherever the call to `RenderBody` is placed, the contents of the view will be rendered.

Sections

A layout can optionally reference one or more *sections*, by calling `RenderSection`. Sections provide a way to organize where certain page elements should be placed. Each call to `RenderSection` can specify whether that section is required or optional:

HTML

```html
<script type="text/javascript" src="~/scripts/global.js"></script>

@RenderSection("Scripts", required: false)
```

If a required section isn't found, an exception is thrown. Individual views specify the content to be rendered within a section using the `@section` Razor syntax. If a page or view defines a section, it must be rendered (or an error will occur).

An example `@section` definition in Razor Pages view:

HTML

```html
@section Scripts {
     <script type="text/javascript" src="~/scripts/main.js"></script>
}
```

In the preceding code, `scripts/main.js` is added to the `scripts` section on a page or view. Other pages or views in the same app might not require this script and wouldn't define a scripts section.

The following markup uses the Partial Tag Helper to render `_ValidationScriptsPartial.cshtml`:

HTML

```html
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

The preceding markup was generated by scaffolding Identity.

Sections defined in a page or view are available only in its immediate layout page. They cannot be referenced from partials, view components, or other parts of the view system.

Ignoring sections

By default, the body and all sections in a content page must all be rendered by the layout page. The Razor view engine enforces this by tracking whether the body and each section have been rendered.

To instruct the view engine to ignore the body or sections, call the `IgnoreBody` and `IgnoreSection` methods.

The body and every section in a Razor page must be either rendered or ignored.

Importing Shared Directives

Views and pages can use Razor directives to import namespaces and use dependency injection. Directives shared by many views may be specified in a common `_ViewImports.cshtml` file. The `_ViewImports` file supports the following directives:

- `@addTagHelper`
- `@removeTagHelper`
- `@tagHelperPrefix`
- `@using`
- `@model`
- `@inherits`
- `@inject`
- `@namespace`

The file doesn't support other Razor features, such as functions and section definitions.

A sample `_ViewImports.cshtml` file:

CSHTML

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The `_ViewImports.cshtml` file for an ASP.NET Core MVC app is typically placed in the *Pages* (or *Views*) folder. A `_ViewImports.cshtml` file can be placed within any folder, in which case it will only be applied to pages or views within that folder and its subfolders. `_ViewImports` files are processed starting at the root level and then for each folder leading up to the location of the page or view itself. `_ViewImports` settings specified at the root level may be overridden at the folder level.

For example, suppose:

- The root level `_ViewImports.cshtml` file includes `@model MyModel1` and `@addTagHelper *, MyTagHelper1`.
- A subfolder `_ViewImports.cshtml` file includes `@model MyModel2` and `@addTagHelper *, MyTagHelper2`.

Pages and views in the subfolder will have access to both Tag Helpers and the `MyModel2` model.

If multiple `_ViewImports.cshtml` files are found in the file hierarchy, the combined behavior of the directives are:

- `@addTagHelper`, `@removeTagHelper`: all run, in order
- `@tagHelperPrefix`: the closest one to the view overrides any others
- `@model`: the closest one to the view overrides any others
- `@inherits`: the closest one to the view overrides any others
- `@using`: all are included; duplicates are ignored
- `@inject`: for each property, the closest one to the view overrides any others with the same property name

Running Code Before Each View

Code that needs to run before each view or page should be placed in the `_ViewStart.cshtml` file. By convention, the `_ViewStart.cshtml` file is located in the *Pages* (or *Views*) folder. The statements listed in `_ViewStart.cshtml` are run before every full view (not layouts, and not partial views). Like ViewImports.cshtml, `_ViewStart.cshtml` is hierarchical. If a `_ViewStart.cshtml` file is defined in the view or pages folder, it will be run after the one defined in the root of the *Pages* (or *Views*) folder (if any).

A sample `_ViewStart.cshtml` file:

```cshtml
@{
    Layout = "_Layout";
}
```

The file above specifies that all views will use the `_Layout.cshtml` layout.

`_ViewStart.cshtml` and `_ViewImports.cshtml` are **not** typically placed in the */Pages/Shared* (or */Views/Shared*) folder. The app-level versions of these files should be placed directly in the */Pages* (or */Views*) folder.

Razor syntax reference for ASP.NET Core

Article • 09/27/2024

By Rick Anderson, Taylor Mullen, and Dan Vicarel

Razor is a markup syntax for embedding .NET based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a `.cshtml` file extension. Razor is also found in Razor component files (`.razor`). Razor syntax is similar to the templating engines of various JavaScript single-page application (SPA) frameworks, such as Angular, React, VueJs, and Svelte. For more information see, The features described in this article are obsolete as of ASP.NET Core 3.0.

Introduction to ASP.NET Web Programming Using the Razor Syntax provides many samples of programming with Razor syntax. Although the topic was written for ASP.NET rather than ASP.NET Core, most of the samples apply to ASP.NET Core.

Rendering HTML

The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in `.cshtml` Razor files is rendered by the server unchanged.

Razor syntax

Razor supports C# and uses the `@` symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output.

When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup. Otherwise, it transitions into plain HTML.

To escape an `@` symbol in Razor markup, use a second `@` symbol:

CSHTML

```
<p>@@Username</p>
```

The code is rendered in HTML with a single `@` symbol:

HTML

```
<p>@Username</p>
```

HTML attributes and content containing email addresses don't treat the `@` symbol as a transition character. The email addresses in the following example are untouched by Razor parsing:

CSHTML

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Scalable Vector Graphics (SVG)

SVG foreignObject elements are supported:

HTML

```
@{
    string message = "foreignObject example with Scalable Vector Graphics
(SVG)";
}

<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
    <rect x="0" y="0" rx="10" ry="10" width="200" height="200"
stroke="black"
        fill="none" />
    <foreignObject x="20" y="20" width="160" height="160">
        <p>@message</p>
    </foreignObject>
</svg>
```

Implicit Razor expressions

Implicit Razor expressions start with `@` followed by C# code:

CSHTML

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

With the exception of the C# `await` keyword, implicit expressions must not contain spaces. If the C# statement has a clear ending, spaces can be intermingled:

CSHTML

```
<p>@await DoSomething("hello", "world")</p>
```

Implicit expressions **cannot** contain C# generics, as the characters inside the brackets (`<>`) are interpreted as an HTML tag. The following code is **not** valid:

CSHTML

```
<p>@GenericMethod<int>()</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element wasn't closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?`

Generic method calls must be wrapped in an explicit Razor expression or a Razor code block.

Explicit Razor expressions

Explicit Razor expressions consist of an `@` symbol with balanced parenthesis. To render last week's time, the following Razor markup is used:

CSHTML

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Any content within the `@()` parenthesis is evaluated and rendered to the output.

Implicit expressions, described in the previous section, generally can't contain spaces. In the following code, one week isn't subtracted from the current time:

CSHTML

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

The code renders the following HTML:

HTML

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

Explicit expressions can be used to concatenate text with an expression result:

CSHTML

```
@{
    var joe = new Person("Joe", 33);
}

<p>Age@(joe.Age)</p>
```

Without the explicit expression, `<p>Age@joe.Age</p>` is treated as an email address, and `<p>Age@joe.Age</p>` is rendered. When written as an explicit expression, `<p>Age33</p>` is rendered.

Explicit expressions can be used to render output from generic methods in `.cshtml` files. The following markup shows how to correct the error shown earlier caused by the brackets of a C# generic. The code is written as an explicit expression:

CSHTML

```
<p>@(GenericMethod<int>())</p>
```

Expression encoding

C# expressions that evaluate to a string are HTML encoded. C# expressions that evaluate to `IHtmlContent` are rendered directly through `IHtmlContent.WriteTo`. C# expressions that don't evaluate to `IHtmlContent` are converted to a string by `ToString` and encoded before they're rendered.

CSHTML

```
@("<span>Hello World</span>")
```

The preceding code renders the following HTML:

HTML

```
&lt;span&gt;Hello World&lt;/span&gt;
```

The HTML is shown in the browser as plain text:

<span>Hello World</span>

`HtmlHelper.Raw` output isn't encoded but rendered as HTML markup.

⚠ **Warning**

Using `HtmlHelper.Raw` on unsanitized user input is a security risk. User input might contain malicious JavaScript or other exploits. Sanitizing user input is difficult. Avoid using `HtmlHelper.Raw` with user input.

CSHTML

```
@Html.Raw("<span>Hello World</span>")
```

The code renders the following HTML:

HTML

```
<span>Hello World</span>
```

Razor code blocks

Razor code blocks start with `@` and are enclosed by `{}`. Unlike expressions, C# code inside code blocks isn't rendered. Code blocks and expressions in a view share the same scope and are defined in order:

CSHTML

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin
Luther King, Jr.";
}

<p>@quote</p>
```

The code renders the following HTML:

HTML

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King,
Jr.</p>
```

In code blocks, declare local functions with markup to serve as templating methods:

CSHTML

```
@{
    void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }

    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}
```

The code renders the following HTML:

HTML

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

Implicit transitions

The default language in a code block is C#, but the Razor Page can transition back to HTML:

CSHTML

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Explicit delimited transition

To define a subsection of a code block that should render HTML, surround the characters for rendering with the Razor `<text>` tag:

CSHTML

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

Use this approach to render HTML that isn't surrounded by an HTML tag. Without an HTML or Razor tag, a Razor runtime error occurs.

The `<text>` tag is useful to control whitespace when rendering content:

- Only the content between the `<text>` tag is rendered.
- No whitespace before or after the `<text>` tag appears in the HTML output.

Explicit line transition

To render the rest of an entire line as HTML inside a code block, use `@:` syntax:

CSHTML

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

Without the `@:` in the code, a Razor runtime error is generated.

Extra `@` characters in a Razor file can cause compiler errors at statements later in the block. These extra `@` compiler errors:

- Can be difficult to understand because the actual error occurs before the reported error.
- Is common after combining multiple implicit and explicit expressions into a single code block.

Conditional attribute rendering

Razor automatically omits attributes that aren't needed. If the value passed in is `null` or `false`, the attribute isn't rendered.

For example, consider the following razor:

CSHTML

```cshtml
<div class="@false">False</div>
<div class="@null">Null</div>
<div class="@("")">Empty</div>
<div class="@("false")">False String</div>
<div class="@("active")">String</div>
<input type="checkbox" checked="@true" name="true" />
<input type="checkbox" checked="@false" name="false" />
<input type="checkbox" checked="@null" name="null" />
```

The preceding Razor markup generates the following HTML:

HTML

```html
<div>False</div>
<div>Null</div>
<div class="">Empty</div>
<div class="false">False String</div>
<div class="active">String</div>
<input type="checkbox" checked="checked" name="true">
<input type="checkbox" name="false">
<input type="checkbox" name="null">
```

Control structures

Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures:

Conditionals `@if, else if, else, and @switch`

`@if` controls when code runs:

CSHTML

```cshtml
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

`else` and `else if` don't require the `@` symbol:

CSHTML

```cshtml
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

The following markup shows how to use a switch statement:

CSHTML

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

Looping `@for, @foreach, @while, and @do while`

Templated HTML can be rendered with looping control statements. To render a list of people:

CSHTML

```
@{
    var people = new Person[]
    {
          new Person("Weston", 33),
          new Person("Johnathon", 41),
          ...
    };
}
```

The following looping statements are supported:

`@for`

CSHTML

```cshtml
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@foreach`

CSHTML

```cshtml
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@while`

CSHTML

```cshtml
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

`@do while`

CSHTML

```cshtml
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

Compound `@using`

In C#, a `using` statement is used to ensure an object is disposed. In Razor, the same mechanism is used to create HTML Helpers that contain additional content. In the following code, HTML Helpers render a `<form>` tag with the `@using` statement:

CSHTML

```
@using (Html.BeginForm())
{
    <div>
        <label>Email: <input type="email" id="Email" value=""></label>
        <button>Register</button>
    </div>
}
```

`@try, catch, finally`

Exception handling is similar to C#:

CSHTML

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

`@lock`

Razor has the capability to protect critical sections with lock statements:

CSHTML

```
@lock (SomeLock)
{
    // Do critical section work
}
```

Comments

Razor supports C# and HTML comments:

CSHTML

```cshtml
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
```

The code renders the following HTML:

HTML

```html
<!-- HTML comment -->
```

Razor comments are removed by the server before the webpage is rendered. Razor uses `@* *@` to delimit comments. The following code is commented out, so the server doesn't render any markup:

CSHTML

```cshtml
@*
    @{
        /* C# comment */
        // Another C# comment
    }
    <!-- HTML comment -->
*@
```

Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive typically changes the way a view is compiled or functions.

Understanding how Razor generates code for a view makes it easier to understand how directives work.

CSHTML

```cshtml
@{
    var quote = "Getting old ain't for wimps! - Anonymous";
}

<div>Quote of the Day: @quote</div>
```

The code generates a class similar to the following:

C#

```csharp
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

Later in this article, the section Inspect the Razor C# class generated for a view explains how to view this generated class.

`@attribute`

The `@attribute` directive adds the given attribute to the class of the generated page or view. The following example adds the `[Authorize]` attribute:

CSHTML

```cshtml
@attribute [Authorize]
```

The `@attribute` directive can also be used to supply a constant-based route template in a Razor component. In the following example, the `@page` directive in a component is replaced with the `@attribute` directive and the constant-based route template in `Constants.CounterRoute`, which is set elsewhere in the app to "`/counter`":

diff

```diff
- @page "/counter"
+ @attribute [Route(Constants.CounterRoute)]
```

`@code`

This scenario only applies to Razor components (`.razor`).

The `@code` block enables a Razor component to add C# members (fields, properties, and methods) to a component:

razor

```razor
@code {
    // C# members (fields, properties, and methods)
}
```

For Razor components, `@code` is an alias of @functions and recommended over `@functions`. More than one `@code` block is permissible.

`@functions`

The `@functions` directive enables adding C# members (fields, properties, and methods) to the generated class:

CSHTML

```cshtml
@functions {
    // C# members (fields, properties, and methods)
}
```

In Razor components, use `@code` over `@functions` to add C# members.

For example:

CSHTML

```cshtml
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

The code generates the following HTML markup:

HTML

```html
<div>From method: Hello</div>
```

The following code is the generated Razor C# class:

C#

```csharp
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Razor;

public class _Views_Home_Test_cshtml : RazorPage<dynamic>
{
    // Functions placed between here
    public string GetHello()
    {
        return "Hello";
    }
    // And here.
#pragma warning disable 1998
    public override async Task ExecuteAsync()
    {
        WriteLiteral("\r\n<div>From method: ");
        Write(GetHello());
        WriteLiteral("</div>\r\n");
    }
#pragma warning restore 1998
```

`@functions` methods serve as templating methods when they have markup:

CSHTML

```cshtml
@{
    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}

@functions {
    private void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }
}
```

The code renders the following HTML:

HTML

```html
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

`@implements`

The `@implements` directive implements an interface for the generated class.

The following example implements System.IDisposable so that the Dispose method can be called:

CSHTML

```cshtml
@implements IDisposable

<h1>Example</h1>

@functions {
    private bool _isDisposed;

    ...

    public void Dispose() => _isDisposed = true;
}
```

`@inherits`

The `@inherits` directive provides full control of the class the view inherits:

CSHTML

```cshtml
@inherits TypeNameOfClassToInheritFrom
```

The following code is a custom Razor page type:

C#

```csharp
using Microsoft.AspNetCore.Mvc.Razor;

public abstract class CustomRazorPage<TModel> : RazorPage<TModel>
{
    public string CustomText { get; } =
        "Gardyloo! - A Scottish warning yelled from a window before dumping"
+
        "a slop bucket on the street below.";
}
```

The `CustomText` is displayed in a view:

CSHTML

```cshtml
@inherits CustomRazorPage<TModel>
```

```
<div>Custom text: @CustomText</div>
```

The code renders the following HTML:

HTML

```html
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before
dumping
    a slop bucket on the street below.
</div>
```

`@model` and `@inherits` can be used in the same view. `@inherits` can be in a `_ViewImports.cshtml` file that the view imports:

CSHTML

```cshtml
@inherits CustomRazorPage<TModel>
```

The following code is an example of a strongly-typed view:

CSHTML

```cshtml
@inherits CustomRazorPage<TModel>

<div>The Login Email: @Model.Email</div>
<div>Custom text: @CustomText</div>
```

If "rick@contoso.com" is passed in the model, the view generates the following HTML markup:

HTML

```html
<div>The Login Email: rick@contoso.com</div>
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before
dumping
    a slop bucket on the street below.
</div>
```

`@inject`

The `@inject` directive enables the Razor Page to inject a service from the service container into a view. For more information, see Dependency injection into views.

`@layout`

This scenario only applies to Razor components (`.razor`).

The `@layout` directive specifies a layout for routable Razor components that have an @page directive. Layout components are used to avoid code duplication and inconsistency. For more information, see ASP.NET Core Blazor layouts.

`@model`

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

The `@model` directive specifies the type of the model passed to a view or page:

CSHTML

```
@model TypeNameOfModel
```

In an ASP.NET Core MVC or Razor Pages app created with individual user accounts, `Views/Account/Login.cshtml` contains the following model declaration:

CSHTML

```
@model LoginViewModel
```

The class generated inherits from `RazorPage<LoginViewModel>`:

C#

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor exposes a `Model` property for accessing the model passed to the view:

CSHTML

```
<div>The Login Email: @Model.Email</div>
```

The `@model` directive specifies the type of the `Model` property. The directive specifies the `T` in `RazorPage<T>` that the generated class that the view derives from. If the `@model` directive isn't specified, the `Model` property is of type `dynamic`. For more information, see Strongly typed models and the @model keyword.

`@namespace`

The `@namespace` directive:

- Sets the namespace of the class of the generated Razor page, MVC view, or Razor component.
- Sets the root derived namespaces of a pages, views, or components classes from the closest imports file in the directory tree, `_ViewImports.cshtml` (views or pages) or `_Imports.razor` (Razor components).

CSHTML

```
@namespace Your.Namespace.Here
```

For the Razor Pages example shown in the following table:

- Each page imports `Pages/_ViewImports.cshtml`.
- `Pages/_ViewImports.cshtml` contains `@namespace Hello.World`.
- Each page has `Hello.World` as the root of it's namespace.

Expand table

Page	Namespace
`Pages/Index.cshtml`	`Hello.World`
`Pages/MorePages/Page.cshtml`	`Hello.World.MorePages`
`Pages/MorePages/EvenMorePages/Page.cshtml`	`Hello.World.MorePages.EvenMorePages`

The preceding relationships apply to import files used with MVC views and Razor components.

When multiple import files have a `@namespace` directive, the file closest to the page, view, or component in the directory tree is used to set the root namespace.

If the `EvenMorePages` folder in the preceding example has an imports file with `@namespace Another.Planet` (or the `Pages/MorePages/EvenMorePages/Page.cshtml` file contains `@namespace Another.Planet`), the result is shown in the following table.

Expand table

Page	Namespace
`Pages/Index.cshtml`	`Hello.World`
`Pages/MorePages/Page.cshtml`	`Hello.World.MorePages`
`Pages/MorePages/EvenMorePages/Page.cshtml`	`Another.Planet`

`@page`

The `@page` directive has different effects depending on the type of the file where it appears. The directive:

- In a `.cshtml` file indicates that the file is a Razor Page. For more information, see Custom routes and Introduction to Razor Pages in ASP.NET Core.
- Specifies that a Razor component should handle requests directly. For more information, see ASP.NET Core Blazor routing and navigation.

`@preservewhitespace`

This scenario only applies to Razor components (`.razor`).

When set to `false` (default), whitespace in the rendered markup from Razor components (`.razor`) is removed if:

- Leading or trailing within an element.
- Leading or trailing within a `RenderFragment` parameter. For example, child content passed to another component.
- It precedes or follows a C# code block, such as `@if` or `@foreach`.

`@rendermode`

This scenario only applies to Razor components (`.razor`).

Sets the render mode of a Razor component:

- `InteractiveServer`: Applies interactive server rendering using Blazor Server.
- `InteractiveWebAssembly`: Applies interactive WebAssembly rendering using Blazor WebAssembly.
- `InteractiveAuto`: Initially applies interactive WebAssembly rendering using Blazor Server, and then applies interactive WebAssembly rendering using WebAssembly on subsequent visits after the Blazor bundle is downloaded.

For a component instance:

```razor
<... @rendermode="InteractiveServer" />
```

In the component definition:

```razor
@rendermode InteractiveServer
```

> ⓘ **Note**
>
> Blazor templates include a static `using` directive for RenderMode in the app's `_Imports` file (`Components/_Imports.razor`) for shorter `@rendermode` syntax:
>
> ```razor
> @using static Microsoft.AspNetCore.Components.Web.RenderMode
> ```
>
> Without the preceding directive, components must specify the static RenderMode class in `@rendermode` syntax explicitly:
>
> ```razor
> <Dialog @rendermode="RenderMode.InteractiveServer" />
> ```

For more information, including guidance on disabling prerendering with the directive/directive attribute, see ASP.NET Core Blazor render modes.

`@section`

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

The `@section` directive is used in conjunction with MVC and Razor Pages layouts to enable views or pages to render content in different parts of the HTML page. For more information, see Layout in ASP.NET Core.

`@typeparam`

This scenario only applies to Razor components (`.razor`).

The `@typeparam` directive declares a generic type parameter for the generated component class:

```razor
@typeparam TEntity
```

Generic types with where type constraints are supported:

```razor
@typeparam TEntity where TEntity : IEntity
```

For more information, see the following articles:

- ASP.NET Core Razor component generic type support
- ASP.NET Core Blazor templated components

`@using`

The `@using` directive adds the C# `using` directive to the generated view:

```CSHTML
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

In Razor components, `@using` also controls which components are in scope.

Directive attributes

Razor directive attributes are represented by implicit expressions with reserved keywords following the `@` symbol. A directive attribute typically changes the way an element is compiled or functions.

`@attributes`

This scenario only applies to Razor components (`.razor`).

`@attributes` allows a component to render non-declared attributes. For more information, see ASP.NET Core Blazor attribute splatting and arbitrary parameters.

`@bind`

This scenario only applies to Razor components (`.razor`).

Data binding in components is accomplished with the `@bind` attribute. For more information, see ASP.NET Core Blazor data binding.

`@bind:culture`

This scenario only applies to Razor components (`.razor`).

Use the `@bind:culture` attribute with the @bind attribute to provide a System.Globalization.CultureInfo for parsing and formatting a value. For more information, see ASP.NET Core Blazor globalization and localization.

`@formname`

This scenario only applies to Razor components (`.razor`).

`@formname` assigns a form name to a Razor component's plain HTML form or a form based on EditForm (Editform documentation). The value of `@formname` should be unique, which prevents form collisions in the following situations:

- A form is placed in a component with multiple forms.
- A form is sourced from an external class library, commonly a NuGet package, for a component with multiple forms, and the app author doesn't control the source code of the library to set a different external form name than a name used by another form in the component.

For more information and examples, see ASP.NET Core Blazor forms overview.

`@on{EVENT}`

This scenario only applies to Razor components (`.razor`).

Razor provides event handling features for components. For more information, see ASP.NET Core Blazor event handling.

`@on{EVENT}:preventDefault`

This scenario only applies to Razor components (`.razor`).

Prevents the default action for the event.

`@on{EVENT}:stopPropagation`

This scenario only applies to Razor components (`.razor`).

Stops event propagation for the event.

`@key`

This scenario only applies to Razor components (`.razor`).

The `@key` directive attribute causes the components diffing algorithm to guarantee preservation of elements or components based on the key's value. For more information, see Retain element, component, and model relationships in ASP.NET Core Blazor.

`@ref`

This scenario only applies to Razor components (`.razor`).

Component references (`@ref`) provide a way to reference a component instance so that you can issue commands to that instance. For more information, see ASP.NET Core Razor components.

Templated Razor delegates

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

Razor templates allow you to define a UI snippet with the following format:

```cshtml
@<tag>...</tag>
```

The following example illustrates how to specify a templated Razor delegate as a Func<T,TResult>. The dynamic type is specified for the parameter of the method that the delegate encapsulates. An object type is specified as the return value of the delegate. The template is used with a List<T> of `Pet` that has a `Name` property.

C#

```
public class Pet
{
    public string Name { get; set; }
}
```

CSHTML

```
@{
    Func<dynamic, object> petTemplate = @<p>You have a pet named
<strong>@item.Name</strong>.</p>;

    var pets = new List<Pet>
    {
        new Pet { Name = "Rin Tin Tin" },
        new Pet { Name = "Mr. Bigglesworth" },
        new Pet { Name = "K-9" }
    };
}
```

The template is rendered with `pets` supplied by a `foreach` statement:

CSHTML

```
@foreach (var pet in pets)
{
    @petTemplate(pet)
}
```

Rendered output:

HTML

```
<p>You have a pet named <strong>Rin Tin Tin</strong>.</p>
<p>You have a pet named <strong>Mr. Bigglesworth</strong>.</p>
<p>You have a pet named <strong>K-9</strong>.</p>
```

You can also supply an inline Razor template as an argument to a method. In the following example, the `Repeat` method receives a Razor template. The method uses the template to produce HTML content with repeats of items supplied from a list:

CSHTML

```
@using Microsoft.AspNetCore.Html

@functions {
    public static IHtmlContent Repeat(IEnumerable<dynamic> items, int times,
        Func<dynamic, IHtmlContent> template)
    {
        var html = new HtmlContentBuilder();

        foreach (var item in items)
        {
            for (var i = 0; i < times; i++)
            {
                html.AppendHtml(template(item));
            }
        }

        return html;
    }
}
```

Using the list of pets from the prior example, the `Repeat` method is called with:

- List<T> of `Pet`.
- Number of times to repeat each pet.
- Inline template to use for the list items of an unordered list.

CSHTML

```
<ul>
    @Repeat(pets, 3, @<li>@item.Name</li>)
</ul>
```

Rendered output:

HTML

```
<ul>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>K-9</li>
    <li>K-9</li>
    <li>K-9</li>
</ul>
```

Tag Helpers

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

There are three directives that pertain to Tag Helpers.

Expand table

Directive	Function
@addTagHelper	Makes Tag Helpers available to a view.
@removeTagHelper	Removes Tag Helpers previously added from a view.
@tagHelperPrefix	Specifies a tag prefix to enable Tag Helper support and to make Tag Helper usage explicit.

Razor reserved keywords

Razor keywords

- `page`
- `namespace`
- `functions`
- `inherits`
- `model`
- `section`
- `helper` (Not currently supported by ASP.NET Core)

Razor keywords are escaped with `@(Razor Keyword)` (for example, `@(functions)`).

C# Razor keywords

- `case`
- `do`
- `default`
- `for`
- `foreach`
- `if`
- `else`
- `lock`

- `switch`
- `try`
- `catch`
- `finally`
- `using`
- `while`

C# Razor keywords must be double-escaped with `@(@C# Razor Keyword)` (for example, `@(@case)`). The first `@` escapes the Razor parser. The second `@` escapes the C# parser.

Reserved keywords not used by Razor

- `class`

Inspect the Razor C# class generated for a view

The Razor SDK handles compilation of Razor files. By default, the generated code files aren't emitted. To enable emitting the code files, set the `EmitCompilerGeneratedFiles` directive in the project file (`.csproj`) to `true`:

XML

```xml
<PropertyGroup>
  <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
</PropertyGroup>
```

When building a 6.0 project (`net6.0`) in the `Debug` build configuration, the Razor SDK generates an `obj/Debug/net6.0/generated/` directory in the project root. Its subdirectory contains the emitted Razor page code files.

View lookups and case sensitivity

The Razor view engine performs case-sensitive lookups for views. However, the actual lookup is determined by the underlying file system:

- File based source:
  - On operating systems with case insensitive file systems (for example, Windows), physical file provider lookups are case insensitive. For example, `return View("Test")` results in matches for `/Views/Home/Test.cshtml`, `/Views/home/test.cshtml`, and any other casing variant.

    - On case-sensitive file systems (for example, Linux, OSX, and with `EmbeddedFileProvider`), lookups are case-sensitive. For example, `return View("Test")` specifically matches `/Views/Home/Test.cshtml`.
- Precompiled views: With ASP.NET Core 2.0 and later, looking up precompiled views is case insensitive on all operating systems. The behavior is identical to physical file provider's behavior on Windows. If two precompiled views differ only in case, the result of lookup is non-deterministic.

Developers are encouraged to match the casing of file and directory names to the casing of:

- Area, controller, and action names.
- Razor Pages.

Matching case ensures the deployments find their views regardless of the underlying file system.

Imports used by Razor

The following imports are generated by the ASP.NET Core web templates to support Razor Files:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
```

Additional resources

Introduction to ASP.NET Web Programming Using the Razor Syntax provides many samples of programming with Razor syntax.

Create reusable UI using the Razor class library project in ASP.NET Core

Article • 07/23/2024

By Rick Anderson

Razor views, pages, controllers, page models, Razor components, View components, and data models can be built into a Razor class library (RCL). The RCL can be packaged and reused. Applications can include the RCL and override the views and pages it contains. When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (`.cshtml` file) in the web app takes precedence.

For information on how to integrate npm and webpack into the build process for a RCL, see Build client web assets for your Razor Class Library.

Create a class library containing Razor UI

Visual Studio

- From Visual Studio select **Create new a new project**.
- Select **Razor Class Library** > **Next**.
- Name the library (for example, "RazorClassLib"), > **Create**. To avoid a file name collision with the generated view library, ensure the library name doesn't end in `.Views`.
- Select **Support pages and views** if you need the library to contain pages and/or views. By default, only Razor components are supported. Select **Create**.

The Razor Class Library template defaults to Razor component development by default. The **Support pages and views** option supports pages and views. For more information on RCL support in Blazor, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Add Razor files to the RCL.

The ASP.NET Core templates assume the RCL content is in the `Areas` folder. See RCL Pages layout below to create an RCL that exposes content in `~/Pages` rather than `~/Areas/Pages`.

Reference RCL content

The RCL can be referenced by:

- NuGet package. See Creating NuGet packages and dotnet add package and Create and publish a NuGet package.
- `{ProjectName}.csproj`. See dotnet-add reference.

Override views, partial views, and pages

When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (`.cshtml` file) in the web app takes precedence. For example, add `WebApp1/Areas/MyFeature/Pages/Page1.cshtml` to WebApp1, and Page1 in the WebApp1 will take precedence over Page1 in the RCL.

In the sample download, rename `WebApp1/Areas/MyFeature2` to `WebApp1/Areas/MyFeature` to test precedence.

Copy the `RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml` partial view to `WebApp1/Areas/MyFeature/Pages/Shared/_Message.cshtml`. Update the markup to indicate the new location. Build and run the app to verify the app's version of the partial is being used.

If the RCL uses Razor Pages, enable the Razor Pages services and endpoints in the hosting app:

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

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

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

RCL Pages layout

To reference RCL content as though it is part of the web app's `Pages` folder, create the RCL project with the following file structure:

- `RazorUIClassLib/Pages`
- `RazorUIClassLib/Pages/Shared`

Suppose `RazorUIClassLib/Pages/Shared` contains two partial files: `_Header.cshtml` and `_Footer.cshtml`. The `<partial>` tags could be added to `_Layout.cshtml` file:

CSHTML

```
<body>
  <partial name="_Header">
  @RenderBody()
  <partial name="_Footer">
</body>
```

Add the `_ViewStart.cshtml` file to the RCL project's `Pages` folder to use the `_Layout.cshtml` file from the host web app:

CSHTML

```
@{
    Layout = "_Layout";
}
```

Create an RCL with static assets

An RCL may require companion static assets that can be referenced by either the RCL or the consuming app of the RCL. ASP.NET Core allows creating RCLs that include static assets that are available to a consuming app.

To include companion assets as part of an RCL, create a `wwwroot` folder in the class library and include any required files in that folder.

When packing an RCL, all companion assets in the `wwwroot` folder are automatically included in the package.

Use the `dotnet pack` command rather than the NuGet.exe version `nuget pack`.

Add client web assets to the build process

Integrating client web assets into the build pipeline is nontrivial. See Build client web assets for your Razor Class Library for more information.

Exclude static assets

To exclude static assets, add the desired exclusion path to the `$(DefaultItemExcludes)` property group in the project file. Separate entries with a semicolon (`;`).

In the following example, the `lib.css` stylesheet in the `wwwroot` folder isn't considered a static asset and isn't included in the published RCL:

```xml
<PropertyGroup>

<DefaultItemExcludes>$(DefaultItemExcludes);wwwroot\lib.css</DefaultItemExcl
udes>
</PropertyGroup>
```

Typescript integration

To include TypeScript files in an RCL:

1. Reference the Microsoft.TypeScript.MSBuild NuGet package in the project.

   Note

   For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

2. Place the TypeScript files (`.ts`) outside of the `wwwroot` folder. For example, place the files in a `Client` folder.

3. Configure the TypeScript build output for the `wwwroot` folder. Set the `TypescriptOutDir` property inside of a `PropertyGroup` in the project file:

   XML

   ```xml
   <TypescriptOutDir>wwwroot</TypescriptOutDir>
   ```

4. Include the TypeScript target as a dependency of the `PrepareForBuildDependsOn` target by adding the following target inside of a `PropertyGroup` in the project file:

   XML

   ```xml
   <PrepareForBuildDependsOn>
     CompileTypeScript;
     GetTypeScriptOutputForPublishing;$(PrepareForBuildDependsOn)
   </PrepareForBuildDependsOn>
   ```

Consume content from a referenced RCL

The files included in the `wwwroot` folder of the RCL are exposed to either the RCL or the consuming app under the prefix `_content/{PACKAGE ID}/`. For example, a library with an assembly name of `Razor.Class.Lib` and without a `<PackageId>` specified in its project file results in a path to static content at `_content/Razor.Class.Lib/`. When producing a NuGet package and the assembly name isn't the same as the package ID (<PackageId> in the library's project file), use the package ID as specified in the project file for `{PACKAGE ID}`.

The consuming app references static assets provided by the library with `<script>`, `<style>`, `<img>`, and other HTML tags. The consuming app must have static file support enabled in:

C#

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

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
```

```
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

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

When running the consuming app from build output (`dotnet run`), static web assets are enabled by default in the Development environment. To support assets in other environments when running from build output, call UseStaticWebAssets on the host builder in `Program.cs`:

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

builder.WebHost.UseWebRoot("wwwroot");
builder.WebHost.UseStaticWebAssets();

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Calling `UseStaticWebAssets` isn't required when running an app from published output (`dotnet publish`).

Multi-project development flow

When the consuming app runs:

- The assets in the RCL stay in their original folders. The assets aren't moved to the consuming app.
- Any change within the RCL's `wwwroot` folder is reflected in the consuming app after the RCL is rebuilt and without rebuilding the consuming app.

When the RCL is built, a manifest is produced that describes the static web asset locations. The consuming app reads the manifest at runtime to consume the assets from referenced projects and packages. When a new asset is added to an RCL, the RCL must be rebuilt to update its manifest before a consuming app can access the new asset.

Publish

When the app is published, the companion assets from all referenced projects and packages are copied into the `wwwroot` folder of the published app under `_content/{PACKAGE ID}/`. When producing a NuGet package and the assembly name isn't the same as the package ID (<PackageId> in the library's project file), use the package ID as specified in the project file for `{PACKAGE ID}` when examining the `wwwroot` folder for the published assets.

Additional resources

- View or download sample code (how to download)
- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- ASP.NET Core Blazor CSS isolation

ASP.NET Core built-in Tag Helpers

Article • 06/03/2022

By Peter Kellner

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

There are built-in Tag Helpers which aren't listed in this document. The unlisted Tag Helpers are used internally by the Razor view engine. The Tag Helper for the `~` (tilde) character is unlisted. The tilde Tag Helper expands to the root path of the website.

Built-in ASP.NET Core Tag Helpers

Anchor

Cache

Component

Distributed Cache

Environment

Form

Form Action

Image

Input

Label

Link

Partial

Persist Component State

Script

Select

Textarea

Validation Message

Validation Summary

Additional resources

- Tag Helpers in ASP.NET Core
- Tag Helper Components in ASP.NET Core

Tag Helpers in ASP.NET Core

Article • 03/05/2024

By Rick Anderson

What are Tag Helpers

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. For example, the built-in `ImageTagHelper` can append a version number to the image name. Whenever the image changes, the server generates a new unique version for the image, so clients are guaranteed to get the current image (instead of a stale cached image). There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LabelTagHelper` can target the HTML `<label>` element when the `LabelTagHelper` attributes are applied. If you're familiar with HTML Helpers, Tag Helpers reduce the explicit transitions between HTML and C# in Razor views. In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. Tag Helpers compared to HTML Helpers explains the differences in more detail.

Tag Helpers aren't supported in Razor components. For more information, see ASP.NET Core Razor components.

What Tag Helpers provide

An HTML-friendly development experience

For the most part, Razor markup using Tag Helpers looks like standard HTML. Front-end designers conversant with HTML/CSS/JavaScript can edit Razor without learning C# Razor syntax.

A rich IntelliSense environment for creating HTML and Razor markup

This is in sharp contrast to HTML Helpers, the previous approach to server-side creation of markup in Razor views. Tag Helpers compared to HTML Helpers explains the differences in more detail. IntelliSense support for Tag Helpers explains the IntelliSense environment. Even developers experienced with Razor C# syntax are more productive using Tag Helpers than writing C# Razor markup.

A way to make you more productive and able to produce more robust, reliable, and maintainable code using information only available on the server

For example, historically the mantra on updating images was to change the name of the image when you change the image. Images should be aggressively cached for performance reasons, and unless you change the name of an image, you risk clients getting a stale copy. Historically, after an image was edited, the name had to be changed and each reference to the image in the web app needed to be updated. Not only is this very labor intensive, it's also error prone (you could miss a reference, accidentally enter the wrong string, etc.) The built-in `ImageTagHelper` can do this for you automatically. The `ImageTagHelper` can append a version number to the image name, so whenever the image changes, the server automatically generates a new unique version for the image. Clients are guaranteed to get the current image. This robustness and labor savings comes essentially free by using the `ImageTagHelper`.

Most built-in Tag Helpers target standard HTML elements and provide server-side attributes for the element. For example, the `<input>` element used in many views in the *Views/Account* folder contains the `asp-for` attribute. This attribute extracts the name of the specified model property into the rendered HTML. Consider a Razor view with the following model:

C#

```csharp
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

The following Razor markup:

CSHTML

```cshtml
<label asp-for="Movie.Title"></label>
```

Generates the following HTML:

HTML

```
<label for="Movie_Title">Title</label>
```

The `asp-for` attribute is made available by the `For` property in the LabelTagHelper. See Author Tag Helpers for more information.

Managing Tag Helper scope

Tag Helpers scope is controlled by a combination of `@addTagHelper`, `@removeTagHelper`, and the "!" opt-out character.

`@addTagHelper` makes Tag Helpers available

If you create a new ASP.NET Core web app named *AuthoringTagHelpers*, the following `Views/_ViewImports.cshtml` file will be added to your project:

CSHTML

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

The `@addTagHelper` directive makes Tag Helpers available to the view. In this case, the view file is `Pages/_ViewImports.cshtml`, which by default is inherited by all files in the *Pages* folder and subfolders; making Tag Helpers available. The code above uses the wildcard syntax ("*") to specify that all Tag Helpers in the specified assembly (*Microsoft.AspNetCore.Mvc.TagHelpers*) will be available to every view file in the *Views* directory or subdirectory. The first parameter after `@addTagHelper` specifies the Tag Helpers to load (we are using "*" for all Tag Helpers), and the second parameter "Microsoft.AspNetCore.Mvc.TagHelpers" specifies the assembly containing the Tag Helpers. *Microsoft.AspNetCore.Mvc.TagHelpers* is the assembly for the built-in ASP.NET Core Tag Helpers.

To expose all of the Tag Helpers in this project (which creates an assembly named *AuthoringTagHelpers*), you would use the following:

CSHTML

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

If your project contains an `EmailTagHelper` with the default namespace (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), you can provide the fully qualified name (FQN) of the Tag Helper:

CSHTML

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper,
AuthoringTagHelpers
```

To add a Tag Helper to a view using an FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (*AuthoringTagHelpers*). Most developers prefer to use the "*" wildcard syntax. The wildcard syntax allows you to insert the wildcard character "*" as the suffix in an FQN. For example, any of the following directives will bring in the `EmailTagHelper`:

CSHTML

```
@addTagHelper AuthoringTagHelpers.TagHelpers.E*, AuthoringTagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.Email*, AuthoringTagHelpers
```

As mentioned previously, adding the `@addTagHelper` directive to the `Views/_ViewImports.cshtml` file makes the Tag Helper available to all view files in the *Views* directory and subdirectories. You can use the `@addTagHelper` directive in specific view files if you want to opt-in to exposing the Tag Helper to only those views.

`@removeTagHelper` removes Tag Helpers

The `@removeTagHelper` has the same two parameters as `@addTagHelper`, and it removes a Tag Helper that was previously added. For example, `@removeTagHelper` applied to a specific view removes the specified Tag Helper from the view. Using `@removeTagHelper` in a `Views/Folder/_ViewImports.cshtml` file removes the specified Tag Helper from all of the views in *Folder*.

Controlling Tag Helper scope with the `_ViewImports.cshtml` file

You can add a `_ViewImports.cshtml` to any view folder, and the view engine applies the directives from both that file and the `Views/_ViewImports.cshtml` file. If you added an empty `Views/Home/_ViewImports.cshtml` file for the *Home* views, there would be no change because the `_ViewImports.cshtml` file is additive. Any `@addTagHelper` directives you add to the `Views/Home/_ViewImports.cshtml` file (that are not in the default `Views/_ViewImports.cshtml` file) would expose those Tag Helpers to views only in the *Home* folder.

Opting out of individual elements

You can disable a Tag Helper at the element level with the Tag Helper opt-out character ("!"). For example, `Email` validation is disabled in the `<span>` with the Tag Helper opt-out character:

CSHTML

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

You must apply the Tag Helper opt-out character to the opening and closing tag. (The Visual Studio editor automatically adds the opt-out character to the closing tag when you add one to the opening tag). After you add the opt-out character, the element and Tag Helper attributes are no longer displayed in a distinctive font.

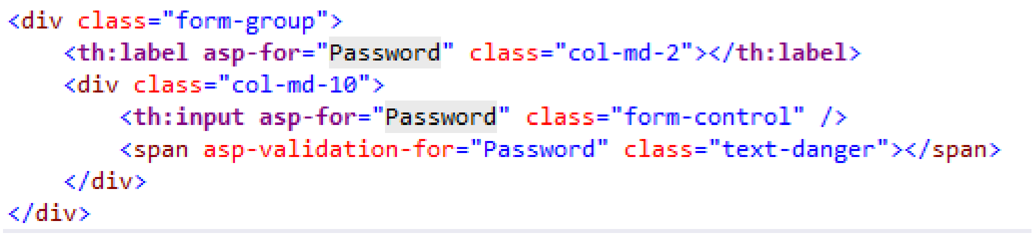
Using `@tagHelperPrefix` to make Tag Helper usage explicit

The `@tagHelperPrefix` directive allows you to specify a tag prefix string to enable Tag Helper support and to make Tag Helper usage explicit. For example, you could add the following markup to the `Views/_ViewImports.cshtml` file:

CSHTML

```
@tagHelperPrefix th:
```

In the code image below, the Tag Helper prefix is set to `th:`, so only those elements using the prefix `th:` support Tag Helpers (Tag Helper-enabled elements have a distinctive font). The `<label>` and `<input>` elements have the Tag Helper prefix and are Tag Helper-enabled, while the `<span>` element doesn't.



```
<div class="form-group">
    <th:label asp-for="Password" class="col-md-2"></th:label>
    <div class="col-md-10">
        <th:input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>
</div>
```

The same hierarchy rules that apply to `@addTagHelper` also apply to `@tagHelperPrefix`.

Self-closing Tag Helpers

Many Tag Helpers can't be used as self-closing tags. Some Tag Helpers are designed to be self-closing tags. Using a Tag Helper that was not designed to be self-closing suppresses the rendered output. Self-closing a Tag Helper results in a self-closing tag in the rendered output. For more information, see this note in Authoring Tag Helpers.

C# in Tag Helpers attribute/declaration

Tag Helpers do not allow C# in the element's attribute or tag declaration area. For example, the following code is not valid:

CSHTML

```
<input asp-for="LastName"
       @(Model?.LicenseId == null ? "disabled" : string.Empty) />
```

The preceding code can be written as:

CSHTML

```
<input asp-for="LastName"
       disabled="@(Model?.LicenseId == null)" />
```

Normally, the `@` operator inserts a textual representation of an expression into the rendered HTML markup. However, when an expression evaluates to logical `false`, the framework removes the attribute instead. In the preceding example, the `disabled` attribute is set to `true` if either `Model` or `LicenseId` is `null`.

Tag helper initializers

While attributes can be used to configure individual instances of tag helpers, ITagHelperInitializer<TTagHelper> can be used to configure all tag helper instances of a specific kind. Consider the following example of a tag helper initializer that configures the `asp-append-version` attribute or `AppendVersion` property for all instances of `ScriptTagHelper` in the app:

C#

```
public class AppendVersionTagHelperInitializer :
ITagHelperInitializer<ScriptTagHelper>
{
    public void Initialize(ScriptTagHelper helper, ViewContext context)
    {
        helper.AppendVersion = true;
    }
}
```

To use the initializer, configure it by registering it as part of the application's startup:

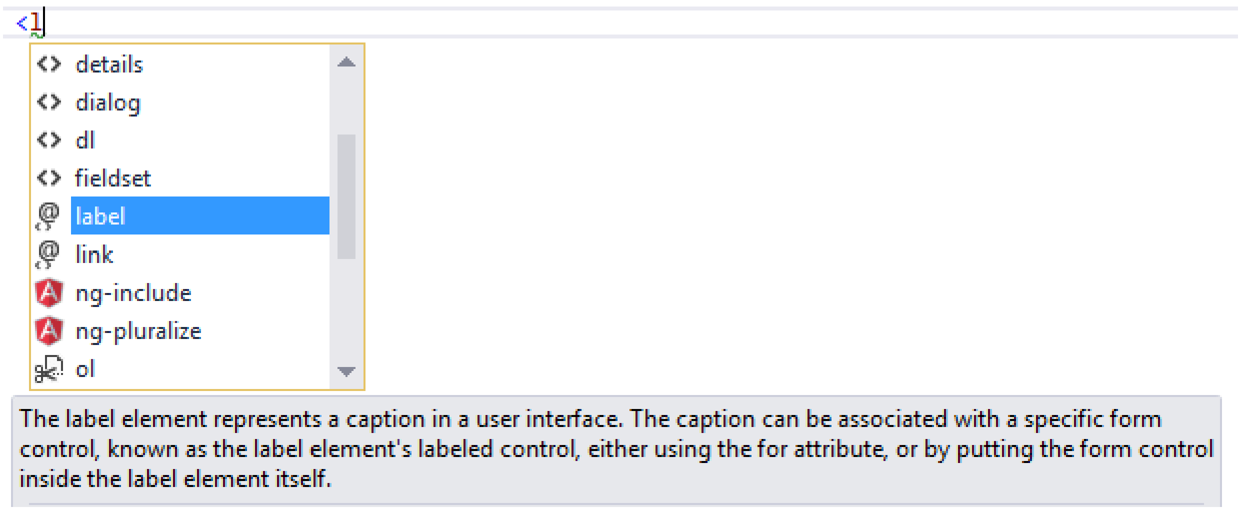
```C#
builder.Services.AddSingleton
    <ITagHelperInitializer<ScriptTagHelper>,
AppendVersionTagHelperInitializer>();
```

Tag Helper automatic version generation outside of wwwroot

For a Tag Helper to generate a version for a static file outside `wwwroot`, see Serve files from multiple locations

IntelliSense support for Tag Helpers

Consider writing an HTML `<label>` element. As soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:

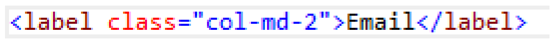


The label element represents a caption in a user interface. The caption can be associated with a specific form control, known as the label element's labeled control, either using the for attribute, or by putting the form control inside the label element itself.

Not only do you get HTML help, but also the icon (the "@" symbol with "<>" under it).

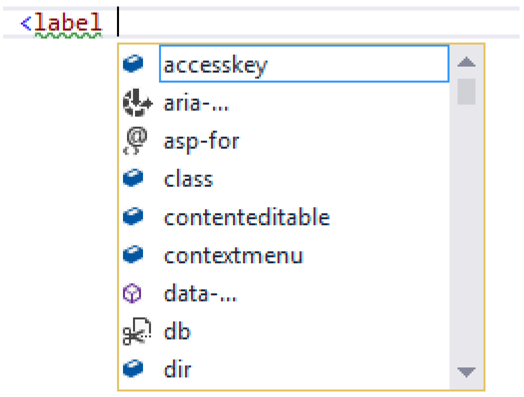
The icon identifies the element as targeted by Tag Helpers. Pure HTML elements (such as the `fieldset`) display the "<>" icon.

A pure HTML `<label>` tag displays the HTML tag (with the default Visual Studio color theme) in a brown font, the attributes in red, and the attribute values in blue.



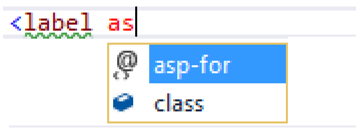
```
<label class="col-md-2">Email</label>
```

After you enter `<label`, IntelliSense lists the available HTML/CSS attributes and the Tag Helper-targeted attributes:



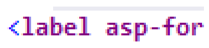
```
<label |
```

IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:



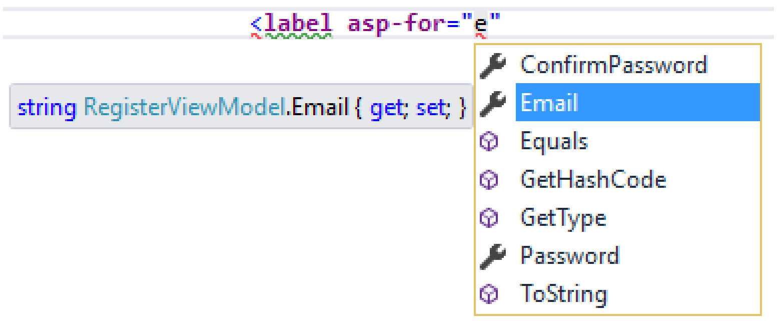
```
<label as|
```

As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio "Blue" or "Light" color theme, the font is bold purple. If you're using the "Dark" theme the font is bold teal. The images in this document were taken using the default theme.



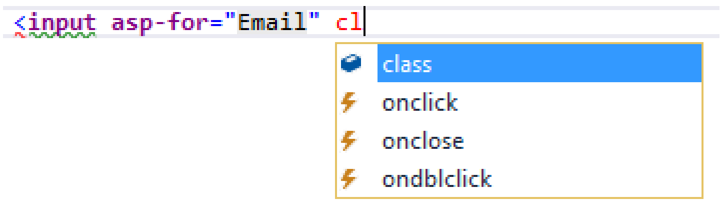
```
<label asp-for
```

You can enter the Visual Studio *CompleteWord* shortcut (Ctrl +spacebar is the default) inside the double quotes (""), and you are now in C#, just like you would be in a C# class. IntelliSense displays all the methods and properties on the page model. The methods and properties are available because the property type is `ModelExpression`. In the image below, I'm editing the `Register` view, so the `RegisterViewModel` is available.

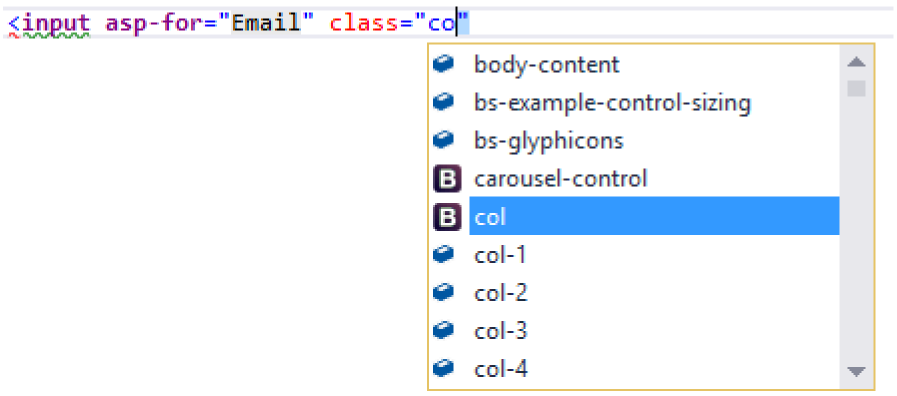


```
<label asp-for="e"
```

IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:



```
<input asp-for="Email" cl
```



```
<input asp-for="Email" class="co"
```

Tag Helpers compared to HTML Helpers

Tag Helpers attach to HTML elements in Razor views, while HTML Helpers are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class "caption":

CSHTML

```
@Html.Label("FirstName", "First Name:", new {@class="caption"})
```

The at (`@`) symbol tells Razor this is the start of code. The next two parameters ("FirstName" and "First Name:") are strings, so IntelliSense can't help. The last argument:

CSHTML

```
new {@class="caption"}
```

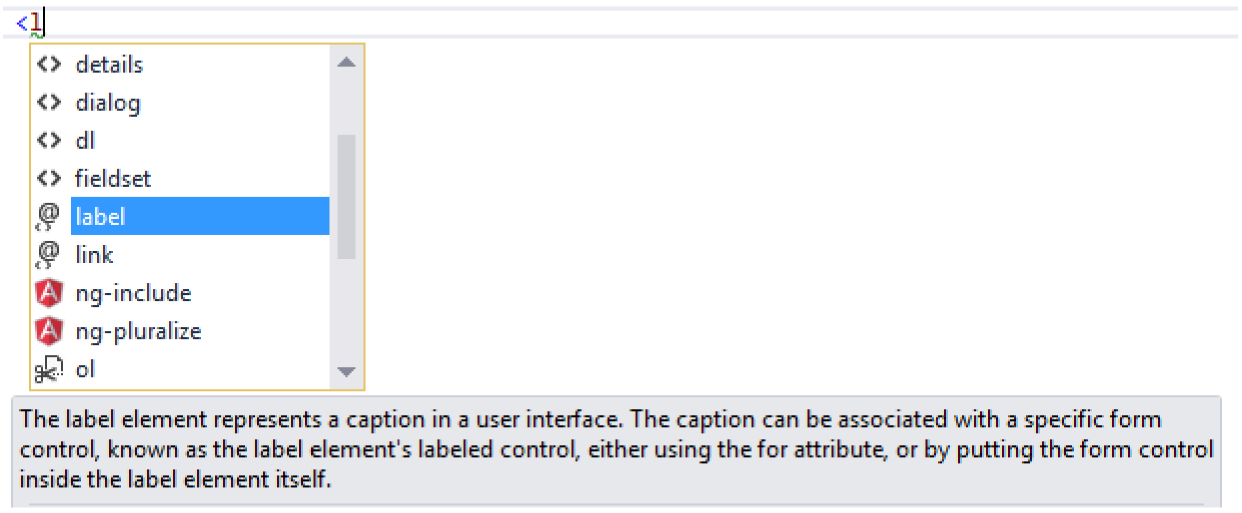
Is an anonymous object used to represent attributes. Because `class` is a reserved keyword in C#, you use the `@` symbol to force C# to interpret `@class=` as a symbol (property name). To a front-end designer (someone familiar with HTML/CSS/JavaScript and other client technologies but not familiar with C# and Razor), most of the line is foreign. The entire line must be authored with no help from IntelliSense.

Using the `LabelTagHelper`, the same markup can be written as:

CSHTML

```
<label class="caption" asp-for="FirstName"></label>
```

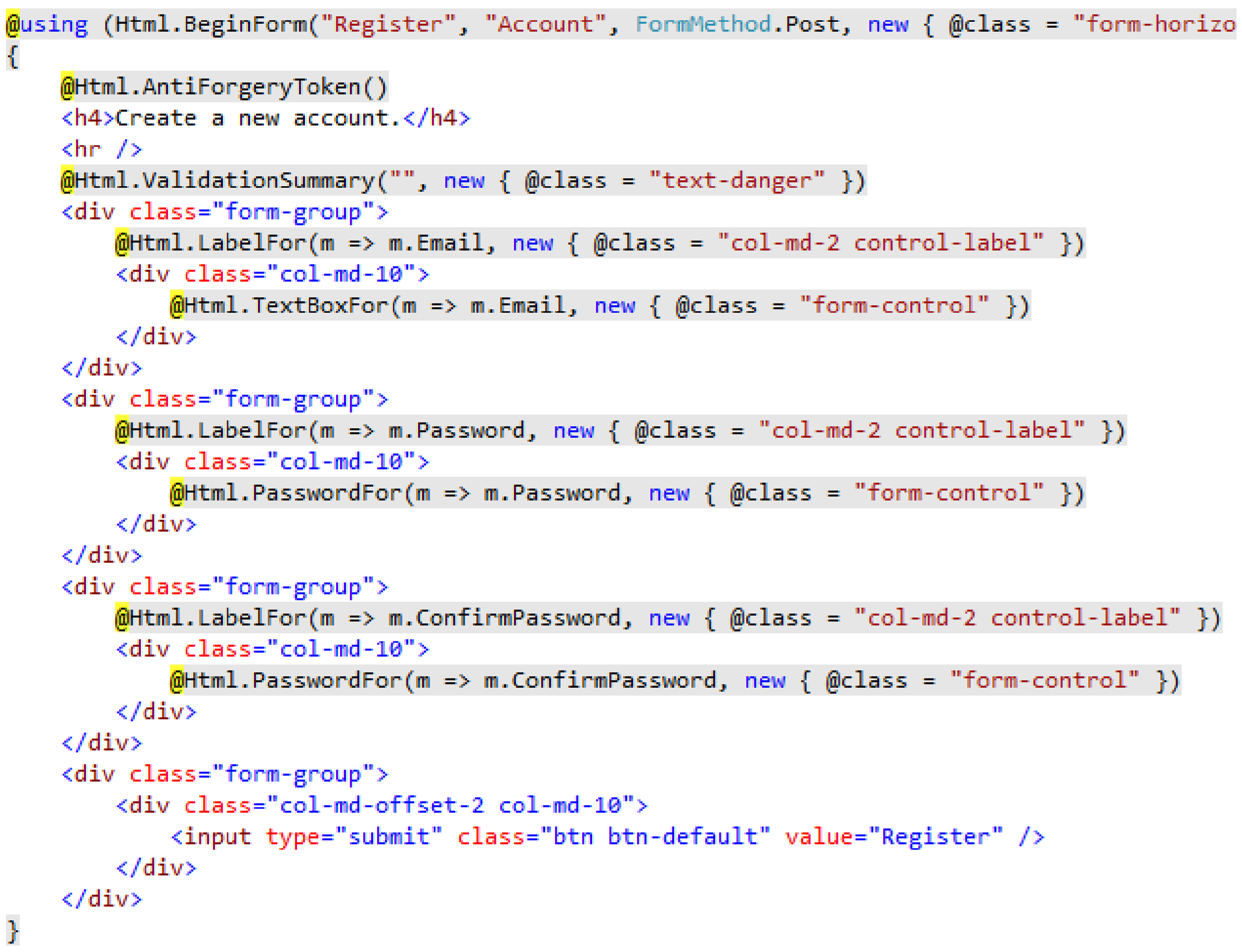
With the Tag Helper version, as soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



The label element represents a caption in a user interface. The caption can be associated with a specific form control, known as the label element's labeled control, either using the for attribute, or by putting the form control inside the label element itself.

IntelliSense helps you write the entire line.

The following code image shows the Form portion of the `Views/Account/Register.cshtml` Razor view generated from the ASP.NET 4.5.x MVC template included with Visual Studio.



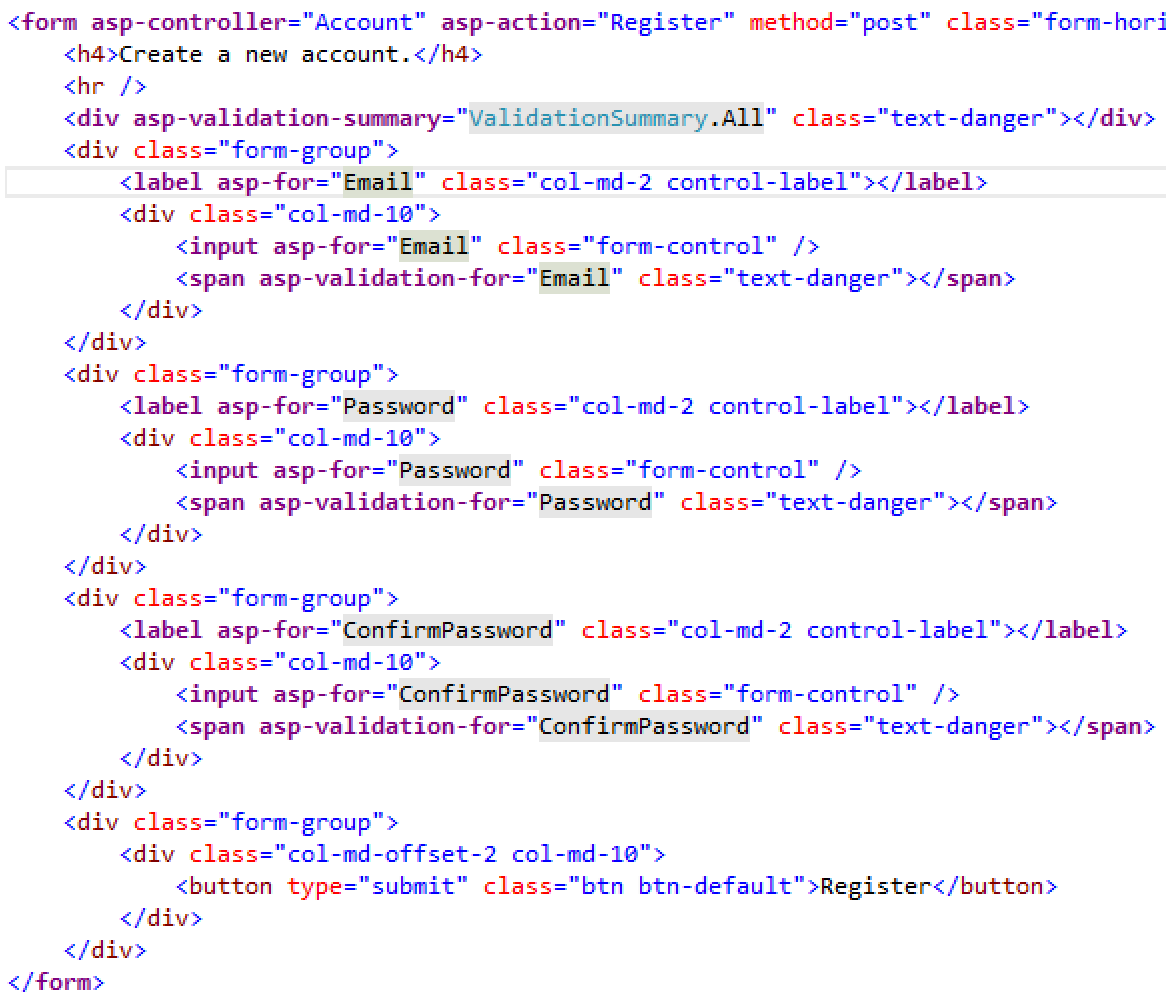
```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

The Visual Studio editor displays C# code with a grey background. For example, the `AntiForgeryToken` HTML Helper:

CSHTML

```
@Html.AntiForgeryToken()
```

is displayed with a grey background. Most of the markup in the Register view is C#. Compare that to the equivalent approach using Tag Helpers:



```
<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>
```

The markup is much cleaner and easier to read, edit, and maintain than the HTML Helpers approach. The C# code is reduced to the minimum that the server needs to know about. The Visual Studio editor displays markup targeted by a Tag Helper in a distinctive font.

Consider the *Email* group:

CSHTML

```
<div class="form-group">
    <label asp-for="Email" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
</div>
```

Each of the "asp-" attributes has a value of "Email", but "Email" isn't a string. In this context, "Email" is the C# model expression property for the `RegisterViewModel`.

The Visual Studio editor helps you write **all** of the markup in the Tag Helper approach of the register form, while Visual Studio provides no help for most of the code in the HTML

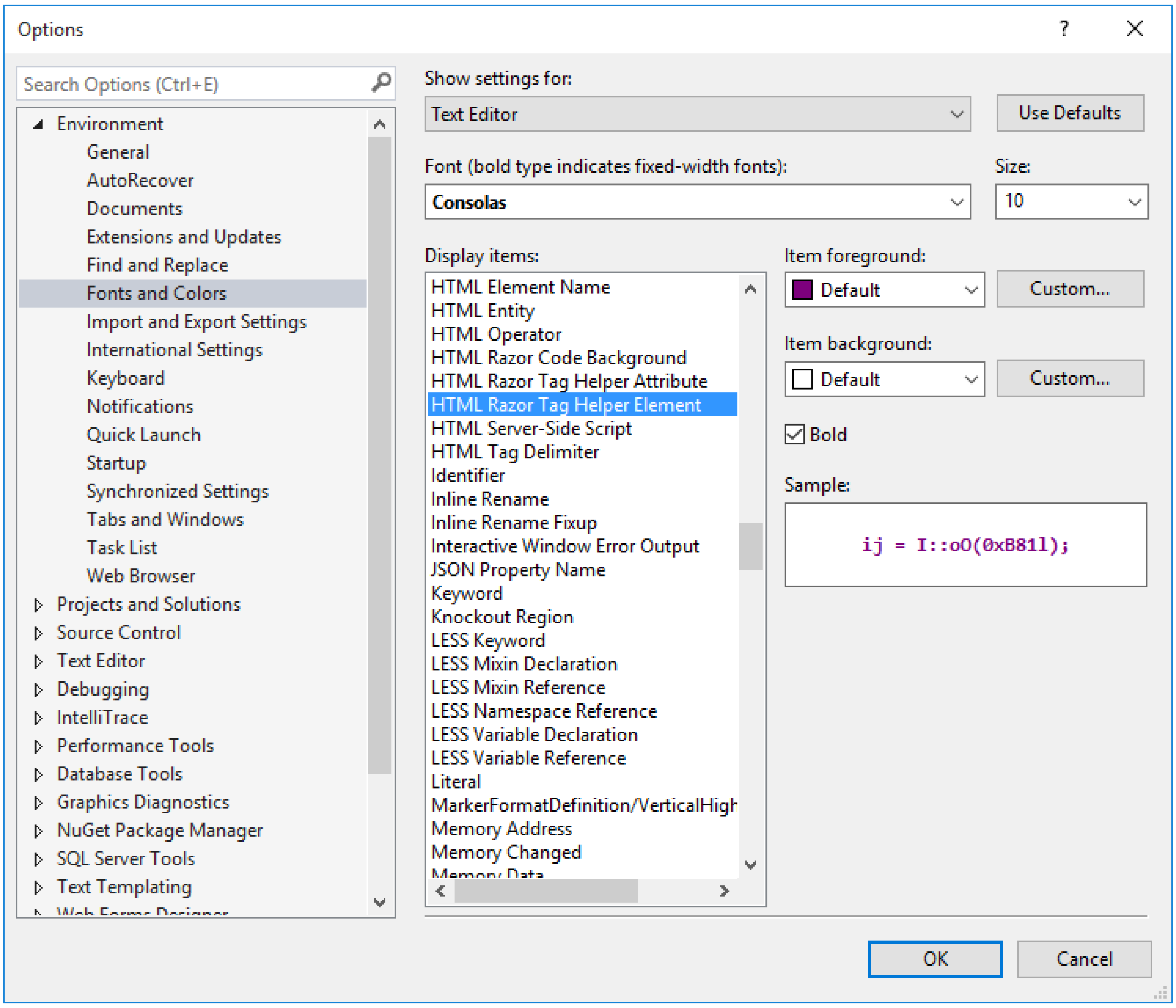
Helpers approach. IntelliSense support for Tag Helpers goes into detail on working with Tag Helpers in the Visual Studio editor.

Tag Helpers compared to Web Server Controls

- Tag Helpers don't own the element they're associated with; they simply participate in the rendering of the element and content. ASP.NET Web Server Controls are declared and invoked on a page.
- ASP.NET Web Server Controls have a non-trivial lifecycle that can make developing and debugging difficult.
- Web Server controls allow you to add functionality to the client DOM elements by using a client control. Tag Helpers have no DOM.
- Web Server controls include automatic browser detection. Tag Helpers have no knowledge of the browser.
- Multiple Tag Helpers can act on the same element (see Avoiding Tag Helper conflicts) while you typically can't compose Web Server controls.
- Tag Helpers can modify the tag and content of HTML elements that they're scoped to, but don't directly modify anything else on a page. Web Server controls have a less specific scope and can perform actions that affect other parts of your page; enabling unintended side effects.
- Web Server controls use type converters to convert strings into objects. With Tag Helpers, you work natively in C#, so you don't need to do type conversion.
- Web Server controls use System.ComponentModel to implement the run-time and design-time behavior of components and controls. `System.ComponentModel` includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components. Contrast that to Tag Helpers, which typically derive from `TagHelper`, and the `TagHelper` base class exposes only two methods, `Process` and `ProcessAsync`.

Customizing the Tag Helper element font

You can customize the font and colorization from **Tools** > **Options** > **Environment** > **Fonts and Colors**:



Show settings for: Text Editor

Font (bold type indicates fixed-width fonts): **Consolas**

Size: 10

Item foreground: Default

Item background: Default

☑ Bold

```
ij = I::oO(0xB81l);
```

Built-in ASP.NET Core Tag Helpers

Anchor

Cache

Component

Distributed Cache

Environment

Form

Form Action

Image

Input

Label

Link

Partial

Persist Component State

Script

Select

Textarea

Validation Message

Validation Summary

Additional resources

- Author Tag Helpers
- Working with Forms
- TagHelperSamples on GitHub contains Tag Helper samples for working with Bootstrap.

Author Tag Helpers in ASP.NET Core

Article • 09/14/2022

By Rick Anderson

View or download sample code (how to download)

Get started with Tag Helpers

This tutorial provides an introduction to programming Tag Helpers. Introduction to Tag Helpers describes the benefits that Tag Helpers provide.

A tag helper is any class that implements the `ITagHelper` interface. However, when you author a tag helper, you generally derive from `TagHelper`, doing so gives you access to the `Process` method.

1. Create a new ASP.NET Core project called **AuthoringTagHelpers**. You won't need authentication for this project.

2. Create a folder to hold the Tag Helpers called *TagHelpers*. The *TagHelpers* folder is *not* required, but it's a reasonable convention. Now let's get started writing some simple tag helpers.

A minimal Tag Helper

In this section, you write a tag helper that updates an email tag. For example:

```HTML
<email>Support</email>
```

The server will use our email tag helper to convert that markup into the following:

```HTML
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

That is, an anchor tag that makes this an email link. You might want to do this if you are writing a blog engine and need it to send email for marketing, support, and other contacts, all to the same domain.

1. Add the following `EmailTagHelper` class to the *TagHelpers* folder.

```csharp
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

namespace AuthoringTagHelpers.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            output.TagName = "a";    // Replaces <email> with <a> tag
        }
    }
}
```

- Tag helpers use a naming convention that targets elements of the root class name (minus the *TagHelper* portion of the class name). In this example, the root name of **EmailTagHelper** is *email*, so the `<email>` tag will be targeted. This naming convention should work for most tag helpers, later on I'll show how to override it.
- The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides methods and properties for writing Tag Helpers.
- The overridden `Process` method controls what the tag helper does when executed. The `TagHelper` class also provides an asynchronous version (`ProcessAsync`) with the same parameters.
- The context parameter to `Process` (and `ProcessAsync`) contains information associated with the execution of the current HTML tag.
- The output parameter to `Process` (and `ProcessAsync`) contains a stateful HTML element representative of the original source used to generate an HTML tag and content.
- Our class name has a suffix of **TagHelper**, which is *not* required, but it's considered a best practice convention. You could declare the class as:

```csharp
public class Email : TagHelper
```

2. To make the `EmailTagHelper` class available to all our Razor views, add the `addTagHelper` directive to the `Views/_ViewImports.cshtml` file:

   CSHTML

   ```
   @using AuthoringTagHelpers
   @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
   @addTagHelper *, AuthoringTagHelpers
   ```

   The code above uses the wildcard syntax to specify all the tag helpers in our assembly will be available. The first string after `@addTagHelper` specifies the tag helper to load (Use "*" for all tag helpers), and the second string "AuthoringTagHelpers" specifies the assembly the tag helper is in. Also, note that the second line brings in the ASP.NET Core MVC tag helpers using the wildcard syntax (those helpers are discussed in Introduction to Tag Helpers.) It's the `@addTagHelper` directive that makes the tag helper available to the Razor view. Alternatively, you can provide the fully qualified name (FQN) of a tag helper as shown below:

C#

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper,
AuthoringTagHelpers
```

To add a tag helper to a view using a FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the **assembly name** (*AuthoringTagHelpers*, not necessarily the `namespace`). Most developers will prefer to use the wildcard syntax. Introduction to Tag Helpers goes into detail on tag helper adding, removing, hierarchy, and wildcard syntax.

1. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

   CSHTML

   ```
   @{
       ViewData["Title"] = "Contact";
   }
   <h2>@ViewData["Title"].</h2>
   <h3>@ViewData["Message"]</h3>

   <address>
       One Microsoft Way<br />
       Redmond, WA 98052<br />
   ```

```
        <abbr title="Phone">P:</abbr>
        425.555.0100
    </address>

    <address>
        <strong>Support:</strong><email>Support</email><br />
        <strong>Marketing:</strong><email>Marketing</email>
    </address>
```

2. Run the app and use your favorite browser to view the HTML source so you can verify that the email tags are replaced with anchor markup (For example, `<a>Support</a>`). *Support* and *Marketing* are rendered as a links, but they don't have an `href` attribute to make them functional. We'll fix that in the next section.

SetAttribute and SetContent

In this section, we'll update the `EmailTagHelper` so that it will create a valid anchor tag for email. We'll update it to take information from a Razor view (in the form of a `mail-to` attribute) and use that in generating the anchor.

Update the `EmailTagHelper` class with the following:

```C#
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // PascalCase gets translated into kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput
output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + address);
        output.Content.SetContent(address);
    }
}
```

- Pascal-cased class and property names for tag helpers are translated into their kebab case. Therefore, to use the `MailTo` attribute, you'll use `<email mail-to="value"/>` equivalent.

- The last line sets the completed content for our minimally functional tag helper.
- The highlighted line shows the syntax for adding attributes:

C#

```csharp
public override void Process(TagHelperContext context, TagHelperOutput
output)
{
    output.TagName = "a";    // Replaces <email> with <a> tag

    var address = MailTo + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
```

That approach works for the attribute "href" as long as it doesn't currently exist in the attributes collection. You can also use the `output.Attributes.Add` method to add a tag helper attribute to the end of the collection of tag attributes.

1. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

   CSHTML

   ```cshtml
   @{
       ViewData["Title"] = "Contact Copy";
   }
   <h2>@ViewData["Title"].</h2>
   <h3>@ViewData["Message"]</h3>

   <address>
       One Microsoft Way Copy Version <br />
       Redmond, WA 98052-6399<br />
       <abbr title="Phone">P:</abbr>
       425.555.0100
   </address>

   <address>
       <strong>Support:</strong><email mail-to="Support"></email><br />
       <strong>Marketing:</strong><email mail-to="Marketing"></email>
   </address>
   ```

2. Run the app and verify that it generates the correct links.

Note

If you were to write the email tag self-closing (`<email mail-to="Rick" />`), the final output would also be self-closing. To enable the ability to write the tag with only a start tag (`<email mail-to="Rick">`) you must mark the class with the following:

```C#
[HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
public class EmailVoidTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    // Code removed for brevity
```

With a self-closing email tag helper, the output would be `<a href="mailto:Rick@contoso.com" />`. Self-closing anchor tags are not valid HTML, so you wouldn't want to create one, but you might want to create a tag helper that's self-closing. Tag helpers set the type of the `TagMode` property after reading a tag.

You can also map a different attribute name to a property using the [HtmlAttributeName] attribute.

To map an attribute named `recipient` to the `MailTo` property:

```C#
[HtmlAttributeName("recipient")]
public string? MailTo { get; set; }
```

Tag Helper for the `recipient` attribute:

```HTML
<email recipient="…"/>
```

ProcessAsync

In this section, we'll write an asynchronous email helper.

1. Replace the `EmailTagHelper` class with the following code:

```C#
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
```

```
    {
        output.TagName = "a";                                 //
Replaces <email> with <a> tag
        var content = await output.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + target);
        output.Content.SetContent(target);
    }
}
```

Notes:

- This version uses the asynchronous `ProcessAsync` method. The asynchronous `GetChildContentAsync` returns a `Task` containing the `TagHelperContent`.
- Use the `output` parameter to get contents of the HTML element.

2. Make the following change to the `Views/Home/Contact.cshtml` file so the tag helper can get the target email.

CSHTML

```cshtml
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

3. Run the app and verify that it generates valid email links.

RemoveAll, PreContent.SetHtmlContent and PostContent.SetHtmlContent

1. Add the following `BoldTagHelper` class to the *TagHelpers* folder.

C#

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = "bold")]
    public class BoldTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            output.Attributes.RemoveAll("bold");
            output.PreContent.SetHtmlContent("<strong>");
            output.PostContent.SetHtmlContent("</strong>");
        }
    }
}
```

- The `[HtmlTargetElement]` attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named "bold" will match, and the `Process` override method in the class will run. In our sample, the `Process` method removes the "bold" attribute and surrounds the containing markup with `<strong></strong>`.

- Because you don't want to replace the existing tag content, you must write the opening `<strong>` tag with the `PreContent.SetHtmlContent` method and the closing `</strong>` tag with the `PostContent.SetHtmlContent` method.

2. Modify the `About.cshtml` view to contain a `bold` attribute value. The completed code is shown below.

CSHTML

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>
```

3. Run the app. You can use your favorite browser to inspect the source and verify the markup.

The `[HtmlTargetElement]` attribute above only targets HTML markup that provides an attribute name of "bold". The `<bold>` element wasn't modified by the tag helper.

4. Comment out the `[HtmlTargetElement]` attribute line and it will default to targeting `<bold>` tags, that is, HTML markup of the form `<bold>`. Remember, the default naming convention will match the class name **Bold**TagHelper to `<bold>` tags.

5. Run the app and verify that the `<bold>` tag is processed by the tag helper.

Decorating a class with multiple `[HtmlTargetElement]` attributes results in a logical-OR of the targets. For example, using the code below, a bold tag or a bold attribute will match.

```C#
[HtmlTargetElement("bold")]
[HtmlTargetElement(Attributes = "bold")]
public class BoldTagHelper : TagHelper
{
    public override void Process(TagHelperContext context, TagHelperOutput 
output)
    {
        output.Attributes.RemoveAll("bold");
        output.PreContent.SetHtmlContent("<strong>");
        output.PostContent.SetHtmlContent("</strong>");
    }
}
```

When multiple attributes are added to the same statement, the runtime treats them as a logical-AND. For example, in the code below, an HTML element must be named "bold" with an attribute named "bold" (`<bold bold />`) to match.

```C#
[HtmlTargetElement("bold", Attributes = "bold")]
```

You can also use the `[HtmlTargetElement]` to change the name of the targeted element. For example if you wanted the `BoldTagHelper` to target `<MyBold>` tags, you would use the following attribute:

```C#
[HtmlTargetElement("MyBold")]
```

Pass a model to a Tag Helper

1. Add a *Models* folder.

2. Add the following `WebsiteContext` class to the *Models* folder:

   C#

```
using System;

namespace AuthoringTagHelpers.Models
{
    public class WebsiteContext
    {
        public Version Version { get; set; }
        public int CopyrightYear { get; set; }
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

3. Add the following `WebsiteInformationTagHelper` class to the *TagHelpers* folder.

   C#

```
using System;
using AuthoringTagHelpers.Models;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
        public WebsiteContext Info { get; set; }

      public override void Process(TagHelperContext context,
TagHelperOutput output)
      {
         output.TagName = "section";
         output.Content.SetHtmlContent(
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
<li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
<li><strong>Approved:</strong> {Info.Approved}</li>
<li><strong>Number of tags to show:</strong> {Info.TagsToShow}</li>
</ul>");
         output.TagMode = TagMode.StartTagAndEndTag;
      }
   }
}
```

   - As mentioned previously, tag helpers translates Pascal-cased C# class names and properties for tag helpers into kebab case. Therefore, to use the

`WebsiteInformationTagHelper` in Razor, you'll write `<website-information />`.

- You are not explicitly identifying the target element with the `[HtmlTargetElement]` attribute, so the default of `website-information` will be targeted. If you applied the following attribute (note it's not kebab case but matches the class name):

C#

```csharp
[HtmlTargetElement("WebsiteInformation")]
```

The kebab case tag `<website-information />` wouldn't match. If you want use the `[HtmlTargetElement]` attribute, you would use kebab case as shown below:

C#

```csharp
[HtmlTargetElement("Website-Information")]
```

- Elements that are self-closing have no content. For this example, the Razor markup will use a self-closing tag, but the tag helper will be creating a section element (which isn't self-closing and you are writing content inside the `section` element). Therefore, you need to set `TagMode` to `StartTagAndEndTag` to write output. Alternatively, you can comment out the line setting `TagMode` and write markup with a closing tag. (Example markup is provided later in this tutorial.)

- The `$` (dollar sign) in the following line uses an interpolated string:

CSHTML

```cshtml
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

4. Add the following markup to the `About.cshtml` view. The highlighted markup displays the web site information.

CSHTML

```cshtml
@using AuthoringTagHelpers.Models
@{
    ViewData["Title"] = "About";
    WebsiteContext webContext = new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 };
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="webContext" />
```

ⓘ **Note**

In the Razor markup shown below:

HTML

```html
<website-information info="webContext" />
```

Razor knows the `info` attribute is a class, not a string, and you want to write C# code. Any non-string tag helper attribute should be written without the `@` character.

5. Run the app, and navigate to the About view to see the web site information.

ⓘ **Note**

You can use the following markup with a closing tag and remove the line with `TagMode.StartTagAndEndTag` in the tag helper:

HTML

```html
<website-information info="webContext" >
</website-information>
```

Condition Tag Helper

The condition tag helper renders output when passed a true value.

1. Add the following `ConditionTagHelper` class to the *TagHelpers* folder.

C#

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = nameof(Condition))]
    public class ConditionTagHelper : TagHelper
    {
        public bool Condition { get; set; }

        public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            if (!Condition)
            {
                output.SuppressOutput();
            }
        }
    }
}
```

2. Replace the contents of the `Views/Home/Index.cshtml` file with the following markup:

CSHTML

```
@using AuthoringTagHelpers.Models
@model WebsiteContext

@{
    ViewData["Title"] = "Home Page";
}

<div>
    <h3>Information about our website (outdated):</h3>
    <Website-InforMation info="Model" />
    <div condition="Model.Approved">
        <p>
            This website has <strong
surround="em">@Model.Approved</strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>
```

3. Replace the `Index` method in the `Home` controller with the following code:

C#

```csharp
public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}
```

4. Run the app and browse to the home page. The markup in the conditional `div` won't be rendered. Append the query string `?approved=true` to the URL (for example, `http://localhost:1235/Home/Index?approved=true`). `approved` is set to true and the conditional markup will be displayed.

ⓘ **Note**

Use the nameof operator to specify the attribute to target rather than specifying a string as you did with the bold tag helper:

C#

```csharp
[HtmlTargetElement(Attributes = nameof(Condition))]
 //   [HtmlTargetElement(Attributes = "condition")]
 public class ConditionTagHelper : TagHelper
{
   public bool Condition { get; set; }

   public override void Process(TagHelperContext context,
TagHelperOutput output)
   {
      if (!Condition)
      {
         output.SuppressOutput();
      }
   }
}
```

The nameof operator will protect the code should it ever be refactored (we might want to change the name to `RedCondition`).

Avoid Tag Helper conflicts

In this section, you write a pair of auto-linking tag helpers. The first will replace markup containing a URL starting with HTTP to an HTML anchor tag containing the same URL (and thus yielding a link to the URL). The second will do the same for a URL starting with WWW.

Because these two helpers are closely related and you may refactor them in the future, we'll keep them in the same file.

1. Add the following `AutoLinkerHttpTagHelper` class to the *TagHelpers* folder.

C#

```
[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor
tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent.GetContent(),
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http
link version}
    }
}
```

ⓘ **Note**

The `AutoLinkerHttpTagHelper` class targets `p` elements and uses Regex to create the anchor.

2. Add the following markup to the end of the `Views/Home/Contact.cshtml` file:

CSHTML

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

<p>Visit us at http://docs.asp.net or at www.microsoft.com</p>
```

3. Run the app and verify that the tag helper renders the anchor correctly.

4. Update the `AutoLinker` class to include the `AutoLinkerWwwTagHelper` which will convert www text to an anchor tag that also contains the original www text. The updated code is highlighted below:

```C#
    [HtmlTargetElement("p")]
    public class AutoLinkerHttpTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext
context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their
anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                 childContent.GetContent(),
                 @"\b(?:https?://)(\S+)\b",
                  "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http
link version}
        }
    }
    [HtmlTargetElement("p")]
    public class AutoLinkerWwwTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext
context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their
anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                childContent.GetContent(),
                 @"\b(www\.)(\S+)\b",
                 "<a target=\"_blank\" href=\"http://$0\">$0</a>"));
// www version
        }
    }
}
```

5. Run the app. Notice the www text is rendered as a link but the HTTP text isn't. If you put a break point in both classes, you can see that the HTTP tag helper class runs first. The problem is that the tag helper output is cached, and when the WWW tag helper is run, it overwrites the cached output from the HTTP tag helper. Later in the tutorial we'll see how to control the order that tag helpers run in. We'll fix the code with the following:

```C#
  public class AutoLinkerHttpTagHelper : TagHelper
  {
      public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
      {
          var childContent = output.Content.IsModified ?
output.Content.GetContent() :
              (await output.GetChildContentAsync()).GetContent();

          // Find Urls in the content and replace them with their
anchor tag equivalent.
          output.Content.SetHtmlContent(Regex.Replace(
               childContent,
              @"\b(?:https?://)(\S+)\b",
               "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http
link version}
      }
  }

  [HtmlTargetElement("p")]
  public class AutoLinkerWwwTagHelper : TagHelper
  {
      public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
      {
          var childContent = output.Content.IsModified ?
output.Content.GetContent() :
              (await output.GetChildContentAsync()).GetContent();

          // Find Urls in the content and replace them with their
anchor tag equivalent.
          output.Content.SetHtmlContent(Regex.Replace(
               childContent,
               @"\b(www\.)(\S+)\b",
               "<a target=\"_blank\" href=\"http://$0\">$0</a>"));  //
www version
      }
  }
```

ⓘ **Note**

In the first edition of the auto-linking tag helpers, you got the content of the target with the following code:

```C#
var childContent = await output.GetChildContentAsync();
```

That is, you call `GetChildContentAsync` using the `TagHelperOutput` passed into the `ProcessAsync` method. As mentioned previously, because the output is cached, the last tag helper to run wins. You fixed that problem with the following code:

```C#
var childContent = output.Content.IsModified ?
output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

The code above checks to see if the content has been modified, and if it has, it gets the content from the output buffer.

6. Run the app and verify that the two links work as expected. While it might appear our auto linker tag helper is correct and complete, it has a subtle problem. If the WWW tag helper runs first, the www links won't be correct. Update the code by adding the `Order` overload to control the order that the tag runs in. The `Order` property determines the execution order relative to other tag helpers targeting the same element. The default order value is zero and instances with lower values are executed first.

```
C#

public class AutoLinkerHttpTagHelper : TagHelper
{
    // This filter must run before the AutoLinkerWwwTagHelper as it
searches and replaces http and
    // the AutoLinkerWwwTagHelper adds http to the markup.
    public override int Order
    {
        get  {  return int.MinValue;   }
    }
```

The preceding code guarantees that the HTTP tag helper runs before the WWW tag helper. Change `Order` to `MaxValue` and verify that the markup generated for the WWW tag is incorrect.

Inspect and retrieve child content

The tag helpers provide several properties to retrieve content.

- The result of `GetChildContentAsync` can be appended to `output.Content`.
- You can inspect the result of `GetChildContentAsync` with `GetContent`.
- If you modify `output.Content`, the TagHelper body won't be executed or rendered unless you call `GetChildContentAsync` as in our auto-linker sample:

```
C#

public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ?
output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag
equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link
version}
    }
}
```

- Multiple calls to `GetChildContentAsync` returns the same value and doesn't re-execute the `TagHelper` body unless you pass in a false parameter indicating not to use the cached result.

Load minified partial view TagHelper

In production environments, performance can be improved by loading minified partial views. To take advantage of minified partial view in production:

- Create/set up a pre-build process that minifies partial views.
- Use the following code to load minified partial views in non-development environments.

```C#
public class MinifiedVersionPartialTagHelper : PartialTagHelper
    {
        public MinifiedVersionPartialTagHelper(ICompositeViewEngine 
viewEngine,
                                IViewBufferScope viewBufferScope)
                               : base(viewEngine, viewBufferScope)
        {

        }

        public override Task ProcessAsync(TagHelperContext context, 
TagHelperOutput output)
        {
            // Append ".min" to load the minified partial view.
            if (!IsDevelopment())
            {
                Name += ".min";
            }

            return base.ProcessAsync(context, output);
        }

        private bool IsDevelopment()
        {
            return 
Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")
                                                 == 
EnvironmentName.Development;
        }
    }
```

Tag Helpers in forms in ASP.NET Core

Article • 09/27/2024

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```CSHTML
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```html
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```cshtml
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```cshtml
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

ⓘ **Note**

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```cshtml
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```HTML
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```CSHTML
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```HTML
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```C#
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```cshtml
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```html
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```cshtml
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```csharp
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```cshtml
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```html
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```html
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```csharp
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```cshtml
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```cshtml
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```html
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```csharp
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```csharp
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```cshtml
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```html
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```csharp
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```csharp
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```cshtml
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```C#
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```CSHTML
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

```CSHTML
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```C#
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```cshtml
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```html
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```csharp
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```cshtml
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```html
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.

- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.

- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

Note

You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```html
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```csharp
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```cshtml
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```html
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```cshtml
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```csharp
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```csharp
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```csharp
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

ⓘ **Note**

We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```csharp
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```csharp
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```cshtml
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```csharp
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```html
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

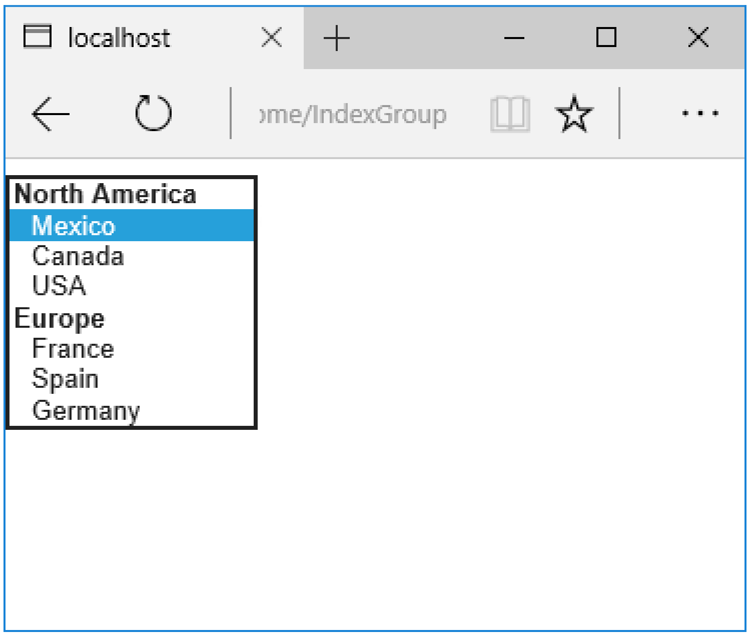
The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



North America
Mexico
Canada
USA
Europe
France
Spain
Germany

The generated HTML:

HTML

```html
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```csharp
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```cshtml
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```html
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for 
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```cshtml
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```cshtml
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```csharp
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Tag Helper Components in ASP.NET Core

Article • 06/18/2024

By Scott Addie and Fiyaz Bin Hasan

A Tag Helper Component is a Tag Helper that allows you to conditionally modify or add HTML elements from server-side code. This feature is available in ASP.NET Core 2.0 or later.

ASP.NET Core includes two built-in Tag Helper Components: `head` and `body`. They're located in the Microsoft.AspNetCore.Mvc.Razor.TagHelpers namespace and can be used in both MVC and Razor Pages. Tag Helper Components don't require registration with the app in `_ViewImports.cshtml`.

View or download sample code (how to download)

Use cases

Two common use cases of Tag Helper Components include:

1. Injecting a <link> into the <head>.
2. Injecting a <script> into the <body>.

The following sections describe these use cases.

Inject into HTML head element

Inside the HTML `<head>` element, CSS files are commonly imported with the HTML `<link>` element. The following code injects a `<link>` element into the `<head>` element using the `head` Tag Helper Component:

C#

```csharp
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressStyleTagHelperComponent : TagHelperComponent
    {
        private readonly string _style =
            @"<link rel=""stylesheet"" href=""/css/address.css"" />";

        public override int Order => 1;

        public override Task ProcessAsync(TagHelperContext context,
                                          TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "head",
                              StringComparison.OrdinalIgnoreCase))
            {
                output.PostContent.AppendHtml(_style);
            }

            return Task.CompletedTask;
        }
    }
}
```

In the preceding code:

- `AddressStyleTagHelperComponent` implements TagHelperComponent. The abstraction:
  - Allows initialization of the class with a TagHelperContext.
  - Enables the use of Tag Helper Components to add or modify HTML elements.
- The Order property defines the order in which the Components are rendered. `Order` is necessary when there are multiple usages of Tag Helper Components in an app.
- ProcessAsync compares the execution context's TagName property value to `head`. If the comparison evaluates to true, the content of the `_style` field is injected into the HTML `<head>` element.

Inject into HTML body element

The `body` Tag Helper Component can inject a `<script>` element into the `<body>` element. The following code demonstrates this technique:

C#

```csharp
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressScriptTagHelperComponent : TagHelperComponent
    {
        public override int Order => 2;

        public override async Task ProcessAsync(TagHelperContext context,
                                                TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "body",
                              StringComparison.OrdinalIgnoreCase))
            {
                var script = await File.ReadAllTextAsync(
                    "TagHelpers/Templates/AddressToolTipScript.html");
                output.PostContent.AppendHtml(script);
            }
        }
    }
}
```

A separate HTML file is used to store the `<script>` element. The HTML file makes the code cleaner and more maintainable. The preceding code reads the contents of `TagHelpers/Templates/AddressToolTipScript.html` and appends it with the Tag Helper output. The `AddressToolTipScript.html` file includes the following markup:

```HTML
<script>
$("address[printable]").hover(function() {
    $(this).attr({
        "data-toggle": "tooltip",
        "data-placement": "right",
        "title": "Home of Microsoft!"
    });
});
</script>
```

The preceding code binds a Bootstrap tooltip widget to any `<address>` element that includes a `printable` attribute. The effect is visible when a mouse pointer hovers over the element.

Register a Component

A Tag Helper Component must be added to the app's Tag Helper Components collection. There are three ways to add to the collection:

- Registration via services container
- Registration via Razor file
- Registration via Page Model or controller

Registration via services container

If the Tag Helper Component class isn't managed with ITagHelperComponentManager, it must be registered with the dependency injection (DI) system. The following `Startup.ConfigureServices` code registers the `AddressStyleTagHelperComponent` and `AddressScriptTagHelperComponent` classes with a transient lifetime:

C#

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddTransient<ITagHelperComponent,
        AddressScriptTagHelperComponent>();
    services.AddTransient<ITagHelperComponent,
        AddressStyleTagHelperComponent>();
}
```

Registration via Razor file

If the Tag Helper Component isn't registered with DI, it can be registered from a Razor Pages page or an MVC view. This technique is used for controlling the injected markup and the component execution order from a Razor file.

`ITagHelperComponentManager` is used to add Tag Helper Components or remove them from the app. The following code demonstrates this technique with `AddressTagHelperComponent`:

CSHTML

```cshtml
@using RazorPagesSample.TagHelpers;
@using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
@inject ITagHelperComponentManager manager;

@{
    string markup;

    if (Model.IsWeekend)
    {
        markup = "<em class='text-warning'>Office closed today!</em>";
    }
    else
    {
        markup = "<em class='text-info'>Office open today!</em>";
    }

    manager.Components.Add(new AddressTagHelperComponent(markup, 1));
}
```

In the preceding code:

- The `@inject` directive provides an instance of `ITagHelperComponentManager`. The instance is assigned to a variable named `manager` for access downstream in the Razor file.
- An instance of `AddressTagHelperComponent` is added to the app's Tag Helper Components collection.

`AddressTagHelperComponent` is modified to accommodate a constructor that accepts the `markup` and `order` parameters:

```C#
private readonly string _markup;

public override int Order { get; }

public AddressTagHelperComponent(string markup = "", int order = 1)
{
    _markup = markup;
    Order = order;
}
```

The provided `markup` parameter is used in `ProcessAsync` as follows:

```C#
public override async Task ProcessAsync(TagHelperContext context,
                                        TagHelperOutput output)
{
    if (string.Equals(context.TagName, "address",
            StringComparison.OrdinalIgnoreCase) &&
        output.Attributes.ContainsName("printable"))
    {
        TagHelperContent childContent = await output.GetChildContentAsync();
        string content = childContent.GetContent();
        output.Content.SetHtmlContent(
            $"<div>{content}<br>{_markup}</div>{_printableButton}");
    }
}
```

Registration via Page Model or controller

If the Tag Helper Component isn't registered with DI, it can be registered from a Razor Pages page model or an MVC controller. This technique is useful for separating C# logic from Razor files.

Constructor injection is used to access an instance of `ITagHelperComponentManager`. The Tag Helper Component is added to the instance's Tag Helper Components collection. The following Razor Pages page model demonstrates this technique with `AddressTagHelperComponent`:

```C#
using System;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesSample.TagHelpers;

public class IndexModel : PageModel
{
    private readonly ITagHelperComponentManager _tagHelperComponentManager;

    public bool IsWeekend
    {
        get
        {
            var dayOfWeek = DateTime.Now.DayOfWeek;

            return dayOfWeek == DayOfWeek.Saturday ||
                   dayOfWeek == DayOfWeek.Sunday;
        }
    }

    public IndexModel(ITagHelperComponentManager tagHelperComponentManager)
    {
        _tagHelperComponentManager = tagHelperComponentManager;
    }

    public void OnGet()
    {
        string markup;

        if (IsWeekend)
        {
            markup = "<em class='text-warning'>Office closed today!</em>";
        }
        else
        {
            markup = "<em class='text-info'>Office open today!</em>";
        }

        _tagHelperComponentManager.Components.Add(
            new AddressTagHelperComponent(markup, 1));
    }
}
```

In the preceding code:

- Constructor injection is used to access an instance of `ITagHelperComponentManager`.
- An instance of `AddressTagHelperComponent` is added to the app's Tag Helper Components collection.

Create a Component

To create a custom Tag Helper Component:

- Create a public class deriving from TagHelperComponentTagHelper.
- Apply an [HtmlTargetElement] attribute to the class. Specify the name of the target HTML element.
- *Optional*: Apply an [EditorBrowsable(EditorBrowsableState.Never)] attribute to the class to suppress the type's display in IntelliSense.

The following code creates a custom Tag Helper Component that targets the `<address>` HTML element:

C#

```
using System.ComponentModel;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Microsoft.Extensions.Logging;

namespace RazorPagesSample.TagHelpers
{
    [HtmlTargetElement("address")]
    [EditorBrowsable(EditorBrowsableState.Never)]
    public class AddressTagHelperComponentTagHelper :
TagHelperComponentTagHelper
    {
        public AddressTagHelperComponentTagHelper(
            ITagHelperComponentManager componentManager,
            ILoggerFactory loggerFactory) : base(componentManager,
loggerFactory)
        {
        }
    }
}
```

Use the custom `address` Tag Helper Component to inject HTML markup as follows:

C#

```csharp
public class AddressTagHelperComponent : TagHelperComponent
{
    private readonly string _printableButton =
        "<button type='button' class='btn btn-info' onclick=\"window.open("
+
        "'https://binged.it/2AXRRYw')\">" +
        "<span class='glyphicon glyphicon-road' aria-hidden='true'></span>"
+
        "</button>";

    public override int Order => 3;

    public override async Task ProcessAsync(TagHelperContext context,
                                            TagHelperOutput output)
    {
        if (string.Equals(context.TagName, "address",
                StringComparison.OrdinalIgnoreCase) &&
            output.Attributes.ContainsName("printable"))
        {
            var content = await output.GetChildContentAsync();
            output.Content.SetHtmlContent(
                $"<div>{content.GetContent()}</div>{_printableButton}");
        }
    }
}
```

The preceding `ProcessAsync` method injects the HTML provided to SetHtmlContent into the matching `<address>` element. The injection occurs when:

- The execution context's `TagName` property value equals `address`.
- The corresponding `<address>` element has a `printable` attribute.

For example, the `if` statement evaluates to true when processing the following `<address>` element:

CSHTML

```cshtml
<address printable>
    One Microsoft Way<br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>
```

Additional resources

- Dependency injection in ASP.NET Core
- Dependency injection into views in ASP.NET Core
- ASP.NET Core built-in Tag Helpers

Anchor Tag Helper in ASP.NET Core

Article • 06/18/2024

By Peter Kellner and Scott Addie

The Anchor Tag Helper enhances the standard HTML anchor (`<a ... ></a>`) tag by adding new attributes. By convention, the attribute names are prefixed with `asp-`. The rendered anchor element's `href` attribute value is determined by the values of the `asp-` attributes.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

View or download sample code (how to download)

SpeakerController is used in samples throughout this document:

```C#
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;

public class SpeakerController : Controller
{
    private List<Speaker> Speakers =
        new List<Speaker>
        {
            new Speaker {SpeakerId = 10},
            new Speaker {SpeakerId = 11},
            new Speaker {SpeakerId = 12}
        };

    [Route("Speaker/{id:int}")]
    public IActionResult Detail(int id) =>
        View(Speakers.FirstOrDefault(a => a.SpeakerId == id));

    [Route("/Speaker/Evaluations",
           Name = "speakerevals")]
    public IActionResult Evaluations() => View();

    [Route("/Speaker/EvaluationsCurrent",
           Name = "speakerevalscurrent")]
    public IActionResult Evaluations(
        int speakerId,
        bool currentYear) => View();

    public IActionResult Index() => View(Speakers);
}

public class Speaker
```

```
{
    public int SpeakerId { get; set; }
}
```

Anchor Tag Helper attributes

asp-controller

The asp-controller attribute assigns the controller used for generating the URL. The following markup lists all speakers:

CSHTML

```cshtml
<a asp-controller="Speaker"
   asp-action="Index">All Speakers</a>
```

The generated HTML:

HTML

```html
<a href="/Speaker">All Speakers</a>
```

If the `asp-controller` attribute is specified and `asp-action` isn't, the default `asp-action` value is the controller action associated with the currently executing view. If `asp-action` is omitted from the preceding markup, and the Anchor Tag Helper is used in *HomeController*'s *Index* view (*/Home*), the generated HTML is:

HTML

```html
<a href="/Home">All Speakers</a>
```

asp-action

The asp-action attribute value represents the controller action name included in the generated `href` attribute. The following markup sets the generated `href` attribute value to the speaker evaluations page:

CSHTML

```cshtml
<a asp-controller="Speaker"
   asp-action="Evaluations">Speaker Evaluations</a>
```

The generated HTML:

```HTML
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

If no `asp-controller` attribute is specified, the default controller calling the view executing the current view is used.

If the `asp-action` attribute value is `Index`, then no action is appended to the URL, leading to the invocation of the default `Index` action. The action specified (or defaulted), must exist in the controller referenced in `asp-controller`.

asp-route-{value}

The asp-route-{value} attribute enables a wildcard route prefix. Any value occupying the `{value}` placeholder is interpreted as a potential route parameter. If a default route isn't found, this route prefix is appended to the generated `href` attribute as a request parameter and value. Otherwise, it's substituted in the route template.

Consider the following controller action:

```C#
private List<Speaker> Speakers =
    new List<Speaker>
    {
        new Speaker {SpeakerId = 10},
        new Speaker {SpeakerId = 11},
        new Speaker {SpeakerId = 12}
    };

[Route("Speaker/{id:int}")]
public IActionResult Detail(int id) =>
    View(Speakers.FirstOrDefault(a => a.SpeakerId == id));
```

With a default route template defined in *Startup.Configure*:

```C#
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
                    template: "
{area:exists}/{controller=Home}/{action=Index}");
```

```
    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

The MVC view uses the model, provided by the action, as follows:

CSHTML

```cshtml
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-id="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
</body>
</html>
```

The default route's `{id?}` placeholder was matched. The generated HTML:

HTML

```html
<a href="/Speaker/Detail/12">SpeakerId: 12</a>
```

Assume the route prefix isn't part of the matching routing template, as with the following MVC view:

CSHTML

```cshtml
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-speakerid="@Model.SpeakerId">SpeakerId:
@Model.SpeakerId</a>
<body>
</html>
```

The following HTML is generated because `speakerid` wasn't found in the matching route:

HTML

```
<a href="/Speaker/Detail?speakerid=12">SpeakerId: 12</a>
```

If either `asp-controller` or `asp-action` aren't specified, then the same default processing is followed as is in the `asp-route` attribute.

asp-route

The asp-route attribute is used for creating a URL linking directly to a named route. Using routing attributes, a route can be named as shown in the `SpeakerController` and used in its `Evaluations` action:

C#

```
[Route("/Speaker/Evaluations",
       Name = "speakerevals")]
```

In the following markup, the `asp-route` attribute references the named route:

CSHTML

```
<a asp-route="speakerevals">Speaker Evaluations</a>
```

The Anchor Tag Helper generates a route directly to that controller action using the URL */Speaker/Evaluations*. The generated HTML:

HTML

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

If `asp-controller` or `asp-action` is specified in addition to `asp-route`, the route generated may not be what you expect. To avoid a route conflict, `asp-route` shouldn't be used with the `asp-controller` and `asp-action` attributes.

asp-all-route-data

The asp-all-route-data attribute supports the creation of a dictionary of key-value pairs. The key is the parameter name, and the value is the parameter value.

In the following example, a dictionary is initialized and passed to a Razor view. Alternatively, the data could be passed in with your model.

CSHTML

```cshtml
@{
var parms = new Dictionary<string, string>
            {
                { "speakerId", "11" },
                { "currentYear", "true" }
            };
}

<a asp-route="speakerevalscurrent"
   asp-all-route-data="parms">Speaker Evaluations</a>
```

The preceding code generates the following HTML:

HTML

```html
<a href="/Speaker/EvaluationsCurrent?speakerId=11&currentYear=true">Speaker
Evaluations</a>
```

The `asp-all-route-data` dictionary is flattened to produce a querystring meeting the requirements of the overloaded `Evaluations` action:

C#

```csharp
public IActionResult Evaluations() => View();

[Route("/Speaker/EvaluationsCurrent",
       Name = "speakerevalscurrent")]
public IActionResult Evaluations(
```

If any keys in the dictionary match route parameters, those values are substituted in the route as appropriate. The other non-matching values are generated as request parameters.

asp-fragment

The asp-fragment attribute defines a URL fragment to append to the URL. The Anchor Tag Helper adds the hash character (#). Consider the following markup:

CSHTML

```cshtml
<a asp-controller="Speaker"
   asp-action="Evaluations"
   asp-fragment="SpeakerEvaluations">Speaker Evaluations</a>
```

The generated HTML:

```HTML
<a href="/Speaker/Evaluations#SpeakerEvaluations">Speaker Evaluations</a>
```

Hash tags are useful when building client-side apps. They can be used for easy marking and searching in JavaScript, for example.

asp-area

The asp-area attribute sets the area name used to set the appropriate route. The following examples depict how the `asp-area` attribute causes a remapping of routes.

Usage in Razor Pages

Razor Pages areas are supported in ASP.NET Core 2.1 or later.

Consider the following directory hierarchy:

- **{Project name}**
  - **wwwroot**
  - **Areas**
    - **Sessions**
      - **Pages**
        - *_ViewStart.cshtml*
        - `Index.cshtml`
        - `Index.cshtml.cs`
  - **Pages**

The markup to reference the *Sessions* area *Index* Razor Page is:

```CSHTML
<a asp-area="Sessions"
   asp-page="/Index">View Sessions</a>
```

The generated HTML:

```HTML
<a href="/Sessions">View Sessions</a>
```

Tip

To support areas in a Razor Pages app, do one of the following in `Startup.ConfigureServices`:

- Set the compatibility version to 2.1 or later.
- Set the RazorPagesOptions.AllowAreas property to `true`:

  C#

  ```
  services.AddMvc()
          .AddRazorPagesOptions(options => options.AllowAreas =
  true);
  ```

Usage in MVC

Consider the following directory hierarchy:

- **{Project name}**
  - **wwwroot**
  - **Areas**
    - **Blogs**
      - **Controllers**
        - `HomeController.cs`
      - **Views**
        - **Home**
          - `AboutBlog.cshtml`
          - `Index.cshtml`
        - *_ViewStart.cshtml*
  - **Controllers**

Setting `asp-area` to "Blogs" prefixes the directory *Areas/Blogs* to the routes of the associated controllers and views for this anchor tag. The markup to reference the *AboutBlog* view is:

CSHTML

```
<a asp-area="Blogs"
   asp-controller="Home"
   asp-action="AboutBlog">About Blog</a>
```