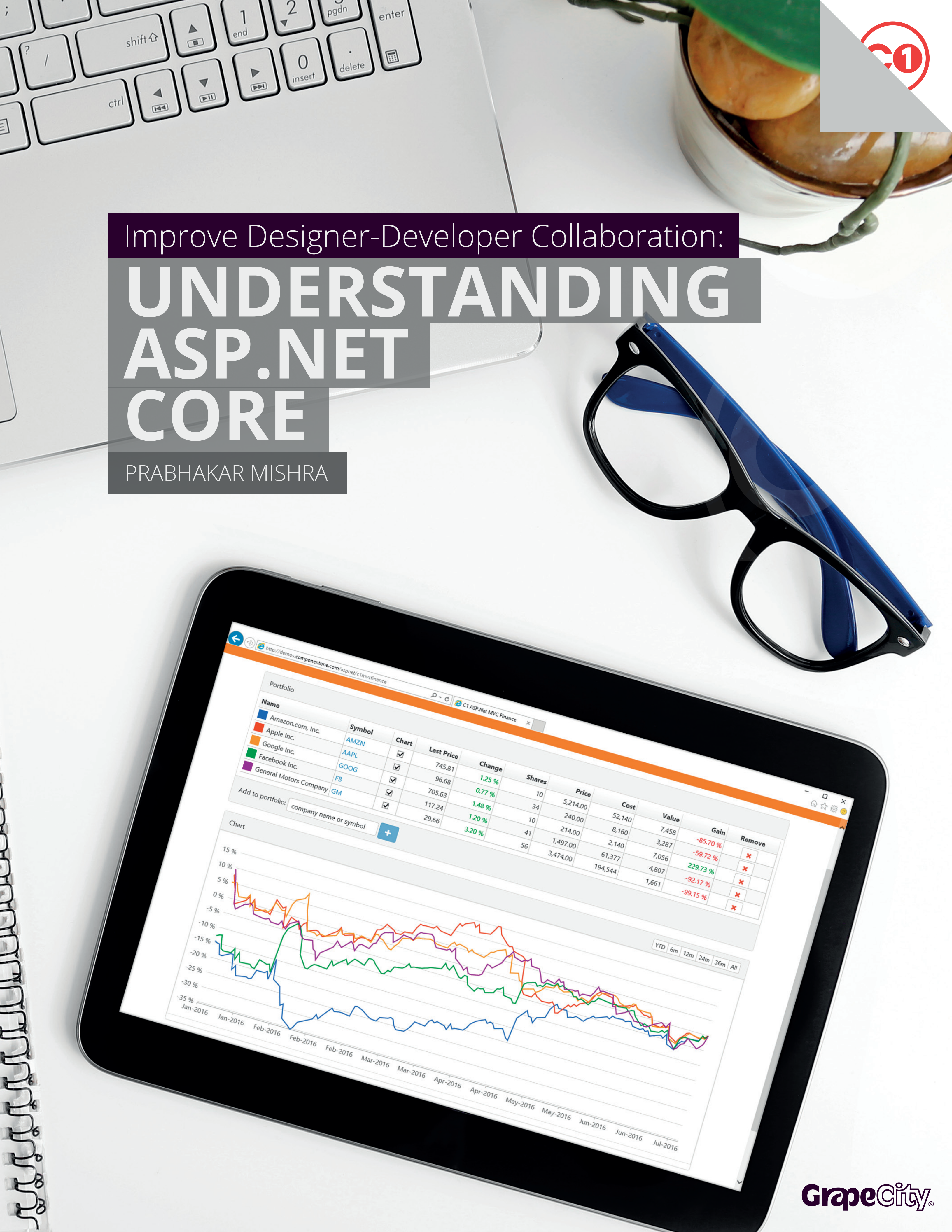
Improve Designer-Developer Collaboration:

UNDERSTANDING ASP.NET CORE

PRABHAKAR MISHRA



Name	Symbol	Chart	Last Price	Change	Shares	Price	Cost	Value	Gain	Remove
🟦 Amazon.com, Inc.	AMZN	✓	745.81	1.25 %	10	5,214.00	52,140	7,458	-85.70 %	✖
🟥 Apple Inc.	AAPL	✓	96.68	0.77 %	34	240.00	8,160	3,287	-59.72 %	✖
🟧 Google Inc.	GOOG	✓	705.63	1.48 %	10	214.00	2,140	7,056	229.73 %	✖
🟩 Facebook Inc.	FB	✓	117.24	1.20 %	41	1,497.00	61,377	4,807	-92.17 %	✖
🟪 General Motors Company	GM	✓	29.66	3.20 %	56	3,474.00	194,544	1,661	-99.15 %	✖

GrapeCity

Abstract

ASP.NET Core is a new technology for building cross-platform web applications. These applications could be developed on Windows, Mac or Linux and can be deployed on Windows or Linux servers.

ASP.NET Core's TagHelper feature provides a readable, HTML-like markup that enables developers and web designers to collaborate more closely and efficiently. In this white paper we'll walk you through the basics of TagHelpers, from concepts to custom development.

You'll learn:

1. What TagHelpers are
2. How to use TagHelpers in your project
3. How to create custom TagHelpers
4. How to create custom TagHelpers based on popular libraries such as jQuery and Bootstrap
5. How ComponentOne ASP.NET MVC Edition integrates TagHelpers

Throughout the paper, you'll see HtmlHelper code compared to TagHelper code, and identified by small icons: @ for HtmlHelper and <> for TagHelper.

Table of Contents

1. What are TagHelpers?

As one of the best new features of ASP.NET Core, TagHelpers simplify the work required to design views that interact with the data model. You can write HTML that not only binds to the data model, but also offers the advantages of HTML and CSS. TagHelpers allow you to provide robust, maintainable, reusable code with information provided on the server.

In a nutshell, TagHelpers allow server-side code to create and render HTML elements in Razor views. Previously, you'd write HtmlHelper code inside Razor to create views written in C# or VB. Although it could be written in markup, the markup did not always provide HtmlHelper advantages like model binding.

TagHelpers, however, provide advantages of both HtmlHelpers and markup. MVC has many built-in TagHelpers for various HTML elements, ranging from inputs and forms to asset libraries. It's also possible to author custom TagHelpers in ASP.NET Core MVC using C#.

Developers coming from Web Forms backgrounds will find TagHelper syntax to be familiar, but that's the only commonality between TagHelpers and Web Forms controls. Where Web Forms controls need full access of the page and have a complex life cycle, TagHelpers can be thought of as directives on the server, similar to what Angular is for client-side programming.

1.1. TagHelper Advantages

- TagHelpers enable you to write robust server-side code that's easy for designers to understand. The HTML-like markup of TagHelpers is clear for non-developers conversant with HTML and CSS, and they need not know C#.
- The markup code for TagHelpers is self-explanatory; we'll look at differences between HtmlHelpers and TagHelpers in the next section.
- The built-in IntelliSense support enhances productivity.
- TagHelper is an opt-in feature. You can choose to use it or to opt out of it.

Let's get started with integrating TagHelpers into your projects.

1.2. Enabling TagHelpers in Projects

First, the “Microsoft.AspNetCore.Mvc.TagHelpers” package should be added to the project via NuGet. This is handled by the template when a project is created.

When a new project is created inside Visual Studio, the following code is added to _ViewImports.cshtml file by default:

```
@addTagHelper *, Microsoft.AspNet.Mvc.TagHelpers
```

This enables the built-in TagHelpers inside the project. Similarly, TagHelpers can be added to a project from a custom assembly:

```
@addTagHelper *, C1,AspNetCore.Mvc
```

To use a specific TagHelper from an assembly, the declaration should include the fully-qualified name of the TagHelper:

```
@addTagHelper C1.AspNetCore.Mvc.TagHelpers.FlexGridTagHelper,
C1. C1.AspNetCore.Mvc
```

1.3. Opting Out of TagHelpers

Use @removeTagHelper to stop using TagHelpers in a particular view. To specifically opt out of TagHelpers for certain tags, use the “!” character.

In this example, the asp-for TagHelper is not applied to the label:

```
<!label asp-for="Users.UserId">User Name</!label>
```

2. Understanding TagHelpers

Now that we've included TagHelpers in our project, let's look at some of the built-in TagHelpers to understand how they make life easy for developers and designers.

2.1. Built-In TagHelpers

ASP.NET Core has the following built-in TagHelpers:

- **Anchor**: Generates hyperlinks
- **Cache**: Manages partial page caching
- **Environment**: Controls content rendering based on the runtime environment
- **Form**: Generates form elements
- **Input**: Generates input elements
- **Label**: Outputs label elements
- **Link**: Processes link elements
- **Option**: Targets individual options in a select list
- **Script**: Processes script tags
- **Select**: Generates dropdown lists
- **TextArea**: Processes textarea tags
- **ValidationMessage**: Generates individual validation errors
- **ValidationSummary**: Renders the validation summary message

2.1.1. Input Control

We'll look at **input** in our example. Here's a typical input created using HtmlHelper:

```
@Html.EditorFor(m=>m.Users.UserId, new { @class="someCss"})
```

The advantage of this code is that we get model binding, but it's also hard to understand for a designer. In addition, class is a reserved word in C#—we can apply a style by using @class in HtmlHelpers, but we'll get no help with HTML styles in the editor, since what follows **class** is a string.

In contrast, the same code could be written using TagHelpers:

```
<input id="user" asp-for="Users.UserId" class="someCss" />
```

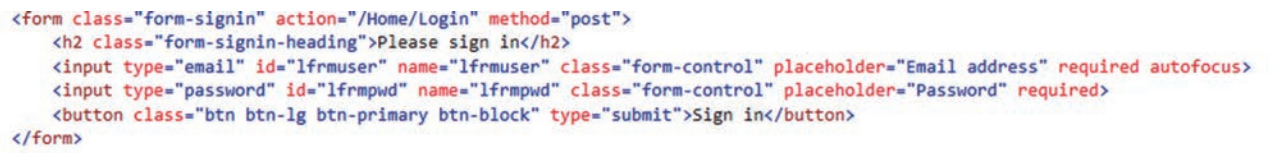
We achieve model-binding with the asp-for attribute, and we get a nice IntelliSense when we apply style to the input sans @class.

The above code translates to this HTML in the browser:

```
<input id="user" class="someCss" type="text" name="Users.UserId" value="" />
```

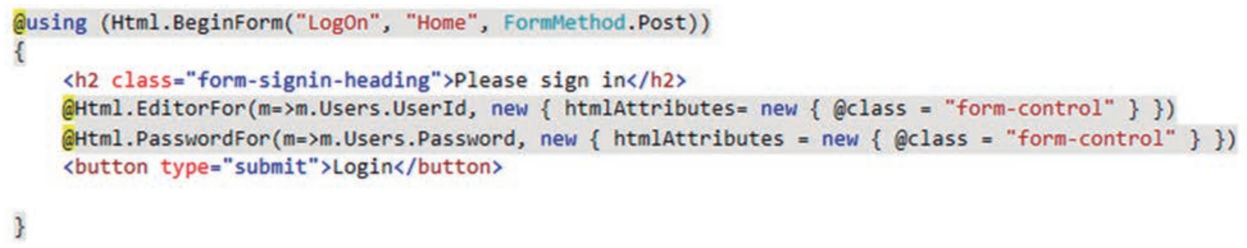
2.1.2. Form

Let's consider another example that involves a typical login screen using form TagHelpers. Here's the basic HTML markup:



```
<form class="form-signin" action="/Home/Login" method="post">
    <h2 class="form-signin-heading">Please sign in</h2>
    <input type="email" id="lfrmuser" name="lfrmuser" class="form-control" placeholder="Email address" required autofocus>
    <input type="password" id="lfrmpwd" name="lfrmpwd" class="form-control" placeholder="Password" required>
    <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
</form>
```

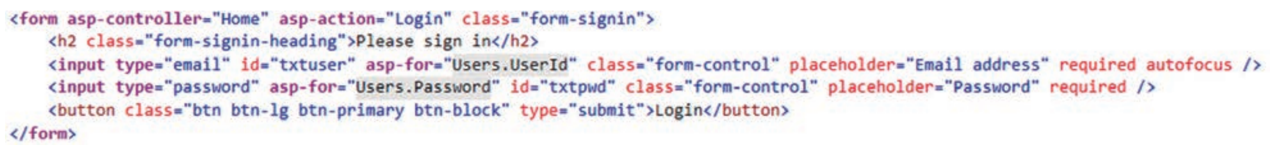
As you can see, we're missing model-binding. Here's the same form in HtmlHelpers, with model-binding:



```
@using (Html.BeginForm("LogOn", "Home", FormMethod.Post))
{
    <h2 class="form-signin-heading">Please sign in</h2>
    @Html.EditorFor(m=>m.Users.UserId, new { htmlAttributes= new { @class = "form-control" } })
    @Html.PasswordFor(m=>m.Users.Password, new { htmlAttributes = new { @class = "form-control" } })
    <button type="submit">Login</button>

}
```

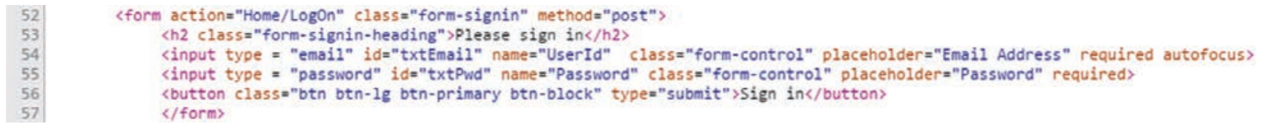
Now let's look at TagHelpers. ASP.NET Core has built-in TagHelpers that target attributes of the form element. For example, **asp-controller** assigns the controller to the form; **asp-action** assigns the action; and the **input** elements can have model binding.

Here's the TagHelpers version:



```
<form asp-controller="Home" asp-action="Login" class="form-signin">
    <h2 class="form-signin-heading">Please sign in</h2>
    <input type="email" id="txtuser" asp-for="Users.UserId" class="form-control" placeholder="Email address" required autofocus />
    <input type="password" asp-for="Users.Password" id="txtpwd" class="form-control" placeholder="Password" required />
    <button class="btn btn-lg btn-primary btn-block" type="submit">Login</button>
</form>
```

In the source generated from the TagHelper form, you can see that **asp-for** is translated to the **name** attribute:



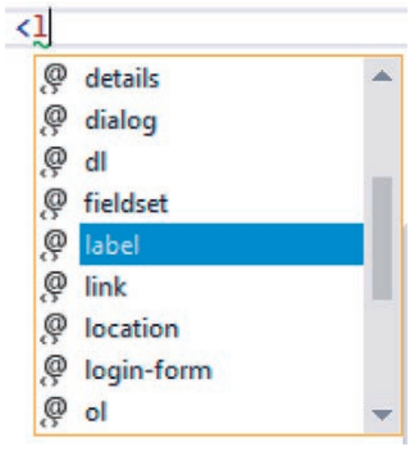
```
<form action="Home/LogOn" class="form-signin" method="post">
    <h2 class="form-signin-heading">Please sign in</h2>
    <input type = "email" id="txtEmail" name="UserId"  class="form-control" placeholder="Email Address" required autofocus>
    <input type = "password" id="txtPwd" name="Password" class="form-control" placeholder="Password" required>
    <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
    </form>
```

The generated source is similar to the form markup listed above. The TagHelpers version not only achieves our model-binding goal, but it's also similar enough to HTML markup that designers will understand what they're seeing.

2.1.3. IntelliSense and Display

You may have noticed that TagHelpers have a distinct font and color. This is because Visual Studio's Editor knows that it's a TagHelper, and thus displays markups targeted by TagHelpers in a distinctive font. This way the developer knows which is plain HTML and which elements are TagHelpers.

Apart from the distinctive font, Visual Studio provides rich IntelliSense for TagHelpers. For example, when you start typing a label, the IntelliSense shows an "@" sign and angular brackets "<>" for the label element. The Angular brackets mean that a TagHelper is available for the element.



As you type, the IntelliSense helps you write complete TagHelper code, displaying suggestions for attributes belonging to that TagHelper.

3. Authoring Custom TagHelpers

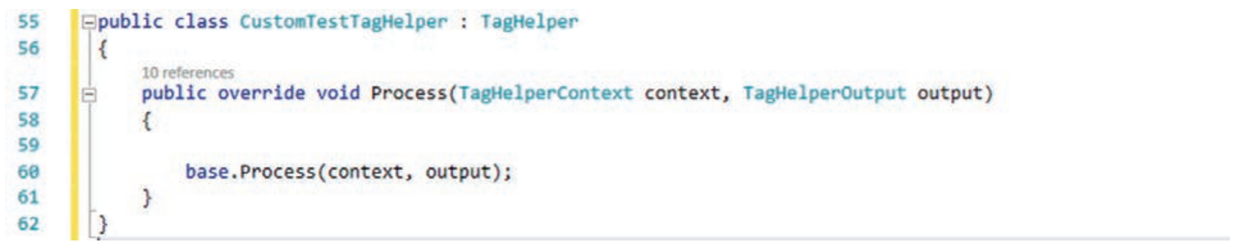
Now that we understand how TagHelpers work, we can create our own TagHelpers targeting HTML elements. Since the available set of TagHelpers is limited, it makes sense to create custom TagHelpers to solve complex problems.

We'll start with the basics of creating a TagHelper and then walk through creating reusable TagHelpers, from simple to complex.

3.1. TagHelper Basics

TagHelpers can be created by inheriting the TagHelper class, which resides in the Microsoft.AspNetCore.Razor.TagHelpers namespace. Inheriting from the TagHelper class allows us to override the process method that controls the behavior of the TagHelper.

Here's a simple custom TagHelper:



```
public class CustomTestTagHelper : TagHelper
{
    10 references
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {

        base.Process(context, output);
    }
}
```

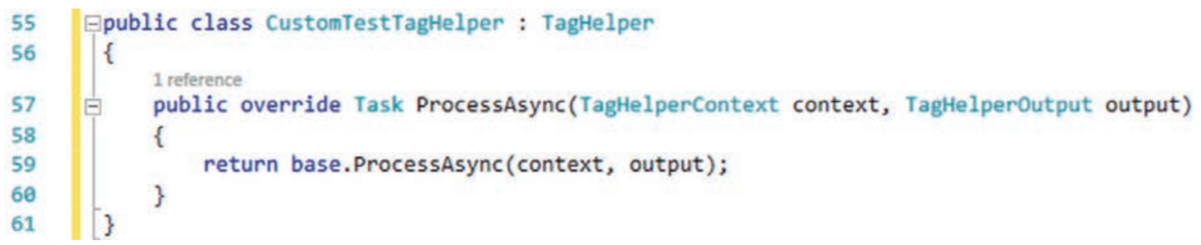
While including "TagHelper" in the name isn't required, adding it is an accepted convention. The TagHelper logic automatically detects and adjusts the class name accordingly. In this case, the TagHelper name would be translated to custom-test.

In cases where the TagHelper name is different than the class name, the class should be decorated with HtmlTargetElement.

The process method has two parameters.

- The **context parameter** contains information associated with the execution of current HTML tag and can be used to interact within nested TagHelpers for sharing context information.
- The **output parameter** contains a representation of the original source used to generate the HTML tag and content. The various properties of the output parameter—like PreContent, Content, and PostContent—help in rendering the final HTML.

The TagHelper class also has a method that helps with asynchronous execution of TagHelper processing.



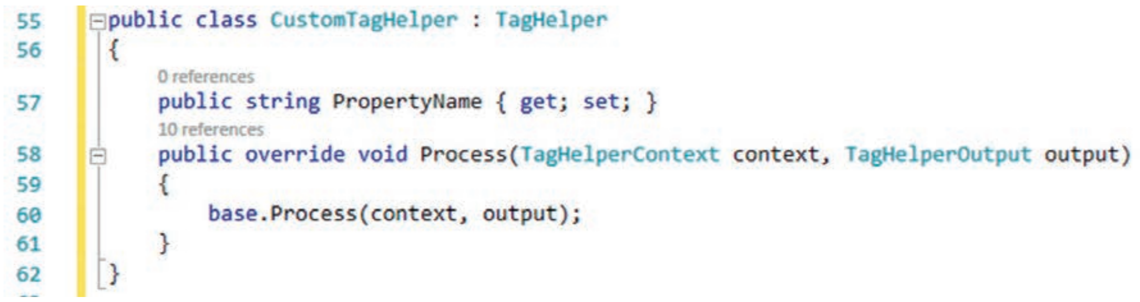
```
public class CustomTestTagHelper : TagHelper
{
    1 reference
    public override Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        return base.ProcessAsync(context, output);
    }
}
```

3.2. TagHelper Attributes

Attributes of a TagHelper are properties. In the example below, “PropertyName” is an attribute of the TagHelper “Custom,” and will be used as:

```
<custom property-name="testValue"></custom>
```

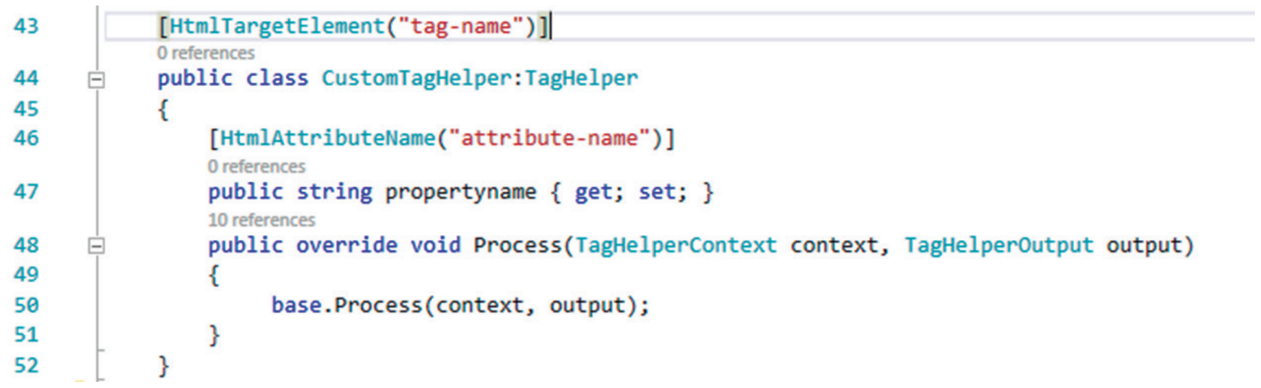
Code:



```
public class CustomTagHelper : TagHelper
{
    0 references
    public string PropertyName { get; set; }
    10 references
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        base.Process(context, output);
    }
}
```

To add a property to the class whose attribute name is different than the property name, it should be decorated with **HtmlAttributeName**.

In this example, the TagHelper is also decorated with HtmlTargetElement attribute since the name of TagHelper element is different than the class name.



```
[HtmlTargetElement("tag-name")]
0 references
public class CustomTagHelper:TagHelper
{
    [HtmlAttributeName("attribute-name")]
    0 references
    public string propertyname { get; set; }
    10 references
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        base.Process(context, output);
    }
}
```

The above TagHelper would be declared as:

```
<tag-name attribute-name="testValue"></tag-name>
```

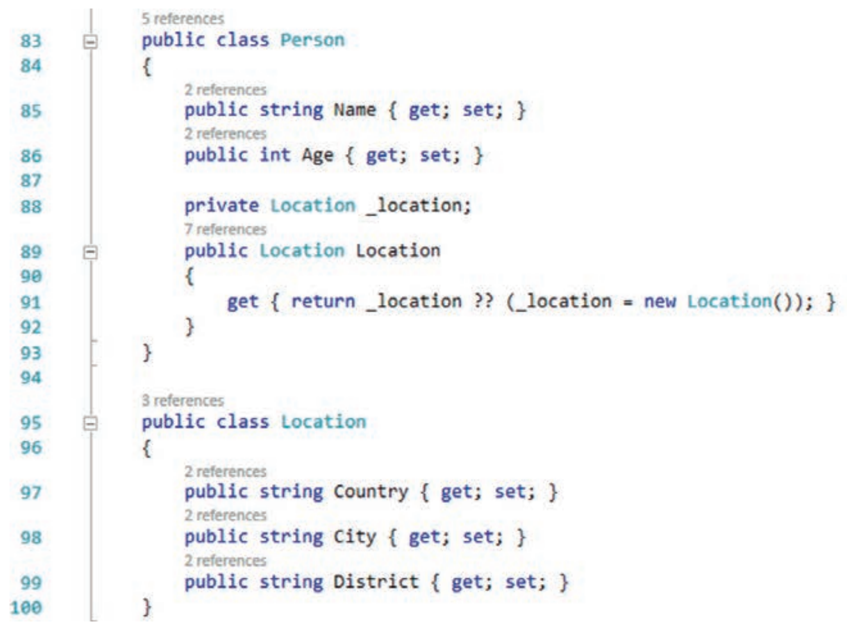
3.3. Nested TagHelpers

We've seen how basic TagHelpers can be created, but when building complex TagHelpers, it's best to break down the logic into several TagHelpers that communicate with each other. This communication is handled by the context parameter of the process method. The context parameter has an Items property that holds information for parent and child TagHelpers.

Let's take a simple example of **Person** and **Location**, where Location is the child TagHelper of Person. Here's the sample markup:

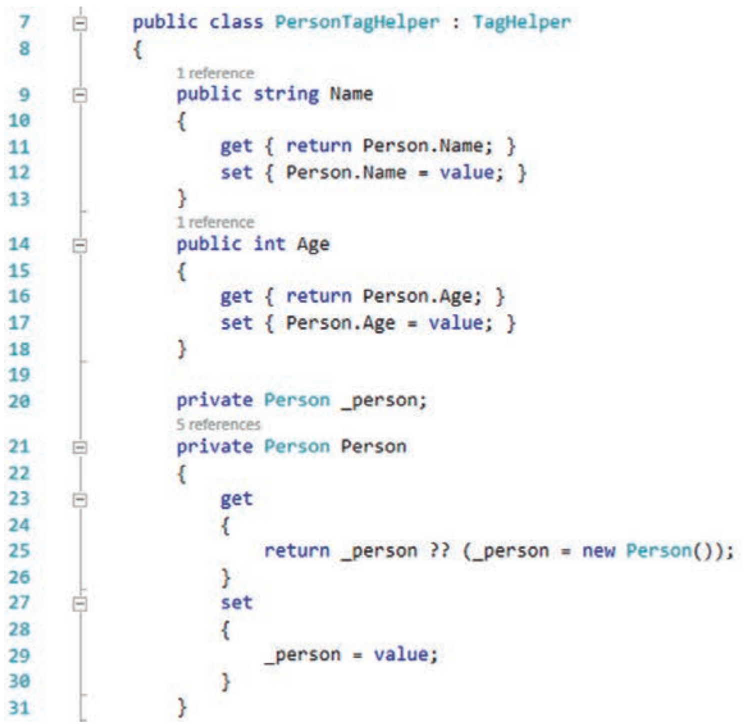
```
<person age="20" name="ABC">
      <location city="NY" country="US" district="NDNY"></location>
</person>
```

We consider two model classes:

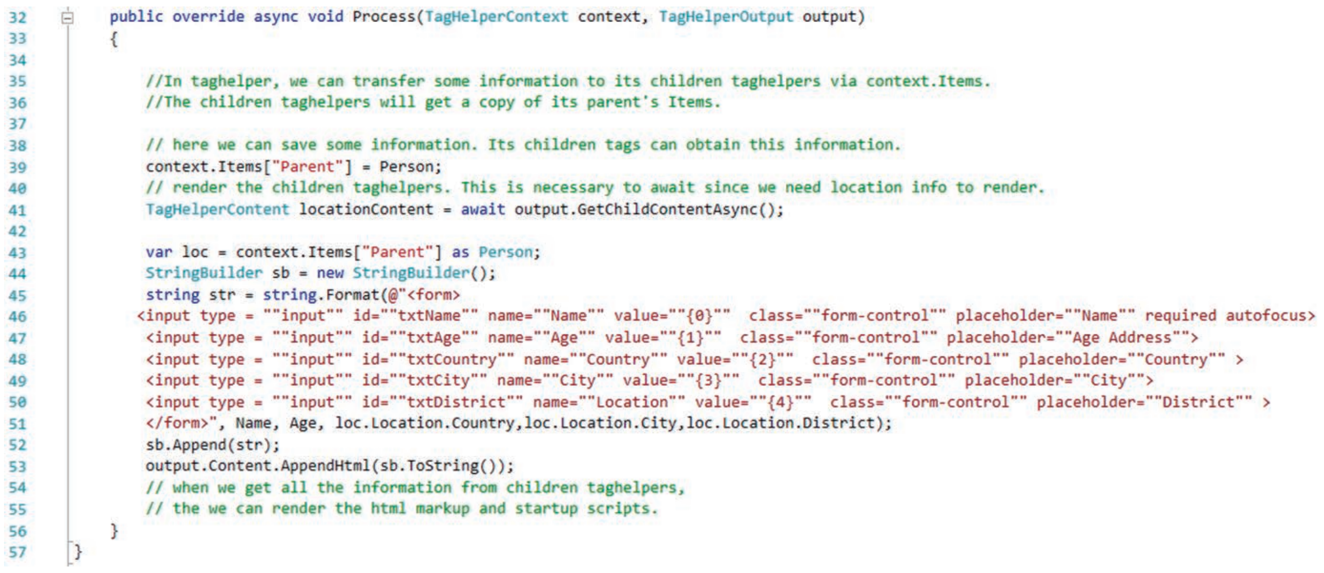


```
5 references
public class Person
{
    2 references
    public string Name { get; set; }
    2 references
    public int Age { get; set; }

    private Location _location;
    7 references
    public Location Location
    {
        get { return _location ?? (_location = new Location()); }
    }
}

3 references
public class Location
{
    2 references
    public string Country { get; set; }
    2 references
    public string City { get; set; }
    2 references
    public string District { get; set; }
}
```

The **Person** TagHelper class is defined below. Note how the context parameter is used to store the Person object's information. This Person object is then used by the child TagHelper **Location** to process location information. The output parameter's **GetChildContentAsync()** method helps to get the content of child TagHelper asynchronously.

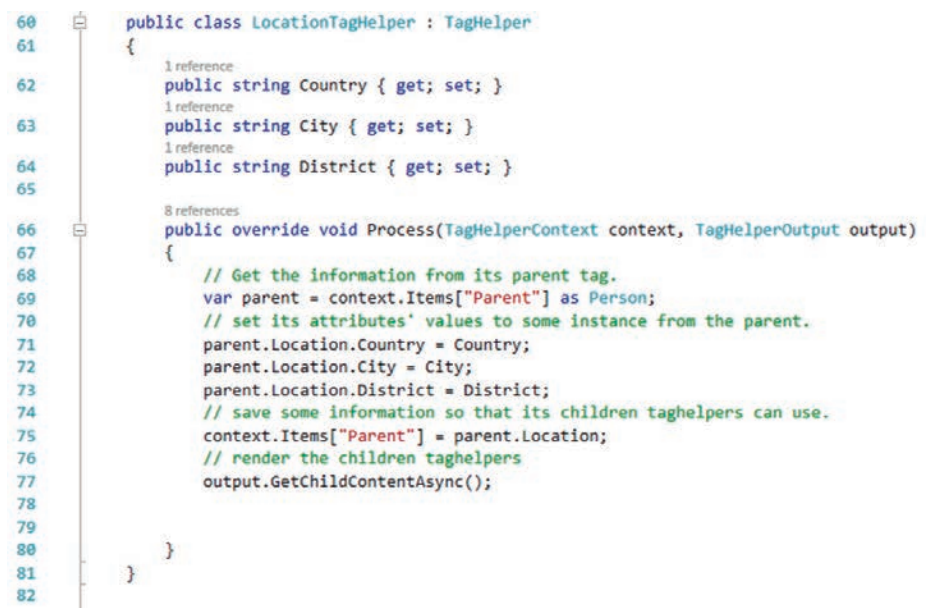


```
public class PersonTagHelper : TagHelper
{
    1 reference
    public string Name
    {
        get { return Person.Name; }
        set { Person.Name = value; }
    }
    1 reference
    public int Age
    {
        get { return Person.Age; }
        set { Person.Age = value; }
    }

    private Person _person;
    5 references
    private Person Person
    {
        get
        {
            return _person ?? (_person = new Person());
        }
        set
        {
            _person = value;
        }
    }
```



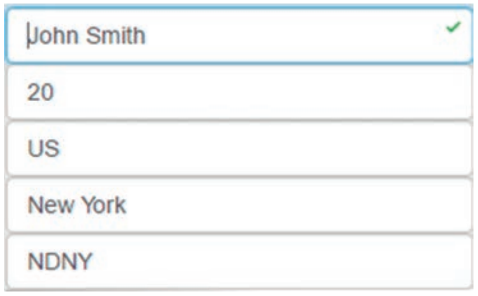
```
    public override async void Process(TagHelperContext context, TagHelperOutput output)
    {

        //In taghelper, we can transfer some information to its children taghelpers via context.Items.
        //The children taghelpers will get a copy of its parent's Items.

        // here we can save some information. Its children tags can obtain this information.
        context.Items["Parent"] = Person;
        // render the children taghelpers. This is necessary to await since we need location info to render.
        TagHelperContent locationContent = await output.GetChildContentAsync();

        var loc = context.Items["Parent"] as Person;
        StringBuilder sb = new StringBuilder();
        string str = string.Format(@"<form>
       <input type = ""input"" id=""txtName"" name=""Name"" value=""{0}""  class=""form-control"" placeholder=""Name"" required autofocus>
        <input type = ""input"" id=""txtAge"" name=""Age"" value=""{1}""  class=""form-control"" placeholder=""Age Address"">
        <input type = ""input"" id=""txtCountry"" name=""Country"" value=""{2}""  class=""form-control"" placeholder=""Country"" >
        <input type = ""input"" id=""txtCity"" name=""City"" value=""{3}""  class=""form-control"" placeholder=""City"">
        <input type = ""input"" id=""txtDistrict"" name=""Location"" value=""{4}""  class=""form-control"" placeholder=""District"" >
        </form>", Name, Age, loc.Location.Country,loc.Location.City,loc.Location.District);
        sb.Append(str);
        output.Content.AppendHtml(sb.ToString());
        // when we get all the information from children taghelpers,
        // the we can render the html markup and startup scripts.
    }
}
```

The Location TagHelper class is defined below. This child TagHelper gets the information of shared Parent object through the Context.Items collection:



```
public class LocationTagHelper : TagHelper
{
    1 reference
    public string Country { get; set; }
    1 reference
    public string City { get; set; }
    1 reference
    public string District { get; set; }

    8 references
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        // Get the information from its parent tag.
        var parent = context.Items["Parent"] as Person;
        // set its attributes' values to some instance from the parent.
        parent.Location.Country = Country;
        parent.Location.City = City;
        parent.Location.District = District;
        // save some information so that its children taghelpers can use.
        context.Items["Parent"] = parent.Location;
        // render the children taghelpers
        output.GetChildContentAsync();


    }
}
```

Figure 1 shows a working Nested TagHelper with input elements displaying Person and Location information inside a form:



John Smith

20

US

New York

NDNY

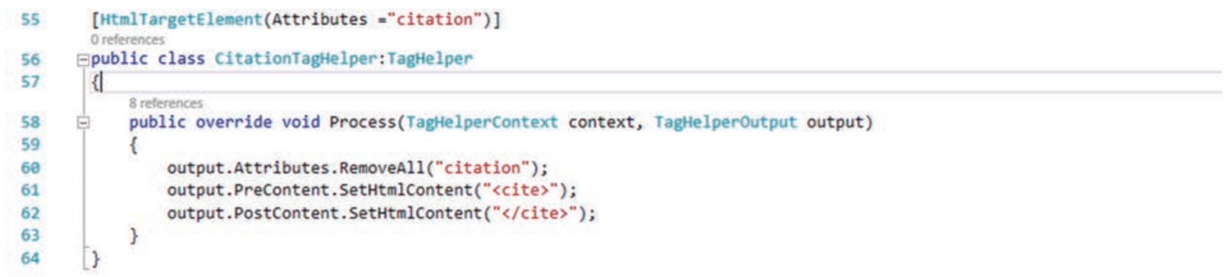
Figure 1: A working Nested TagHelper with input elements inside a form

3.4. Creating a Simple TagHelper

Now that we've learned the basics of creating a TagHelper, let's look at some examples of TagHelpers targeting HTML attributes.

In this example, we create a TagHelper for an HTML attribute that, when applied to an HTML paragraph, converts the content to a citation. A citation begins with `<cite>` and ends with `</cite>` tags in HTML; we wish to use the "citation" attribute in a paragraph to convert the content.

The logic of the process method would add a PreContent and PostContent to the rendered HTML wherever it finds the citation attribute.



```
[HtmlTargetElement(Attributes ="citation")]
0 references
public class CitationTagHelper:TagHelper
{
    8 references
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.Attributes.RemoveAll("citation");
        output.PreContent.SetHtmlContent("<cite>");
        output.PostContent.SetHtmlContent("</cite>");
    }
}
```

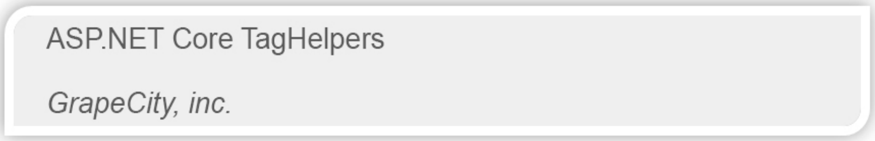
3.4.1. Usage

```
<p>ASP.NET Core TagHelpers</p>
<p citation>GrapeCity, inc.</p>
```

3.4.2. Generated HTML

```
<p>ASP.NET Core TagHelpers</p>
<p><cite>GrapeCity, inc.</cite></p>
```

The browser renders the TagHelper as normal HTML citation:



ASP.NET Core TagHelpers

GrapeCity, inc.

Figure 2: A simple citation TagHelper

3.5. Creating a Login Form TagHelper

Let's consider a slightly more complex example of a form TagHelper. In this scenario, we wish to write just one line of TagHelper code to generate a login form that not only binds the fields to a model, but also displays a form with a nice Bootstrap style applied.

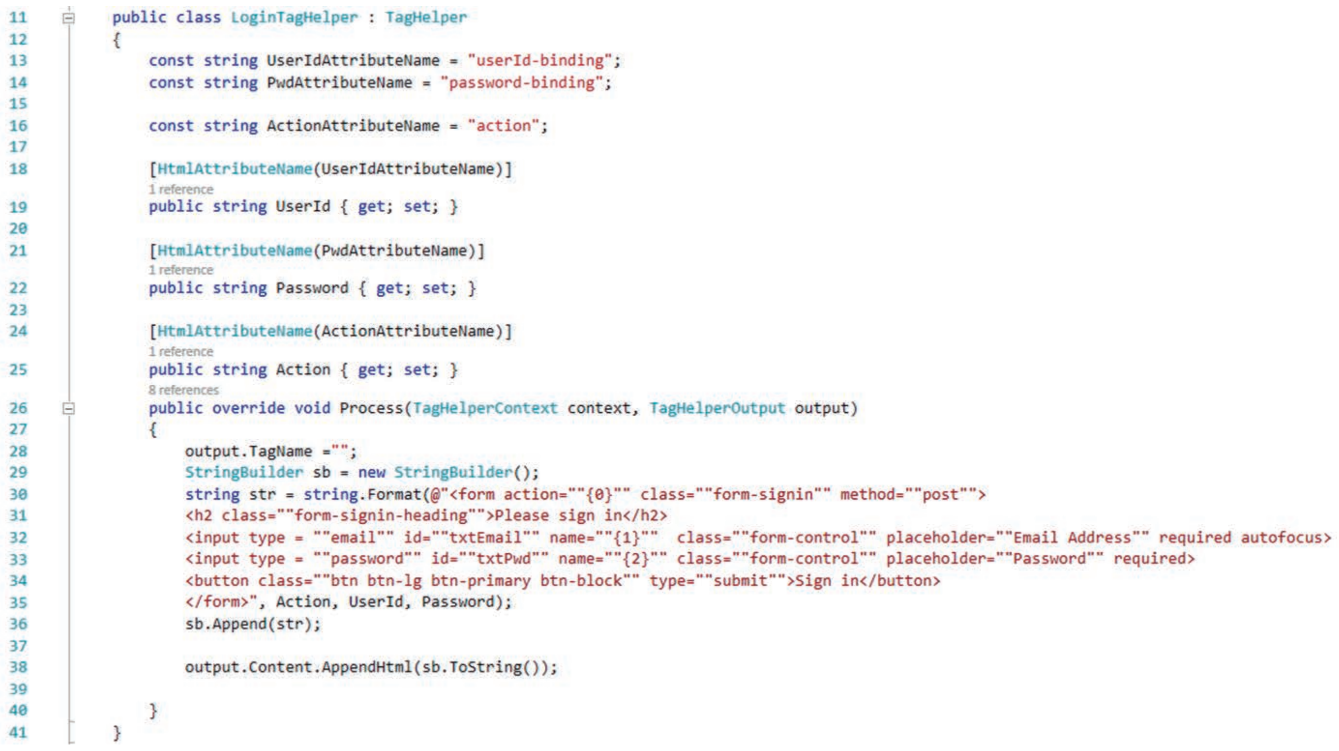
A form TagHelper with an email and password field would have three attributes defined:

1. User ID
2. Password
3. An attribute to define the controller and action

We will define properties to effectively manage the binding.

The process method will include the required markup for the form and define properties applied to appropriate attributes. (For example, the action property is set to action attribute, along with respective name attributes for userid and password fields.)

Finally, the output sets the created HTML string in place of the TagHelper:



```
public class LoginTagHelper : TagHelper
{
    const string UserIdAttributeName = "userId-binding";
    const string PwdAttributeName = "password-binding";

    const string ActionAttributeName = "action";

    [HtmlAttributeName(UserIdAttributeName)]
    1 reference
    public string UserId { get; set; }

    [HtmlAttributeName(PwdAttributeName)]
    1 reference
    public string Password { get; set; }

    [HtmlAttributeName(ActionAttributeName)]
    1 reference
    public string Action { get; set; }
    8 references
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName ="";
        StringBuilder sb = new StringBuilder();
        string str = string.Format(@"<form action=""{0}"" class=""form-signin"" method=""post"">
        <h2 class=""form-signin-heading"">Please sign in</h2>
        <input type = ""email"" id=""txtEmail"" name=""{1}""  class=""form-control"" placeholder=""Email Address"" required autofocus>
        <input type = ""password"" id=""txtPwd"" name=""{2}"" class=""form-control"" placeholder=""Password"" required>
        <button class=""btn btn-lg btn-primary btn-block"" type=""submit"">Sign in</button>
        </form>", Action, UserId, Password);
        sb.Append(str);

        output.Content.AppendHtml(sb.ToString());

    }
}
```

3.5.1. Usage

```
<login-form action="Home/LogOn" userId-binding="UserId"
password-binding="Password" ></login-form>
```

3.5.2. Rendered TagHelper

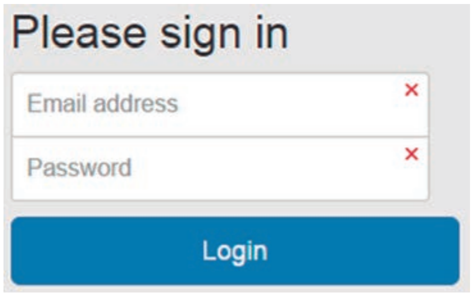


Figure 3: A login form using TagHelpers

Up to this point, we've worked with TagHelpers that target HTML elements and involve manipulation of HTML attributes. Next we'll consider advanced TagHelpers, targeting jQuery UI widgets and Bootstrap components that handle both HTML and JavaScript.

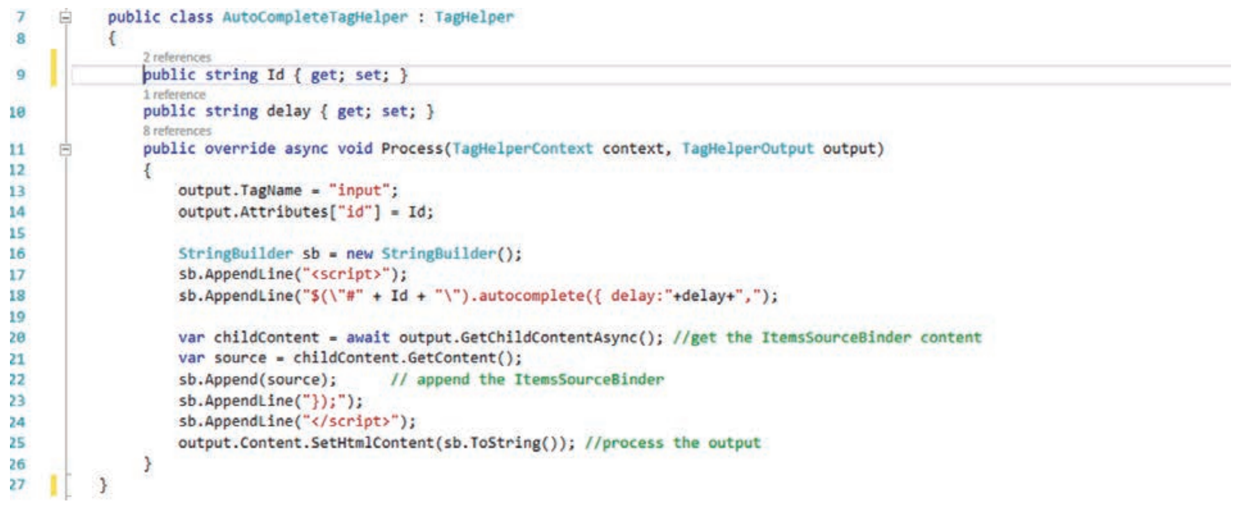
3.6. Creating a TagHelper for a jQuery UI Widget: AutoComplete

jQuery UI has an AutoComplete widget that can bind to a JavaScript array. In this example, we create a TagHelper for the AutoComplete widget, whose minimum set of options are id and source.

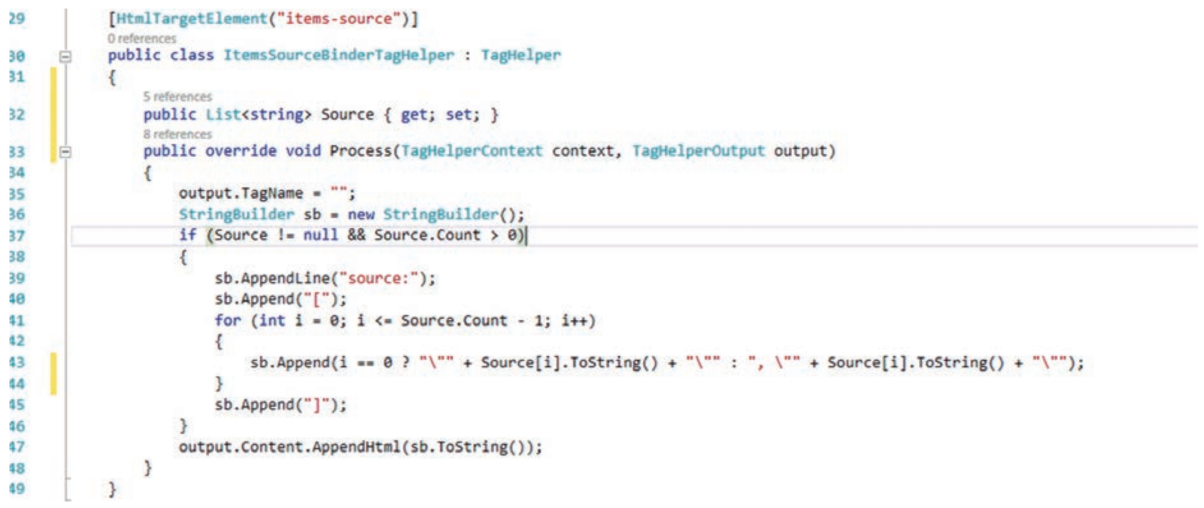
This example also demonstrates a nested TagHelper: ItemsSourceBinderTagHelper is a child of AutoComplete TagHelper.

The process method does the following:

1. Initializes an input element with provided id and attaches a script.
2. This script initializes an AutoComplete jQuery UI widget.
3. For each item in itemssource or datasource, the child ItemsSourceBinder TagHelper adds an item to the source option of AutoComplete.
4. The parent AutoComplete TagHelper processes the child TagHelper and appends its content to the output.
5. Finally, the output sets the HTML content for the TagHelper.



```
public class AutoCompleteTagHelper : TagHelper
{
    2 references
    public string Id { get; set; }
    1 reference
    public string delay { get; set; }
    8 references
    public override async void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "input";
        output.Attributes["id"] = Id;

        StringBuilder sb = new StringBuilder();
        sb.AppendLine("<script>");
        sb.AppendLine("$(\"#" + Id + "\").autocomplete({ delay:"+delay+",");

        var childContent = await output.GetChildContentAsync(); //get the ItemsSourceBinder content
        var source = childContent.GetContent();
        sb.Append(source);      // append the ItemsSourceBinder
        sb.AppendLine("});");
        sb.AppendLine("</script>");
        output.Content.SetHtmlContent(sb.ToString()); //process the output
    }
}
```


```
[HtmlTargetElement("items-source")]
0 references
public class ItemsSourceBinderTagHelper : TagHelper
{
    5 references
    public List<string> Source { get; set; }
    8 references
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "";
        StringBuilder sb = new StringBuilder();
        if (Source != null && Source.Count > 0)
        {
            sb.AppendLine("source:");
            sb.Append("[");
            for (int i = 0; i <= Source.Count - 1; i++)
            {
                sb.Append(i == 0 ? "\"" + Source[i].ToString() + "\"" : ", \"" + Source[i].ToString() + "\"");
            }
            sb.Append("]");
        }
        output.Content.AppendHtml(sb.ToString());
    }
}
```

In this example, the two TagHelpers do not share information, but it's easy to implement that through context parameter. (Refer to the nested TagHelpers section.) Nesting can help when there are more attributes set for AutoComplete, or when ItemsSource is more complex and has other attributes like service binding, sort expressions, update, or delete attributes.

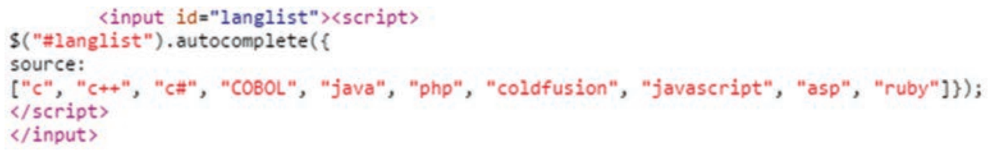
3.6.1. Dependency

This example needs jQuery, jQueryUI and jQueryUI CSS to be referenced locally or via CDN inside the view_Layout.cshtml.

3.6.2. Usage

```
<auto-complete id="langlist">
    <items-source source="Model.Lang.LanguageList"></items-source>
</auto-complete>
```

3.6.3. Generated HTML


```
        <input id="langlist"><script>
$("#langlist").autocomplete({
source:
["c", "c++", "c#", "COBOL", "java", "php", "coldfusion", "javascript", "asp", "ruby"]});
</script>
</input>
```

3.6.4. Rendered AutoComplete TagHelper

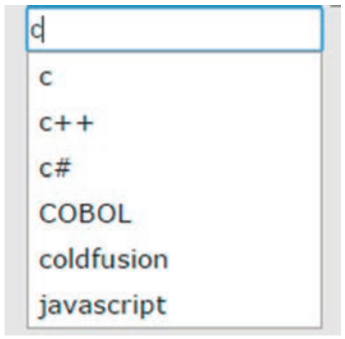
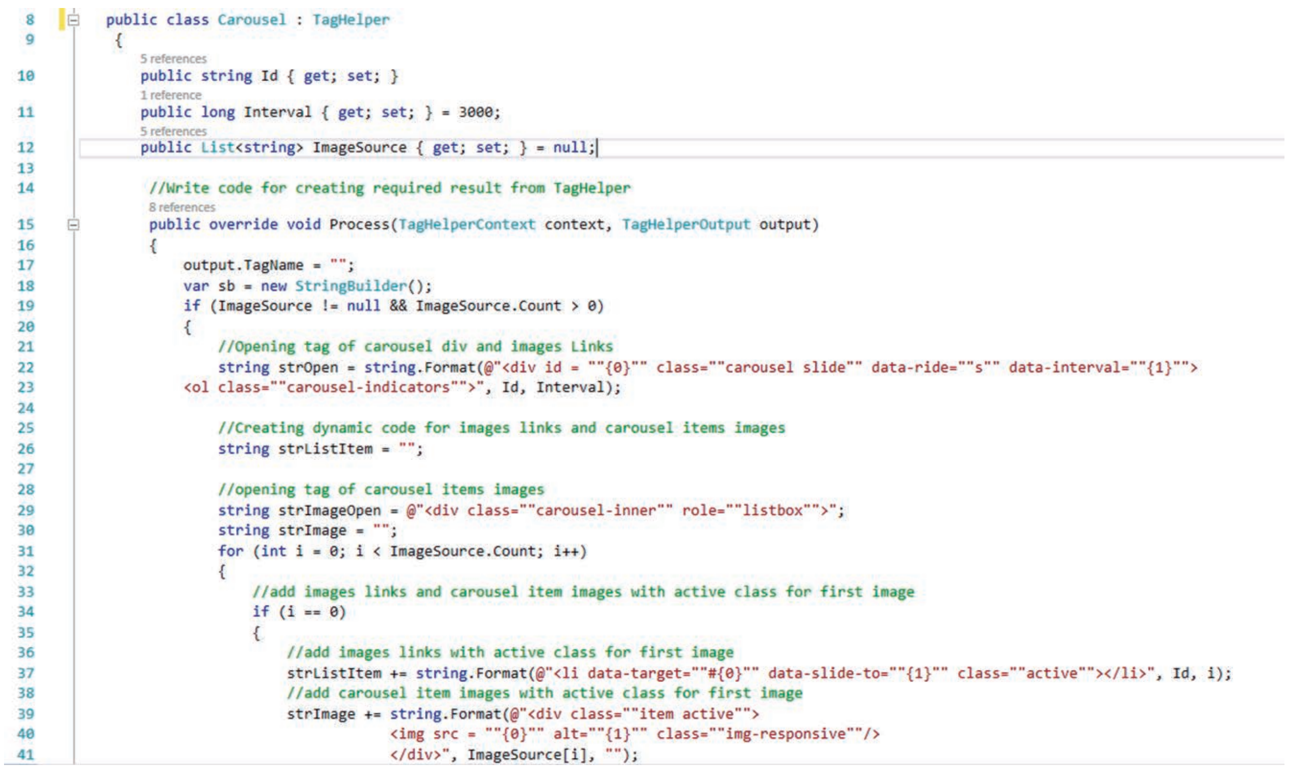


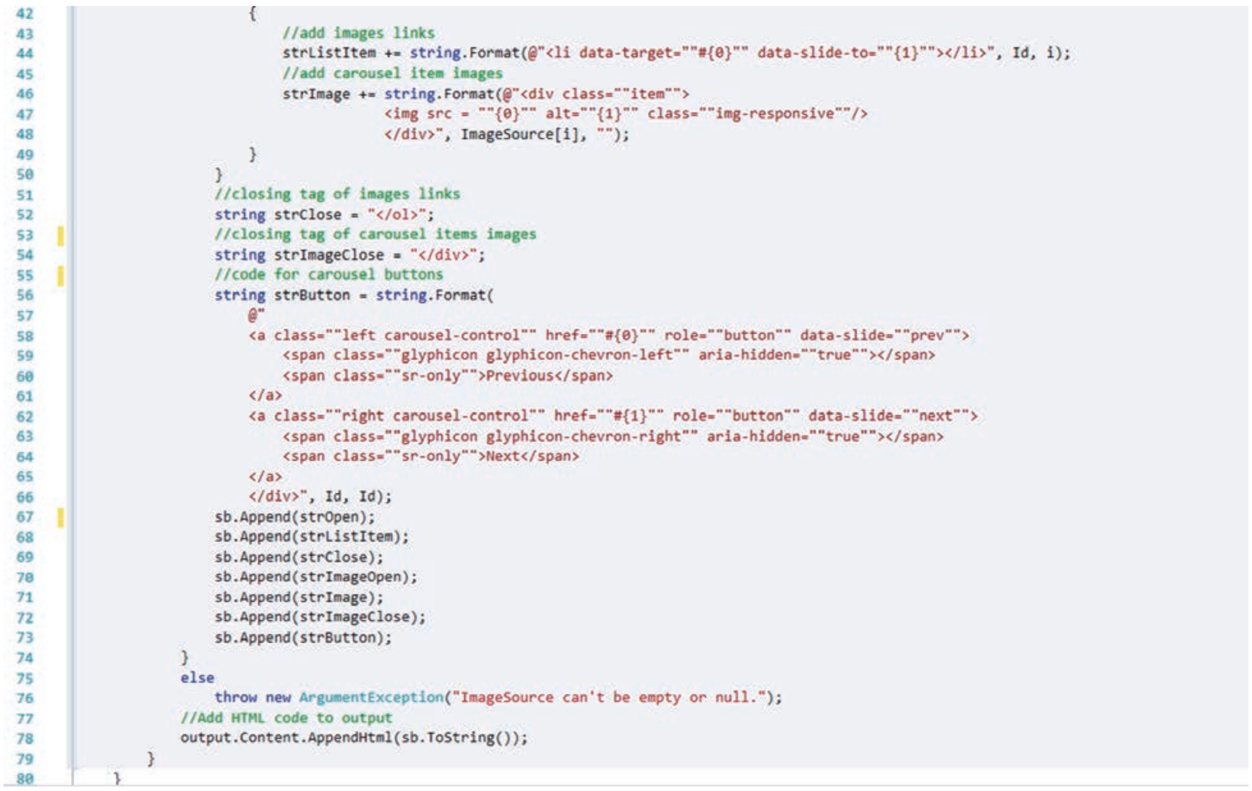
Figure 4: An AutoComplete TagHelper

3.7. TagHelper for a Bootstrap Component: Carousel

Bootstrap has a very good carousel component. Here, we'll consider creating a server-side TagHelper that displays images from a list. Generally, writing Bootstrap carousel code would involve around twenty lines of code. We'll create a reusable TagHelper that does it in one line.



```
public class Carousel : TagHelper
 {
    5 references
    public string Id { get; set; }
    1 reference
    public long Interval { get; set; } = 3000;
    5 references
    public List<string> ImageSource { get; set; } = null;

     //Write code for creating required result from TagHelper
     8 references
     public override void Process(TagHelperContext context, TagHelperOutput output)
     {
         output.TagName = "";
         var sb = new StringBuilder();
         if (ImageSource != null && ImageSource.Count > 0)
         {
             //Opening tag of carousel div and images Links
             string strOpen = string.Format(@"<div id = ""{0}"" class=""carousel slide"" data-ride=""s"" data-interval=""{1}"">
         <ol class=""carousel-indicators"">", Id, Interval);

             //Creating dynamic code for images links and carousel items images
             string strListItem = "";

             //opening tag of carousel items images
             string strImageOpen = @"<div class=""carousel-inner"" role=""listbox"">";
             string strImage = "";
             for (int i = 0; i < ImageSource.Count; i++)
             {
                 //add images links and carousel item images with active class for first image
                 if (i == 0)
                 {
                     //add images links with active class for first image
                     strListItem += string.Format(@"<li data-target=""#{0}"" data-slide-to=""{1}"" class=""active""></li>", Id, i);
                     //add carousel item images with active class for first image
                     strImage += string.Format(@"<div class=""item active"">
                                 <img src = ""{0}"" alt=""{1}"" class=""img-responsive""/>
                                 </div>", ImageSource[i], "");
```



```
                {
                    //add images links
                    strListItem += string.Format(@"<li data-target=""#{0}"" data-slide-to=""{1}""></li>", Id, i);
                    //add carousel item images
                    strImage += string.Format(@"<div class=""item"">
                                <img src = ""{0}"" alt=""{1}"" class=""img-responsive""/>
                                </div>", ImageSource[i], "");
                }
            }
            //closing tag of images links
            string strClose = "</ol>";
            //closing tag of carousel items images
            string strImageClose = "</div>";
            //code for carousel buttons
            string strButton = string.Format(
                @"
                <a class=""left carousel-control"" href=""#{0}"" role=""button"" data-slide=""prev"">
                    <span class=""glyphicon glyphicon-chevron-left"" aria-hidden=""true""></span>
                    <span class=""sr-only"">Previous</span>
                </a>
                <a class=""right carousel-control"" href=""#{1}"" role=""button"" data-slide=""next"">
                    <span class=""glyphicon glyphicon-chevron-right"" aria-hidden=""true""></span>
                    <span class=""sr-only"">Next</span>
                </a>
                </div>", Id, Id);
            sb.Append(strOpen);
            sb.Append(strListItem);
            sb.Append(strClose);
            sb.Append(strImageOpen);
            sb.Append(strImage);
            sb.Append(strImageClose);
            sb.Append(strButton);
        }
        else
            throw new ArgumentException("ImageSource can't be empty or null.");
        //Add HTML code to output
        output.Content.AppendHtml(sb.ToString());
    }
}
```

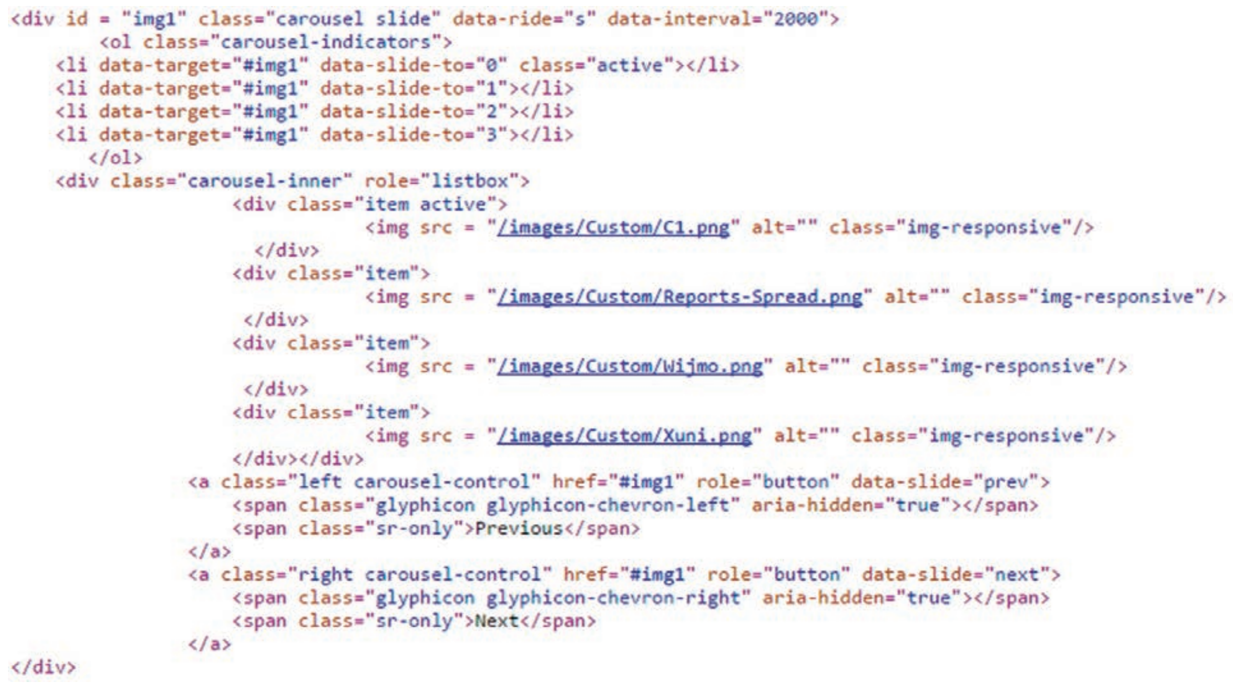
3.7.1. Dependency

This example needs Bootstrap referenced locally or via CDN inside the **view_Layout.cshtml**. Visual Studio automatically adds Bootstrap support for the project.

3.7.2. Usage

```
<carousel id="productImg" image-source="Model.ImageSource.ImageList"
interval="2000"/>
```

3.7.3. Generated HTML



```
<div id = "img1" class="carousel slide" data-ride="s" data-interval="2000">
        <ol class="carousel-indicators">
    <li data-target="#img1" data-slide-to="0" class="active"></li>
    <li data-target="#img1" data-slide-to="1"></li>
    <li data-target="#img1" data-slide-to="2"></li>
    <li data-target="#img1" data-slide-to="3"></li>
       </ol>
    <div class="carousel-inner" role="listbox">
                    <div class="item active">
                                <img src = "/images/Custom/C1.png" alt="" class="img-responsive"/>
                      </div>
                    <div class="item">
                                <img src = "/images/Custom/Reports-Spread.png" alt="" class="img-responsive"/>
                     </div>
                    <div class="item">
                                <img src = "/images/Custom/Wijmo.png" alt="" class="img-responsive"/>
                     </div>
                    <div class="item">
                                <img src = "/images/Custom/Xuni.png" alt="" class="img-responsive"/>
                    </div></div>
                <a class="left carousel-control" href="#img1" role="button" data-slide="prev">
                    <span class="glyphicon glyphicon-chevron-left" aria-hidden="true"></span>
                    <span class="sr-only">Previous</span>
                </a>
                <a class="right carousel-control" href="#img1" role="button" data-slide="next">
                    <span class="glyphicon glyphicon-chevron-right" aria-hidden="true"></span>
                    <span class="sr-only">Next</span>
                </a>
</div>
```

3.7.4. Rendered Carousel

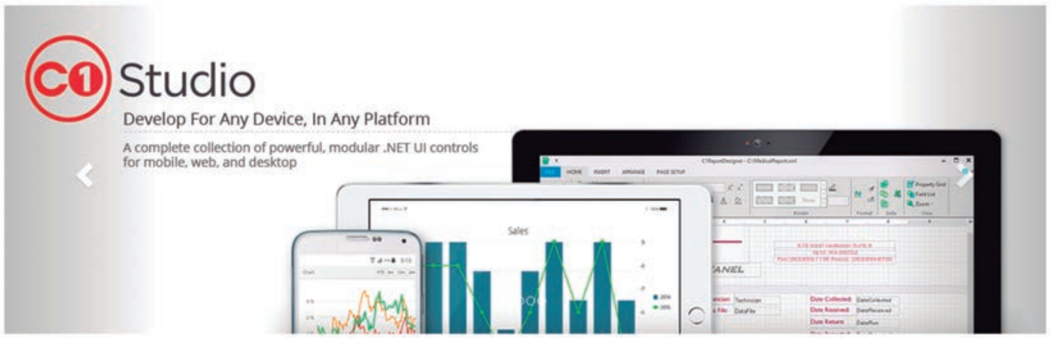

C1 Studio
Develop For Any Device, In Any Platform
A complete collection of powerful, modular .NET UI controls for mobile, web, and desktop
Sales

Figure 5: Carousel rendered with TagHelpers

We've seen how to create reusable TagHelpers based on HTML attributes and other libraries, but we don't always have the time and resources to build solutions involving complex logic and UI. Here, you might want to look into easy-to-use third-party controls that offer the advantages of TagHelpers. We'll cover that in the next section.

4. ComponentOne ASP.NET MVC Edition Controls

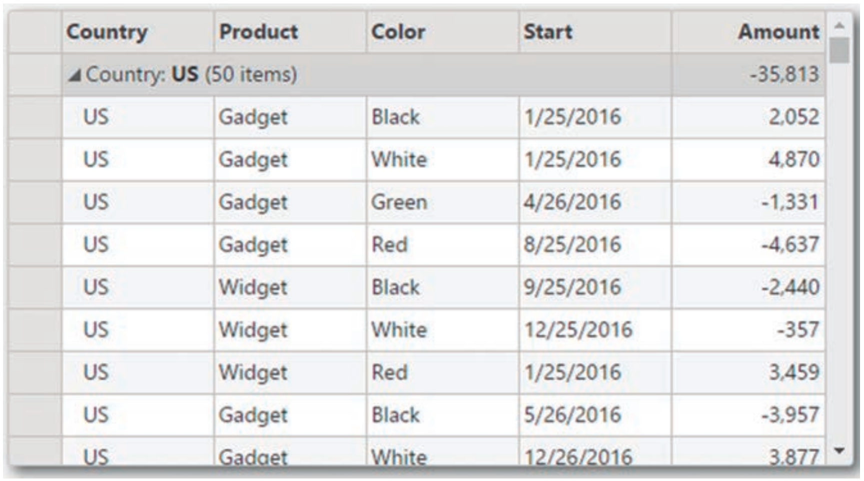
The ComponentOne ASP.NET MVC Edition controls have supported ASP.NET Core and TagHelpers since Beta 4. All controls provide rich IntelliSense to write server-side code, which makes coding in Razor views a breeze; even the scaffolders generate TagHelper code for all controls. The ASP.NET Core packages are available via the GrapeCity NuGet and require minimal configuration to get started.

Let's look at a few examples.

4.1. FlexGrid

FlexGrid for ASP.NET MVC is a powerful, full-featured grid with a small footprint that allows you to display, edit, summarize, format, and print data in a tabular format. With built-in and extensible features like data mapping, filtering, grouping, and virtualization, FlexGrid has a lot that your users need.

In this example, FlexGrid includes grouping and column definition. The items-source has the source-collection attribute set to current model data. The items-source also supports AJAX data through URL binding.



Country	Product	Color	Start	Amount
◢ Country: **US** (50 items)				-35,813
US	Gadget	Black	1/25/2016	2,052
US	Gadget	White	1/25/2016	4,870
US	Gadget	Green	4/26/2016	-1,331
US	Gadget	Red	8/25/2016	-4,637
US	Widget	Black	9/25/2016	-2,440
US	Widget	White	12/25/2016	-357
US	Widget	Red	1/25/2016	3,459
US	Gadget	Black	5/26/2016	-3,957
US	Gadget	White	12/26/2016	3,877

Figure 6: FlexGrid datagrid with grouping and columns

```
<c1-flex-grid id="gFlexGrid"
              auto-generate-columns="false"
              allow-sorting="true" group-by="Country">
    <c1-flex-grid-column binding="Country" width="*"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Product" width="*"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Color" width="*"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Start" width="*"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Amount" format="n0" aggregate="Sum" width="*"></c1-flex-grid-column>
    <c1-items-source source-collection="Model.CountryData"></c1-items-source>
</c1-flex-grid>
```

4.2. FlexChart

FlexChart is an SVG-based chart with a simple API and powerful data visualization features. Included is every popular chart type such as bar, column, line, area, scatter, spline, bubble, and financial. Customization options include built-in labels, markers, headers, annotations and footers, and you can also mix chart types and show data across multiple axes. Deep customization is possible using FlexChart's ItemFormatters and palettes, and you have access to individual data points using Hit Test.

This FlexChart depicts:

- Multiple series
- Chart types
- Legends

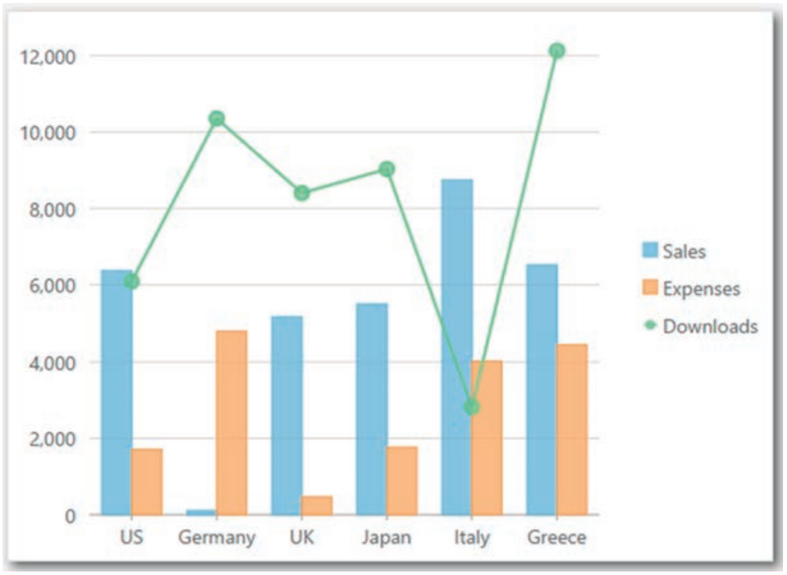


Figure 7: FlexChart with multiple series, legends and chart types

```
<c1-flex-chart id="mixedTypesChart" binding-x="Country">
    <c1-items-source source-collection="Model.CountrySalesData"></c1-items-source>
    <c1-flex-chart-series binding="Sales" name="Sales"></c1-flex-chart-series>
    <c1-flex-chart-series binding="Expenses" name="Expenses"></c1-flex-chart-series>
    <c1-flex-chart-series binding="Downloads" name="Downloads" chart-type="LineSymbols"></c1-flex-chart-series>
</c1-flex-chart>
...
```

A FlexChart with multiple axes defined:

```
<c1-flex-chart id="@Model.ControlId" header="Sales">
    <c1-flex-chart-series name="2013" binding-x="ID" binding="Count">
    <c1-items-source source-collection="@OrderDatas["2013"]"></c1-items-source>
    </c1-flex-chart-series>
     <c1-flex-chart-series name="2014" chart-type="LineSymbols" binding-x="ID" binding="Count">
    <c1-items-source source-collection="@OrderDatas["2014"]"></c1-items-source>
    <c1-flex-chart-axis c1-property="AxisY" position="Right"></c1-flex-chart-axis>
    </c1-flex-chart-series>
    <c1-flex-chart-axis c1-property="AxisX" binding="ID,Country">
    <c1-items-source source-collection="@OrderDatas["CountryNames"]"></c1-items-source>
    </c1-flex-chart-axis>
</c1-flex-chart>
```

4.3. ListBox

Next, let's look at a ListBox with custom template defined. Notice how easy it is to define the element bindings. The input controls can also be used inside a form.

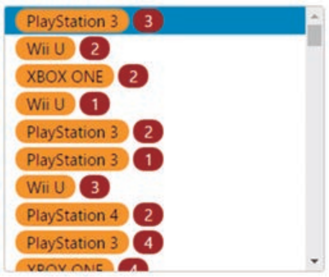
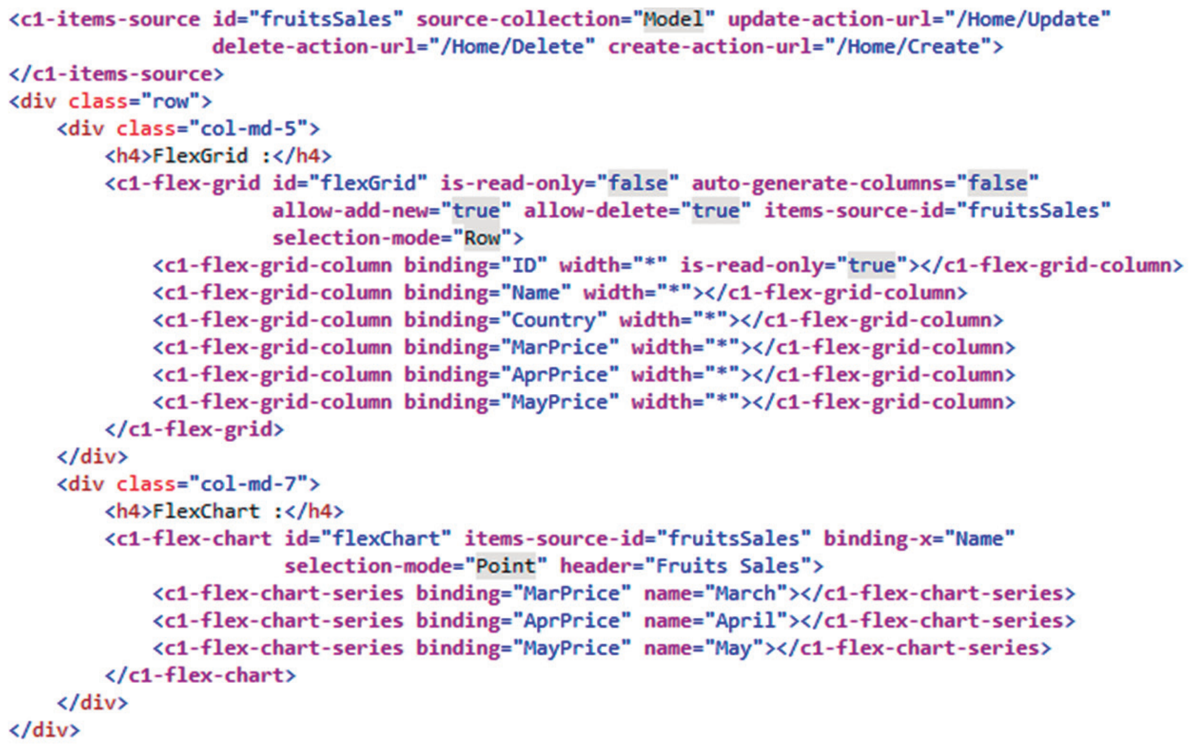


Figure 8: ListBox using a custom template

```
<c1-list-box width="300px" height="250px">
    <c1-input-item-template>
        <span>
            <span class="label">{{Product}}</span>
            <span class="badge">{{Count}}</span>
        </span>
    </c1-input-item-template>
    <c1-items-source source-collection="Model"></c1-items-source>
</c1-list-box>
```

4.4. Multiple Control Binding

When multiple controls share the same data source, any change in the data affects all the controls. In this example, FlexGrid and FlexChart are both bound to same "fruitSales" ItemsSource; any change to the data inside the grid redraws the affected series.



```
<c1-items-source id="fruitsSales" source-collection="Model" update-action-url="/Home/Update"
                 delete-action-url="/Home/Delete" create-action-url="/Home/Create">
</c1-items-source>
<div class="row">
    <div class="col-md-5">
        <h4>FlexGrid :</h4>
        <c1-flex-grid id="flexGrid" is-read-only="false" auto-generate-columns="false"
                      allow-add-new="true" allow-delete="true" items-source-id="fruitsSales"
                      selection-mode="Row">
            <c1-flex-grid-column binding="ID" width="*" is-read-only="true"></c1-flex-grid-column>
            <c1-flex-grid-column binding="Name" width="*"></c1-flex-grid-column>
            <c1-flex-grid-column binding="Country" width="*"></c1-flex-grid-column>
            <c1-flex-grid-column binding="MarPrice" width="*"></c1-flex-grid-column>
            <c1-flex-grid-column binding="AprPrice" width="*"></c1-flex-grid-column>
            <c1-flex-grid-column binding="MayPrice" width="*"></c1-flex-grid-column>
        </c1-flex-grid>
    </div>
    <div class="col-md-7">
        <h4>FlexChart :</h4>
        <c1-flex-chart id="flexChart" items-source-id="fruitsSales" binding-x="Name"
                       selection-mode="Point" header="Fruits Sales">
            <c1-flex-chart-series binding="MarPrice" name="March"></c1-flex-chart-series>
            <c1-flex-chart-series binding="AprPrice" name="April"></c1-flex-chart-series>
            <c1-flex-chart-series binding="MayPrice" name="May"></c1-flex-chart-series>
        </c1-flex-chart>
    </div>
</div>
```

5. HtmlHelpers to TagHelpers: C1Finance Application

So far in this whitepaper, we've looked at the basics of using TagHelpers; creating custom TagHelpers; and how TagHelpers can be used in conjunction with third-party controls. Let's pull together all we've learned into a single application. The C1Finance app is designed to display a dashboard of financial information so users can immediately assess stock market health and changes. The single-view page includes a FlexGrid, AutoComplete, and FlexChart control.

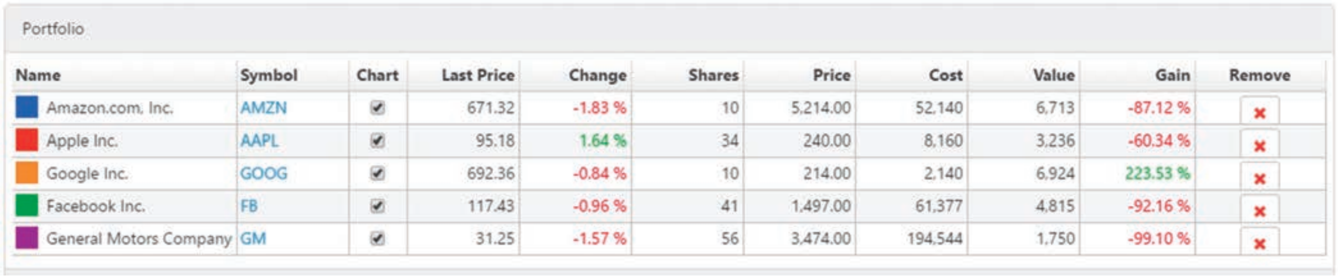
The app was originally created using ASP.NET 4.0 and used the C1Studio MVC HtmlHelpers. (Link to download available in the references.) We'll look at each control and convert to TagHelpers in an ASP.NET Core app.

5.1. FlexGrid

The FlexGrid displays company and respective symbols, price changes and gain.

It includes:

- Editable checkboxes
- Conditional formatting to show percentage changes in stock prices
- Embedded image to represent a delete button
- Custom formatting to color and text for company name
- Embedded hyperlink in Symbol names that navigate to company stock information page on Yahoo

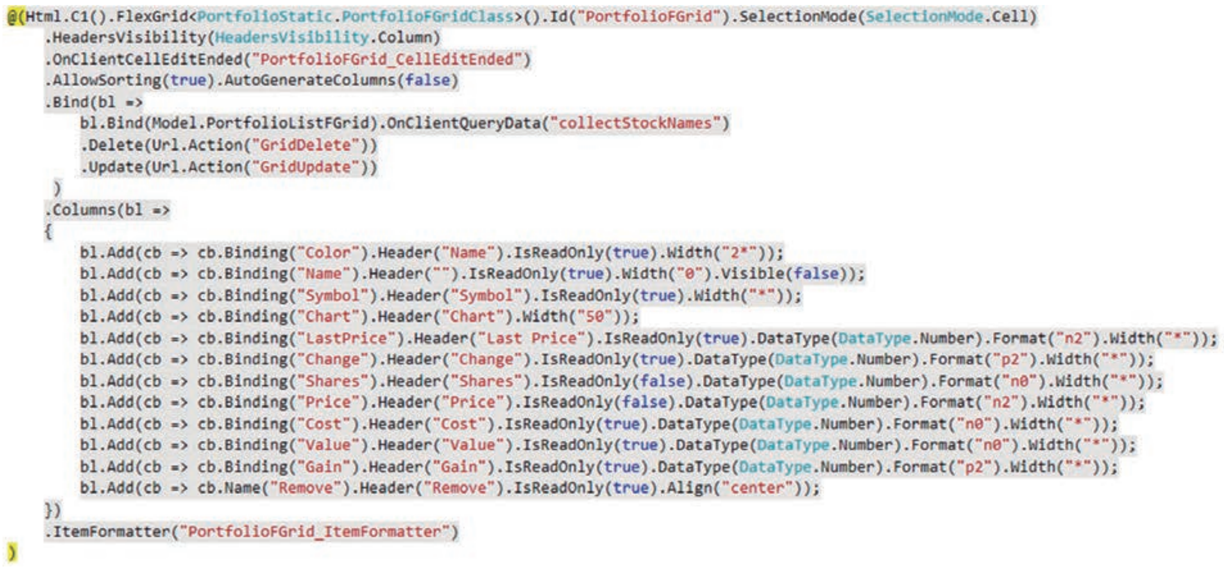


Portfolio

Name	Symbol	Chart	Last Price	Change	Shares	Price	Cost	Value	Gain	Remove
🟦 Amazon.com, Inc.	AMZN	✓	671.32	-1.83 %	10	5,214.00	52,140	6,713	-87.12 %	✖
🟥 Apple Inc.	AAPL	✓	95.18	1.64 %	34	240.00	8,160	3,236	-60.34 %	✖
🟧 Google Inc.	GOOG	✓	692.36	-0.84 %	10	214.00	2,140	6,924	223.53 %	✖
🟩 Facebook Inc.	FB	✓	117.43	-0.96 %	41	1,497.00	61,377	4,815	-92.16 %	✖
🟪 General Motors Company	GM	✓	31.25	-1.57 %	56	3,474.00	194,544	1,750	-99.10 %	✖

Figure 9: FlexGrid with editable fields, conditional and custom formatting, and embedded images

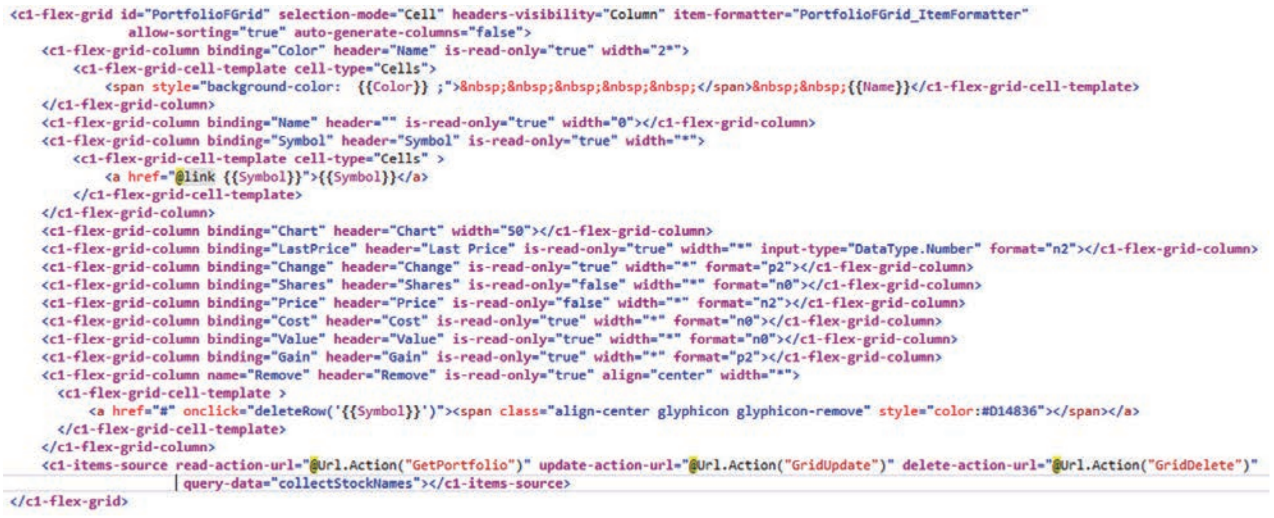
Here's the HtmlHelper code:



```
@(Html.C1().FlexGrid<PortfolioStatic.PortfolioFGridClass>().Id("PortfolioFGrid").SelectionMode(SelectionMode.Cell)
    .HeadersVisibility(HeadersVisibility.Column)
    .OnClientCellEditEnded("PortfolioFGrid_CellEditEnded")
    .AllowSorting(true).AutoGenerateColumns(false)
    .Bind(bl =>
        bl.Bind(Model.PortfolioListFGrid).OnClientQueryData("collectStockNames")
        .Delete(Url.Action("GridDelete"))
        .Update(Url.Action("GridUpdate"))
    )
    .Columns(bl =>
    {
        bl.Add(cb => cb.Binding("Color").Header("Name").IsReadOnly(true).Width("2*"));
        bl.Add(cb => cb.Binding("Name").Header("").IsReadOnly(true).Width("0").Visible(false));
        bl.Add(cb => cb.Binding("Symbol").Header("Symbol").IsReadOnly(true).Width("*"));
        bl.Add(cb => cb.Binding("Chart").Header("Chart").Width("50"));
        bl.Add(cb => cb.Binding("LastPrice").Header("Last Price").IsReadOnly(true).DataType(DataType.Number).Format("n2").Width("*"));
        bl.Add(cb => cb.Binding("Change").Header("Change").IsReadOnly(true).DataType(DataType.Number).Format("p2").Width("*"));
        bl.Add(cb => cb.Binding("Shares").Header("Shares").IsReadOnly(false).DataType(DataType.Number).Format("n0").Width("*"));
        bl.Add(cb => cb.Binding("Price").Header("Price").IsReadOnly(false).DataType(DataType.Number).Format("n2").Width("*"));
        bl.Add(cb => cb.Binding("Cost").Header("Cost").IsReadOnly(true).DataType(DataType.Number).Format("n0").Width("*"));
        bl.Add(cb => cb.Binding("Value").Header("Value").IsReadOnly(true).DataType(DataType.Number).Format("n0").Width("*"));
        bl.Add(cb => cb.Binding("Gain").Header("Gain").IsReadOnly(true).DataType(DataType.Number).Format("p2").Width("*"));
        bl.Add(cb => cb.Name("Remove").Header("Remove").IsReadOnly(true).Align("center"));
    })
    .ItemFormatter("PortfolioFGrid_ItemFormatter")
)
```

That piece of code makes sense to a developer, but for a designer, its meaning isn't so apparent.
Further, the Symbol and Remove columns involve additional formatting handled in the grids itemFormatter method. Ease in creating inline templates that handle complex formatting is great feature of TagHelpers. The TagHelper version includes the complex formatting definition of Name, Symbol and Remove columns within the template itself, rather than in the grid's itemFormatter JavaScript method.

As you can see, the TagHelper code is concise, easy to write, and designer-friendly:



```
<c1-flex-grid id="PortfolioFGrid" selection-mode="Cell" headers-visibility="Column" item-formatter="PortfolioFGrid_ItemFormatter"
              allow-sorting="true" auto-generate-columns="false">
    <c1-flex-grid-column binding="Color" header="Name" is-read-only="true" width="2*">
        <c1-flex-grid-cell-template cell-type="Cells">
            <span style="background-color:  {{Color}} ;">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;</span>&nbsp;&nbsp;{{Name}}</c1-flex-grid-cell-template>
    </c1-flex-grid-column>
    <c1-flex-grid-column binding="Name" header="" is-read-only="true" width="0"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Symbol" header="Symbol" is-read-only="true" width="*">
        <c1-flex-grid-cell-template cell-type="Cells" >
            <a href="@link {{Symbol}}">{{Symbol}}</a>
        </c1-flex-grid-cell-template>
    </c1-flex-grid-column>
    <c1-flex-grid-column binding="Chart" header="Chart" width="50"></c1-flex-grid-column>
    <c1-flex-grid-column binding="LastPrice" header="Last Price" is-read-only="true" width="*" input-type="DataType.Number" format="n2"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Change" header="Change" is-read-only="true" width="*" format="p2"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Shares" header="Shares" is-read-only="false" width="*" format="n0"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Price" header="Price" is-read-only="false" width="*" format="n2"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Cost" header="Cost" is-read-only="true" width="*" format="n0"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Value" header="Value" is-read-only="true" width="*" format="n0"></c1-flex-grid-column>
    <c1-flex-grid-column binding="Gain" header="Gain" is-read-only="true" width="*" format="p2"></c1-flex-grid-column>
    <c1-flex-grid-column name="Remove" header="Remove" is-read-only="true" align="center" width="*">
      <c1-flex-grid-cell-template >
          <a href="#" onclick="deleteRow('{{Symbol}}')"><span class="align-center glyphicon glyphicon-remove" style="color:#D14836"></span></a>
      </c1-flex-grid-cell-template>
    </c1-flex-grid-column>
    <c1-items-source read-action-url="@Url.Action("GetPortfolio")" update-action-url="@Url.Action("GridUpdate")" delete-action-url="@Url.Action("GridDelete")"
                     query-data="collectStockNames"></c1-items-source>
</c1-flex-grid>
```

5.2. AutoComplete

The AutoComplete feature allows users to easily add companies from a database to the current dashboard.

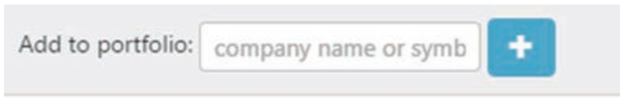
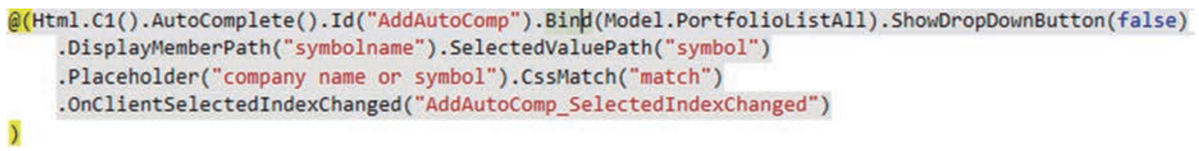


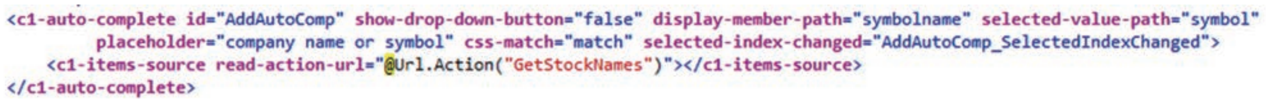
Figure 10: AutoComplete field

The AutoComplete binds to a collection of objects and enables searching the symbol, but it displays the actual Company name. It has a custom CSS attached to it through the CssMatch property, which highlights any parts of the content that match the search terms. Again, the HtmlHelper code functions well enough, but isn't so readable.



```
@(Html.C1().AutoComplete().Id("AddAutoComp").Bind(Model.PortfolioListAll).ShowDropDownButton(false)
    .DisplayMemberPath("symbolname").SelectedValuePath("symbol")
    .Placeholder("company name or symbol").CssMatch("match")
    .OnClientSelectedIndexChanged("AddAutoComp_SelectedIndexChanged")
)
```

Here's the new TagHelper version of the AutoComplete control. This AutoComplete binds to data returned through read-action-url.



```
<c1-auto-complete id="AddAutoComp" show-drop-down-button="false" display-member-path="symbolname" selected-value-path="symbol"
        placeholder="company name or symbol" css-match="match" selected-index-changed="AddAutoComp_SelectedIndexChanged">
    <c1-items-source read-action-url="@Url.Action("GetStockNames")"></c1-items-source>
</c1-auto-complete>
```

The FlexChart visualizes the rise and fall of stock prices over a given time span.

We'll need to customize:

- Type of chart
- Name of the chart
- Formatted X-Axis
- Selectable series

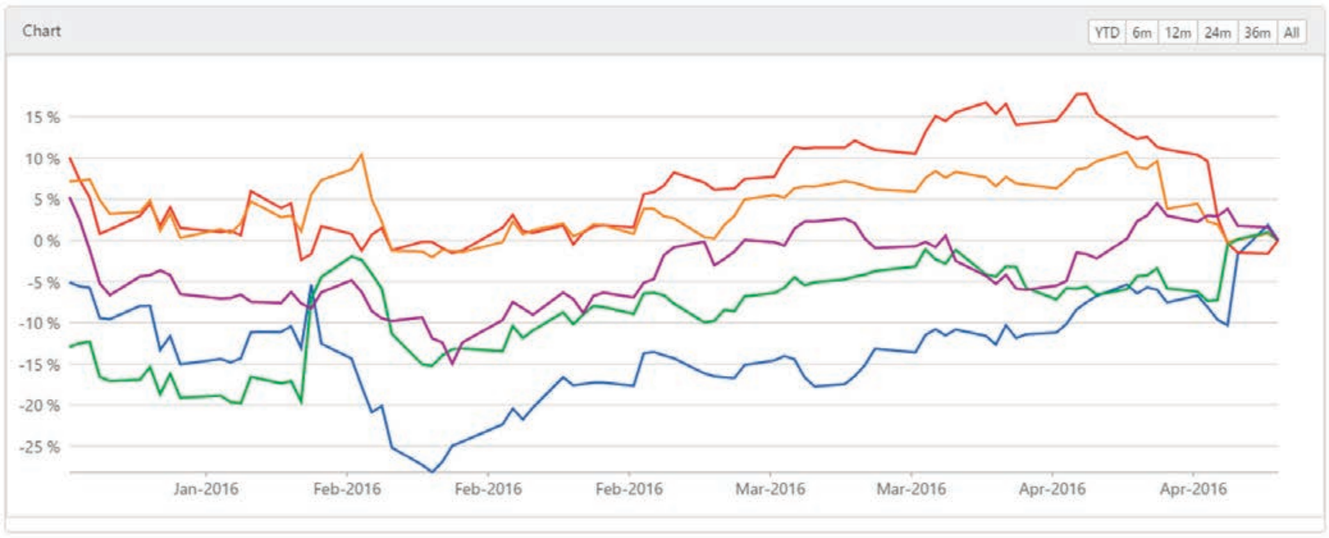


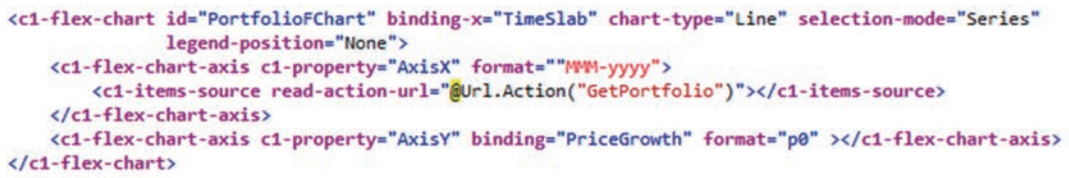
Figure 11: FlexChart with formatted X-Axis and selectable series

The HtmlHelper code below sets the above features of FlexChart. A common method of setting values for complex properties is through lambda expressions in HtmlHelpers. The Axis is a complex property in FlexChart, but you can see how easy it is to set the same in TagHelpers.

@

```
@(Html.C1().FlexChart().Id("PortfolioFChart")
    .BindingX("TimeSlab").ChartType(C1.Web.Mvc.Chart.ChartType.Line)
    .AxisX(x => x.Bind(Model.PortfolioListFGrid).Format("MMM-yyyy"))
    .AxisY(y => y.Format("p0").Binding("PriceGrowth")).Legend(C1.Web.Mvc.Chart.Position.None)
    .SelectionMode(C1.Web.Mvc.Chart.SelectionMode.Series)
  )
```

Now we'll convert to TagHelpers. Since complex properties can be child TagHelpers, it's easy to set these properties with TagHelpers. The chart could also bind to a Model, but this example shows how to get data from the action using read action URL. You get automatic IntelliSense for the enum properties of chart, and each property is self-explanatory.



```
<c1-flex-chart id="PortfolioFChart" binding-x="TimeSlab" chart-type="Line" selection-mode="Series"
               legend-position="None">
    <c1-flex-chart-axis c1-property="AxisX" format=""MMM-yyyy">
        <c1-items-source read-action-url="@Url.Action("GetPortfolio")"></c1-items-source>
    </c1-flex-chart-axis>
    <c1-flex-chart-axis c1-property="AxisY" binding="PriceGrowth" format="p0" ></c1-flex-chart-axis>
</c1-flex-chart>
```

6. Conclusion

TagHelpers are a helpful feature of ASP.NET Core MVC that solve HtmlHelper's issues while simultaneously providing new, powerful MVC features. In this paper, we looked at out-of-box TagHelpers and how Visual Studio makes it easy to use them. They're simple to understand, and we can even create our own TagHelpers.
Though they look like the Web Forms controls, they don't carry the baggage of control lifecycles that marred the Web Forms controls.

In future releases of ASP.NET Core, TagHelpers support may be extended to web pages, and it may be possible to author TagHelpers in Visual Basic as well.

7. References

- Download the white paper samples (zip):
  http://our.componentone.com/wp-content/uploads/2016/08/TagHelpersTutorial.zip
- ASP.NET Core TagHelpers documentation:
  https://docs.asp.net/en/latest/mvc/views/tag-helpers/intro.HTML
- jQueryUI AutoComplete Widget documentation:
  https://jqueryui.com/autocomplete/
- Bootstrap Carousel documentation:
  http://getbootstrap.com/javascript/#carousel
- Learn more about ComponentOne Studio MVC Edition:
  http://www.componentone.com/Studio/Platform/MVC
- MVC Edition ASP.NET Core Control Explorer:
  http://demos.componentone.com/ASPNET/5/MVCExplorer/
- ASP.NET MVC C1Finance demo:
  http://demos.componentone.com/aspnet/c1mvcfinance
- ASP.NET Core MVC C1Finance demo:
  http://demos.componentone.com/ASPNET/5/C1Finance/