100 XP

Create a user interface with Blazor components

7 minutes

Blazor components let you define webpages or portions of HTML that include dynamic content by using .NET code. In Blazor, you can formulate dynamic content by using C#, instead of using JavaScript.

Suppose you're working for a pizza delivery company to create a new modern website. You're starting with a welcome page that will become the landing page for most site users. You want to display special deals and popular pizzas on that page.

In this unit, you'll learn how to create components in Blazor and write code that renders dynamic content on those components.

Understand Blazor components

Blazor is a framework that developers can use to create a rich interactive user interface (UI) by writing C# code. With Blazor, you can use the same language for all your code, both server-side and client-side, and render it for display in many different browsers, including browsers on mobile devices.

Note

There are two hosting models for code in Blazor apps:

- **Blazor Server**: In this model, the app is executed on the web server within an ASP.NET Core app. UI updates, events, and JavaScript calls on the client side are sent through a **SignalR** connection between the client and the server. In this module, we'll discuss and code for this model.
- **Blazor WebAssembly**: In this model, the Blazor app, its dependencies, and the .NET runtime are downloaded and run on the browser.

In Blazor, you build the UI from self-contained portions of code called *components*. Each component can contain a mix of HTML and C# code. Components are written by using *Razor syntax*, in which code is marked with the `@code` directive. Other directives can be used to access variables, bind to values, and achieve other rendering tasks. When the app is compiled, the HTML and code are compiled into a component class. Components are written as files with a **.razor** extension.

Note

Razor syntax is used for embedding .NET code into webpages. You can use it in ASP.NET MVC applications, where files have a **.cshtml** extension. Razor syntax is used in Blazor to write components. These components have the **.razor** extension instead, and there's no strict separation between controllers and views.

Here's a simple example of a Blazor component:

razor

```razor
@page "/index"

<h1>Welcome to Blazing Pizza</h1>
```

```
<p>@welcomeMessage</p>

@code {
  private string welcomeMessage = "However you like your pizzas, we can deliver them blazing fast!";
}
```

In this example, the code sets the value of a string variable, named `welcomeMessage`. That variable is rendered within `<p>` tags in the HTML. We'll examine more complex examples later in this unit.

Create Blazor components

When you create a Blazor app by using the **blazorserver** template in the **dotnet** command-line interface (CLI), several components are included by default:

Bash

```bash
dotnet new blazorserver -o BlazingPizzaSite -f net6.0
```

The default components include the **Index.razor** home page and the **Counter.razor** demo component. Both of these components are placed in the **Pages** folder. You can either modify these views to fit your needs or delete them and replace them with new components.

To add a new component to an existing web app, use this command:

Bash

```bash
dotnet new razorcomponent -n PizzaBrowser -o Pages -f net6.0
```

- The `-n` option specifies the name of the component to add. This example adds a new file named **PizzaBrowser.razor**.
- The `-o` option specifies the folder that will contain the new component.
- The `-f` option forces the application to be built with the framework version .NET 6.

Important

The name of a Blazor component must begin with an uppercase character.

After you create the component, you can open it to be edited with Visual Studio Code:

Bash

```bash
code Pages/PizzaBrowser
```

Write code in a Blazor component

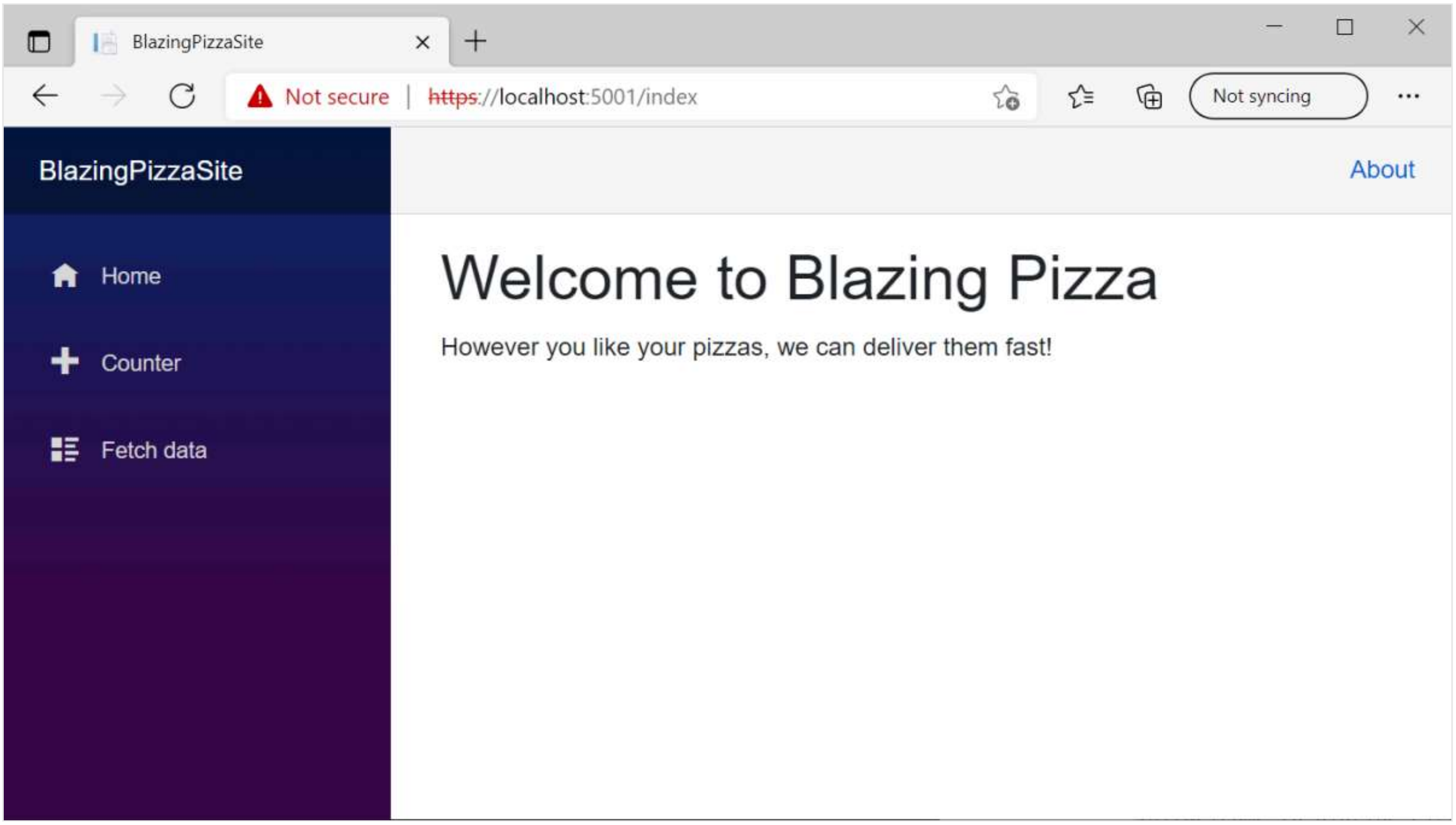
When you build a UI in Blazor, you mix static HTML and CSS markup with C# code, often in the same file. To differentiate these types of code, you use Razor syntax. Razor syntax includes directives, prefixed with the `@` symbol, that delimit C# code, routing parameters, bound data, imported classes, and other features.

Let's consider this example component again:

```razor
@page "/index"

<h1>Welcome to Blazing Pizza</h1>

<p>@welcomeMessage</p>

@code {
  private string welcomeMessage = "However you like your pizzas, we can deliver them fast!";
}
```

You can recognize the HTML markup with `<h1>` and `<p>` tags. This markup is the static framework of the page, into which your code will insert dynamic content. The Razor markup consists of:

- **The `@page` directive**: This directive provides a route template to Blazor. At runtime, Blazor locates a page to render by matching this template to the URL that the user requested. In this case, it might match a URL of the form `http://yourdomain.com/index`.
- **The `@code` directive**: This directive declares that the text in the following block is C# code. You can put as many code blocks as you need in a component. You can define component class members in these code blocks and set their values from calculation, data lookup operations, or other sources. In this case, the code defines a single component member called `welcomeMessage` and sets its string value.
- **Member access directives**: If you want to include the value of a member in your rendering logic, use the `@` symbol followed by a C# expression, such as the name of the member. In this case, the `@welcomeMessage` directive is used to render the value of the `welcomeMessage` member in the `<p>` tags.



Welcome to Blazing Pizza

However you like your pizzas, we can deliver them fast!