Template MS NET8 Blazor VS

GitHub Repository

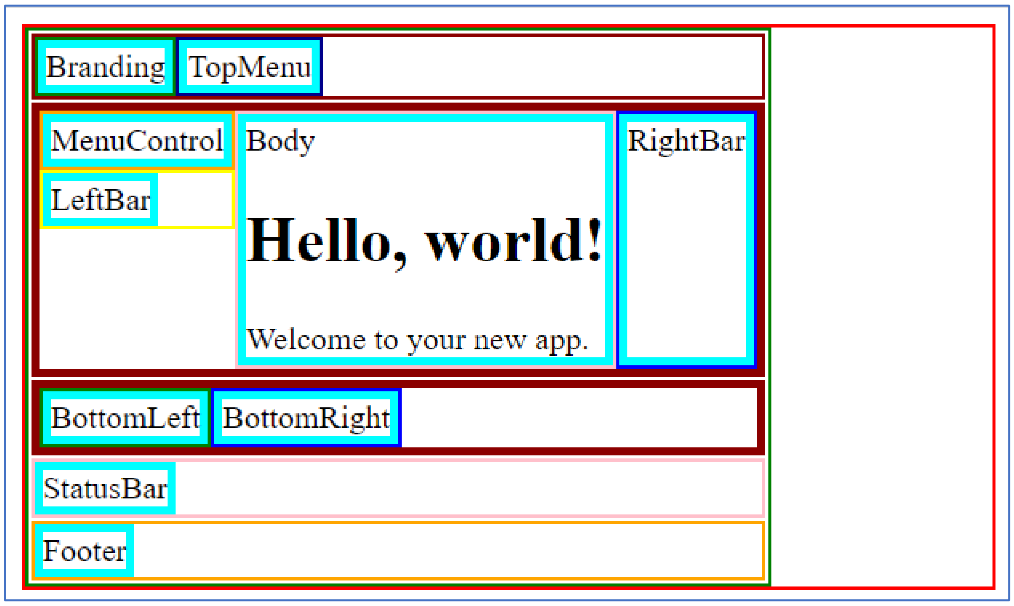
[Template_MS_NET8_Blazor_VS_V1/ at master · Developer-SLM-com/Template_MS_NET8_Blazor_VS_V1 (github.com)]

What this document includes

1. Using Microsoft-only controls, .NET 8, Blazor, and Visual Studio.
2. We will create a Visual Studio-style layout showing how to lay out several component scenarios.
3. We will create a data-driven mockup to manage the visibility of the components.
4. We will create a data-driven mockup for administrator-managed and user-selectable theme styles.
5. We will enhance MainLayout.razor to apply data-driven styles to the components.
6. We will describe in detail each step during the derivation of this layout so developers can enhance it as needed.

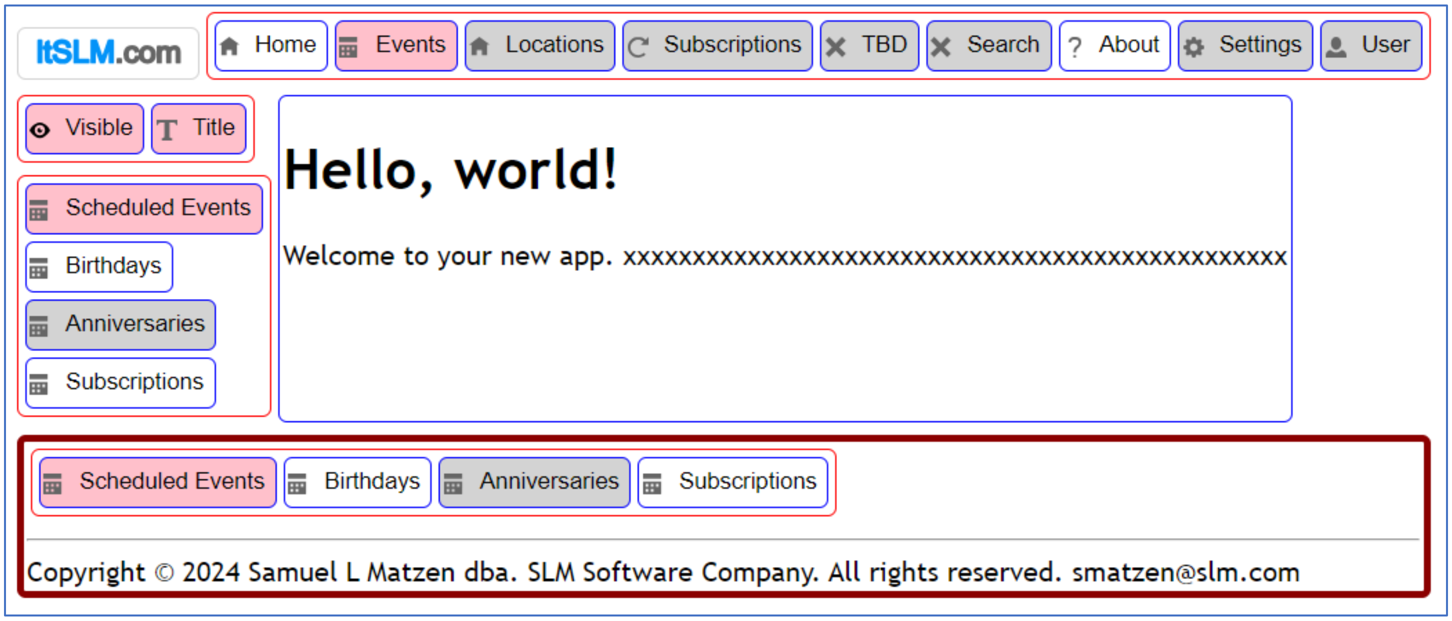
Final results from this tutorial

Each of the aqua borders is displayed by a separate component or the razor body:



Results after menus are added in the next tutorial

The following screen cap shows where we are going with this and is from a follow-up tutorial:



Hello, world!

Welcome to your new app. xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Copyright © 2024 Samuel L Matzen dba. SLM Software Company. All rights reserved. smatzen@slm.com

This looks kind of messy with all of the diagnostic borders turned on. We will clean it up later with the multiple Themes.

Disclaimer

I have been programming for quite a while and have spent the last 20 years away from the .NET environment. I am refreshing my skills in the Blazor/Razor arena and am just getting started. I don't know where all the Blazor buttons and dials are and how to use them properly. So, what I am writing most likely is not the "official" Microsoft way but should be helpful to others on the same journey.

Table of Contents

Revision Log

Version	Date	Environment	Author
1	4/11/2024	MS Controls - .NET 8.0 – Blazor – Visual Studio	Sam Matzen

Uses and Includes

- Microsoft, .NET 8.0., Blazor, Visual Studio.
- No third-party controls
- Template includes multiple navigation components, including branding, top menu, and side menu.
- Optional additional components to provide a Visual Studio-style environment.
- Data-Driven Component, Menu, and Appearance attributes (in Mockup).

Prerequisites

- Familiar with Visual Studio and how to create a new project.

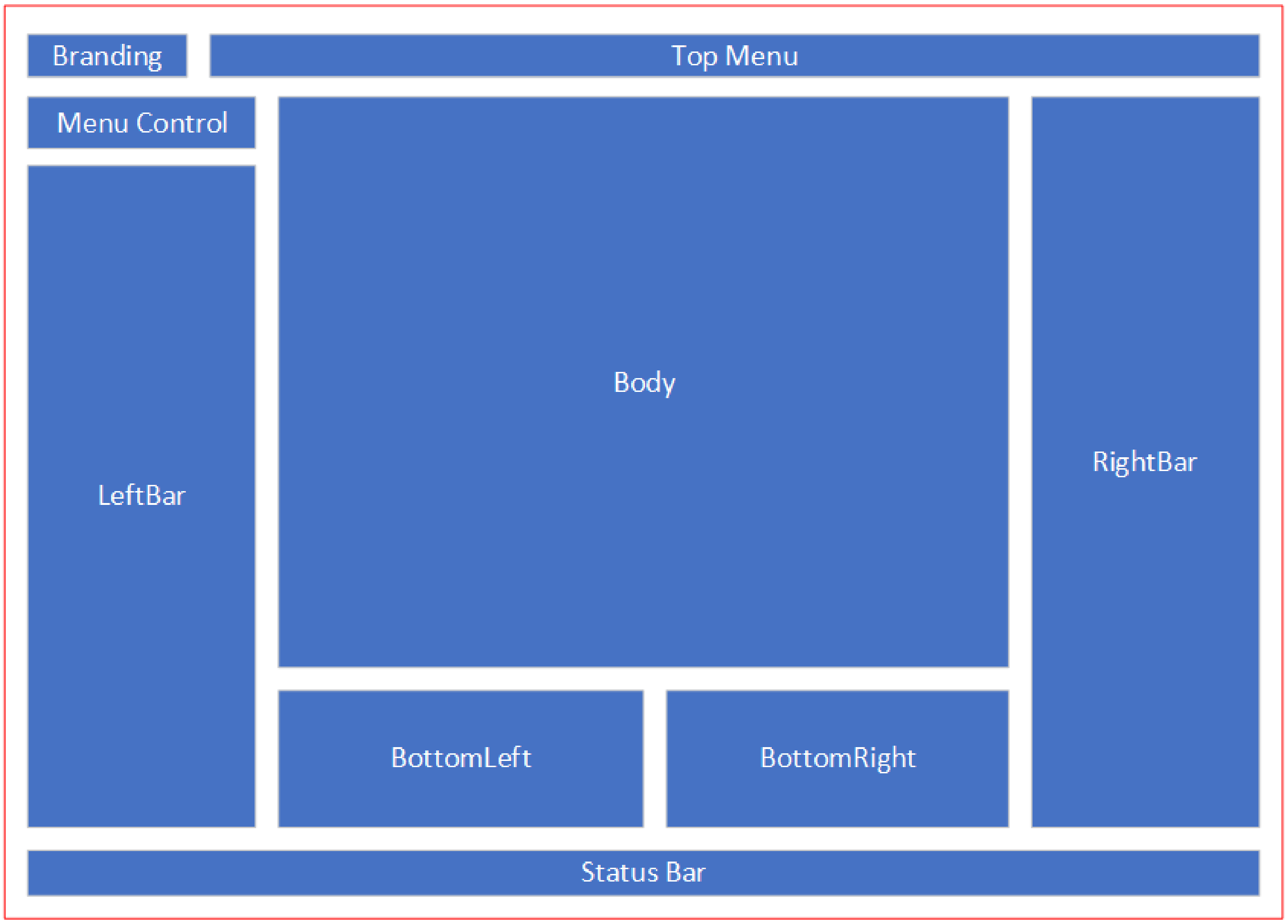
Patterns Implemented

- Data Models with Mockup
- Dependency Injection

Implemented Concepts

Components

We are creating a browser-based template with components to provide the Visual Studio-style interface. We know we will not get everything we want without much work, but we will be close and learn much about the MainLayout. A data model drives dynamic control of the components, and the data is in a mockup. The component naming is illustrated in the following diagram:



Menus

Menus have become quite sophisticated, but we are implementing a simplified structure with no third-party components for this example (Menus are implemented in a follow-up tutorial). The “Top Menu” consists of a list of selectable menu items, with the “Left Menu” showing sub-menu items associated with the selected “Top Menu” item. This simplified approach allows for themes that work well with large button accessibility and portable devices. The currently selected button is highlighted, and button styles change as the mouse moves over the bar.

“Menu Control” provides behavior to remove text from menu items and hide the menus if additional screen real estate is needed. This behavior is primarily targeted to the “Left Menu,” but for this example, it is implemented on both the “Top Menu” and “Left Menu.”

A data model drives dynamic control of the menus, and the data is in a mockup.

Styles

Styles are data-driven and dynamically inserted to allow real-time style configuration. A themes paradigm allows the user to select their desired theme, and the administrator can change and add themes as needed.

A data model drives dynamic control of the menus, and the data is in a mockup.

Walk through Steps Included

- Create initial Components.
- Initial MainLayout.
- Create and Populate Component Data Model.
- Apply Component Data Model to show/hide components.
- Create and Populate Data Models.

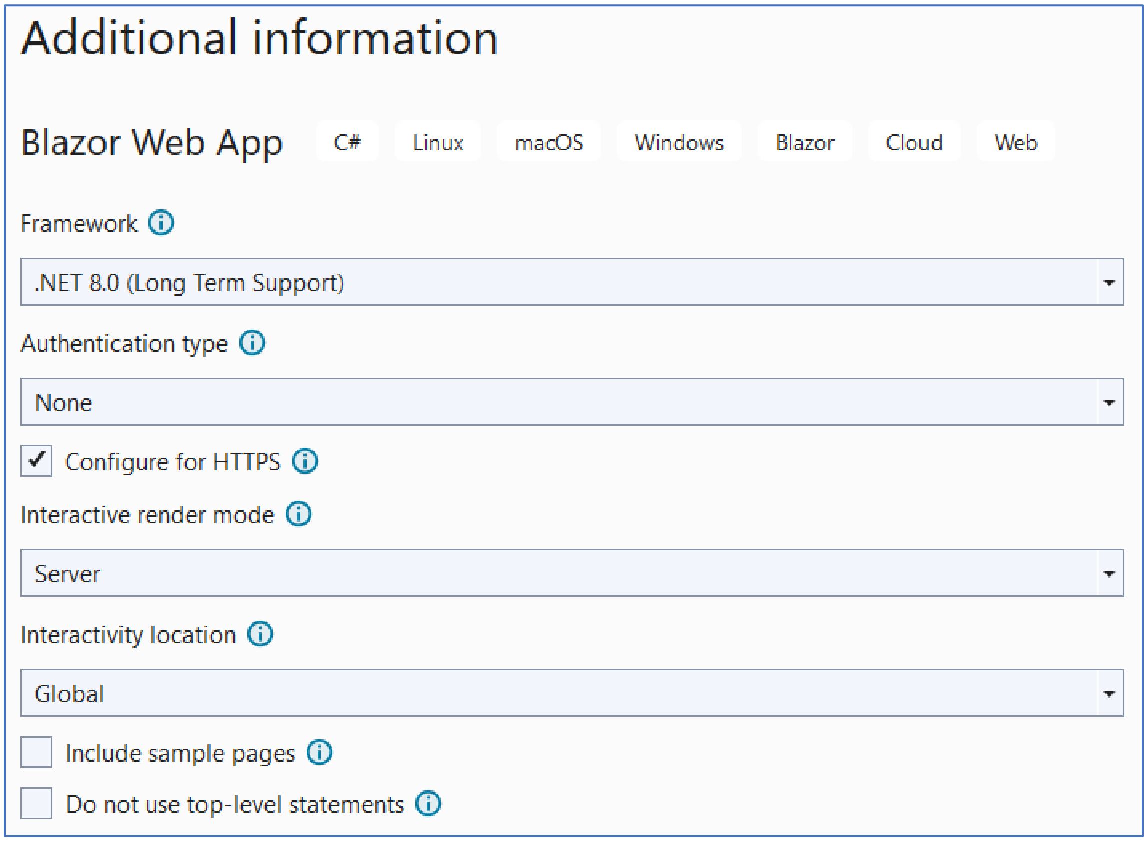
New Project

Launch Visual Studio 2022 17.9.3+

Select <Create a new project> and <Next> to view the “Create a new project” dialog.

Select <Blazor Web App> and <Next> to view the “Configure your new project” dialog.

Enter a Project Name and <Next> to view the Additional Information dialog.



Framework: .NET 8.0 (Long Term Support)

Authentication type: None

[✓] Configure for HTTPS

Interactive render mode: Server

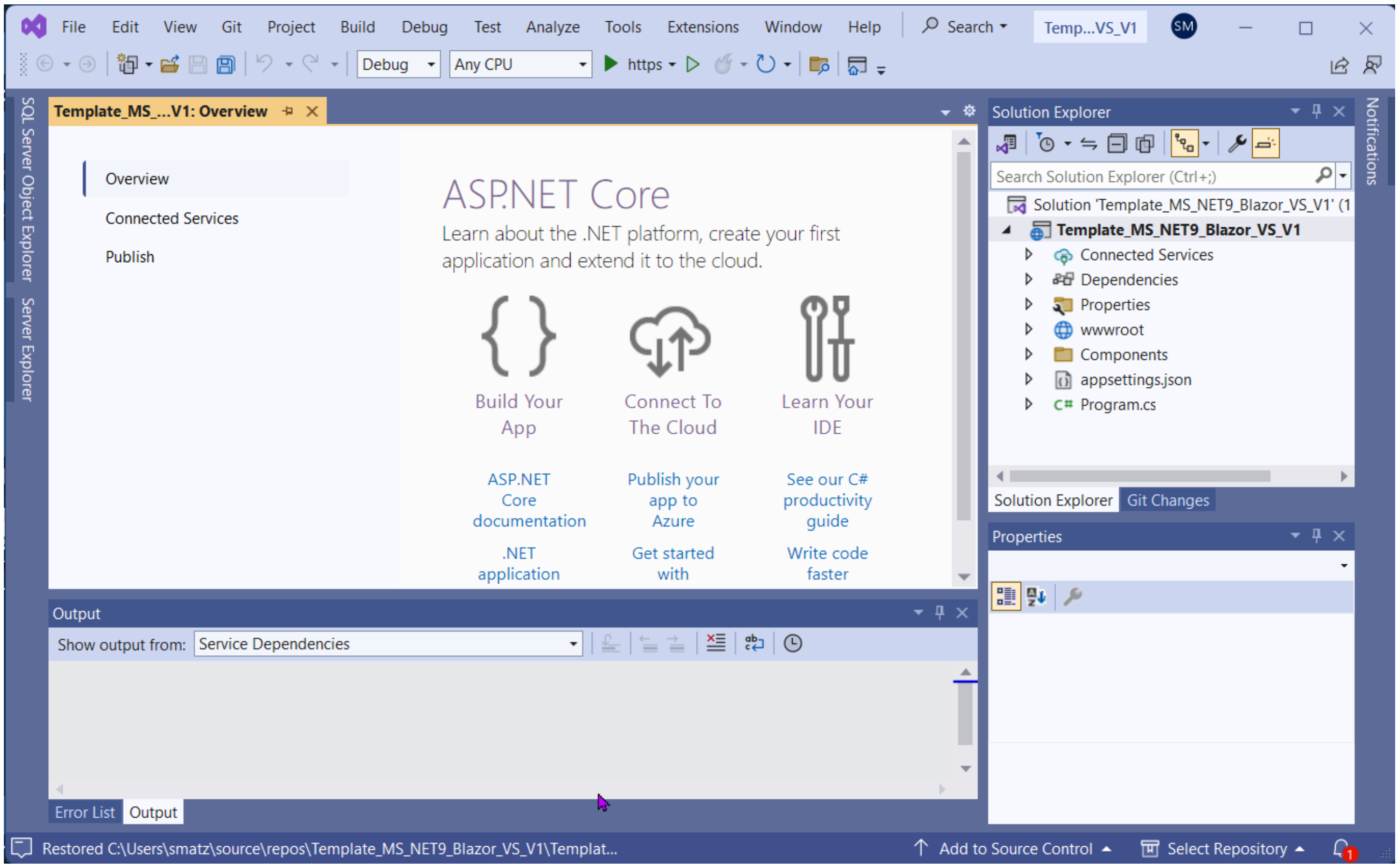
Interactivity location: Global

[ ] Include sample pages

[ ] Do not use top-level statements

Notice that even though you have the .NET 9 SDK installed, you can not select NET9 in the Framework drop-down. We will upgrade the project after it is created.

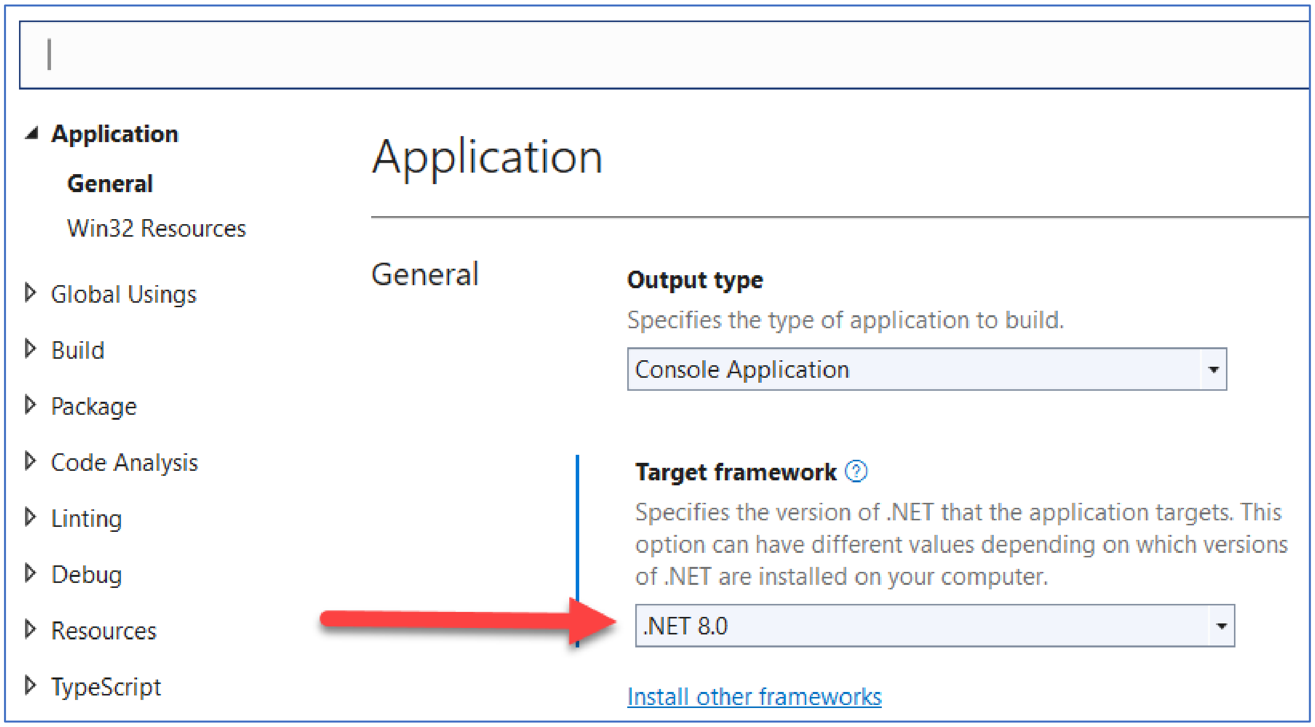
Configure “Additional information” and <Create>



You should see something like the above:

Verify "Target Framework"

Select <Project> -> <Properties> -> <Application> -> <General>



Output type
Specifies the type of application to build.
Console Application

Target framework
Specifies the version of .NET that the application targets. This option can have different values depending on which versions of .NET are installed on your computer.
.NET 8.0

Install other frameworks

As you can see, this project targets .NET 8.0. You may be able to walk through the steps below, but you may need to adjust for later versions.

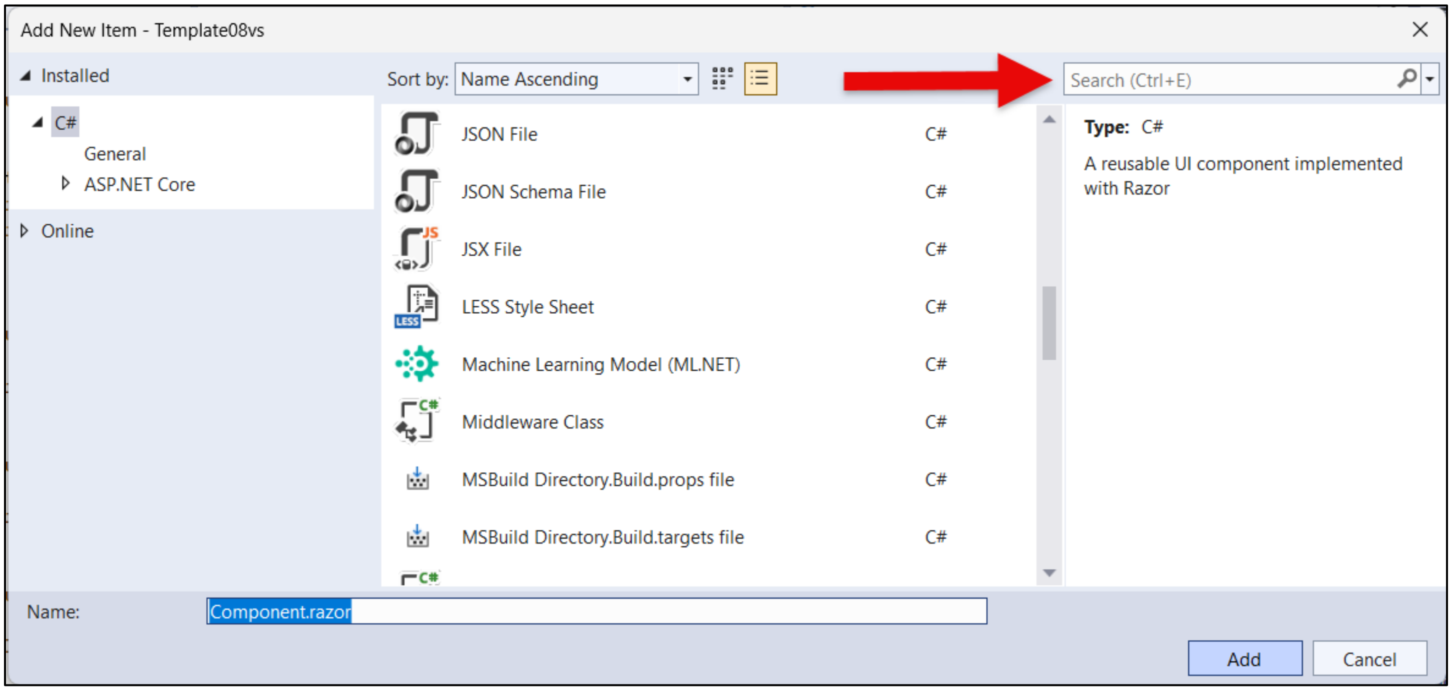
Create initial Components.

In the Components -> Layouts folder, create the following components:

- BottomLeft
- BottomRight
- Branding
- Footer
- LeftBar
- MenuControl
- RightBar
- StatusBar
- TopMenu

Creating a new Razor Component

<Right-click> on Components -> Layout and select <Add><Razor Component>



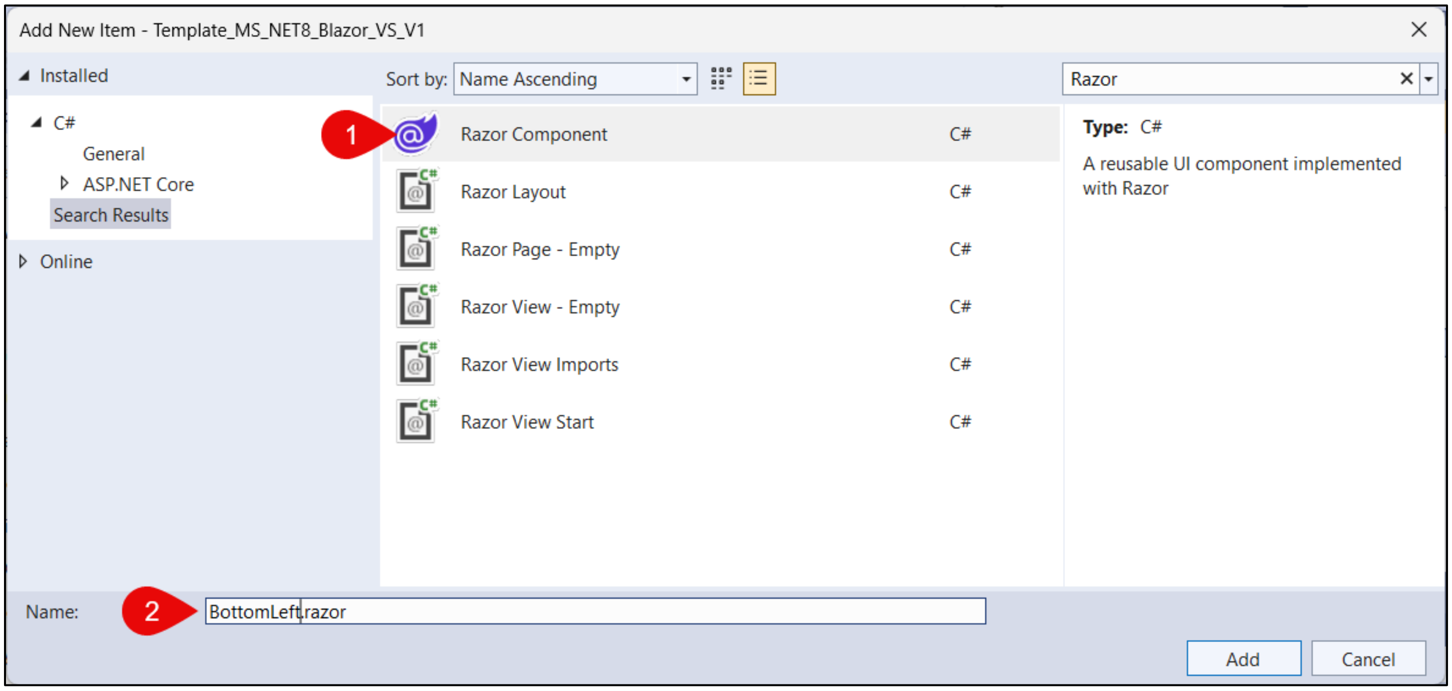
JSON File	C#
JSON Schema File	C#
JSX File	C#
LESS Style Sheet	C#
Machine Learning Model (ML.NET)	C#
Middleware Class	C#
MSBuild Directory.Build.props file	C#
MSBuild Directory.Build.targets file	C#

Type: C#

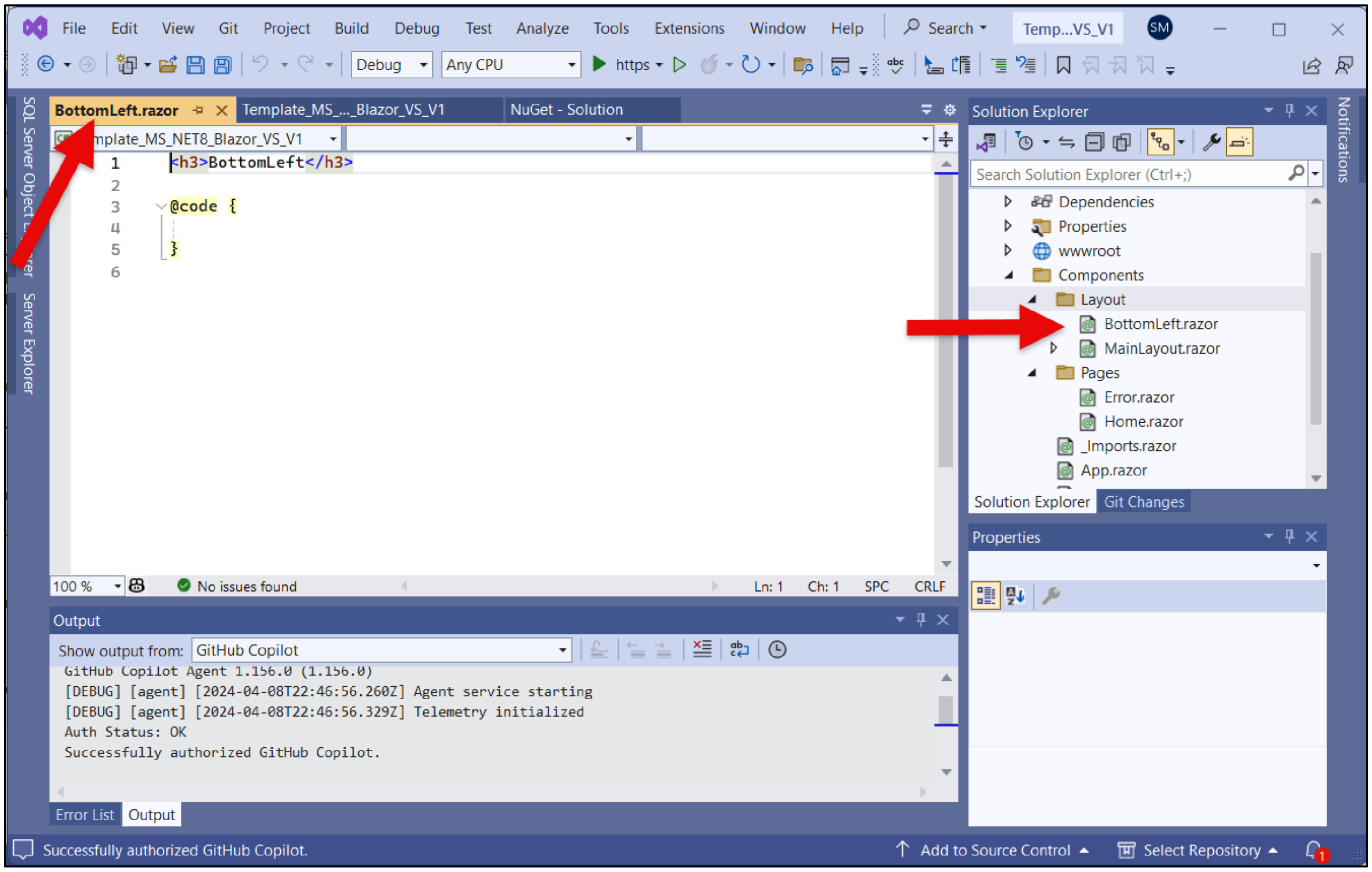
A reusable UI component implemented with Razor

Name: Component.razor

Type “Razor” in the “Search” box:



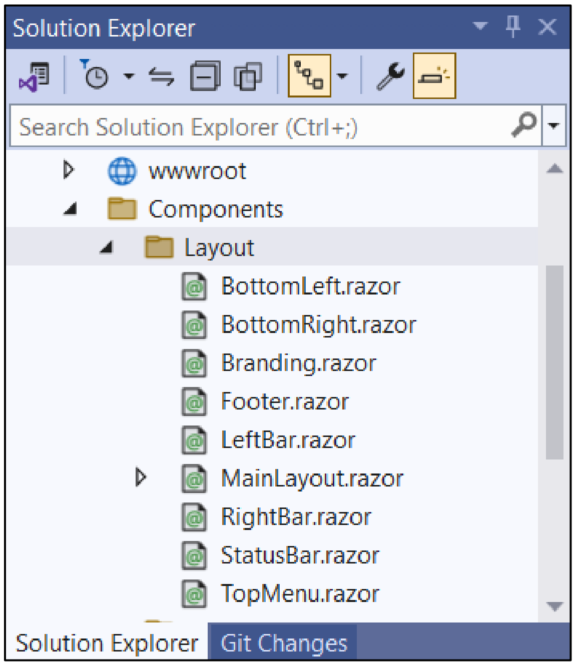
Select <Razor Component>, enter the Component Name, and select <Add>.


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search Temp...VS_V1
Debug Any CPU https
BottomLeft.razor Template_MS_..._Blazor_VS_V1 NuGet - Solution
Solution Explorer
Search Solution Explorer (Ctrl+;)
Dependencies
Properties
wwwroot
Components
Layout
BottomLeft.razor
MainLayout.razor
Pages
Error.razor
Home.razor
_Imports.razor
App.razor
Solution Explorer Git Changes
Properties
100 % No issues found Ln: 1 Ch: 1 SPC CRLF
Output
Show output from: GitHub Copilot
Error List Output
Successfully authorized GitHub Copilot.
Add to Source Control Select Repository

```
<h3>BottomLeft</h3>

@code {

}
```

```
GitHub Copilot Agent 1.156.0 (1.156.0)
[DEBUG] [agent] [2024-04-08T22:46:56.260Z] Agent service starting
[DEBUG] [agent] [2024-04-08T22:46:56.329Z] Telemetry initialized
Auth Status: OK
Successfully authorized GitHub Copilot.
```

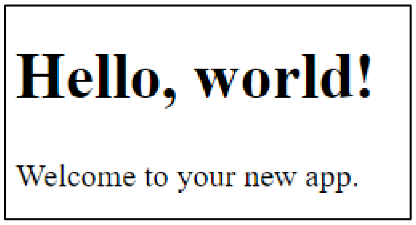
Perform the same steps to add the other components:



You should now see the components in the Solution Explorer.

Initial MainLayout

Run the project and review the output:



Hello, world!

Welcome to your new app.

Now let's look at the MainLayout.razor component:

```
@inherits LayoutComponentBase

@Body

<div id="blazor-error-ui">
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">🗙</a>
</div>
```

This is basic. We will not change the “blazer-error-ui” div; it will just be moved down as we add our code, and for now, we will illustrate the @inherits declaration optionally.

Now, this is where you will need to trust me on some things related to getting Blazor to arrange things. In the first pass, we will use a Table to arrange the components into rows, and later, we may try to use some flex magic.

Add a “container”

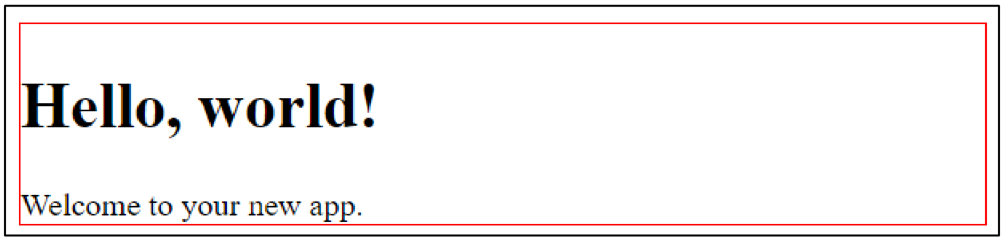
So we can easily show the display area we are working with, we are going to start with a div-class called “container” to define our display area:

```
<div class="container">

    @Body

</div>
```

And, we will add the “container” style to MainLayout.razor.css:

```
.container {
    padding-right: 14px;
    margin-left: 0px;
    margin-top: 0px;
    padding-left: 0px;
    padding-top: 0px;
    font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans-serif
}
```

Now we can see the container we are working with:



Hello, world!

Welcome to your new app.

Add a "table" inside the container

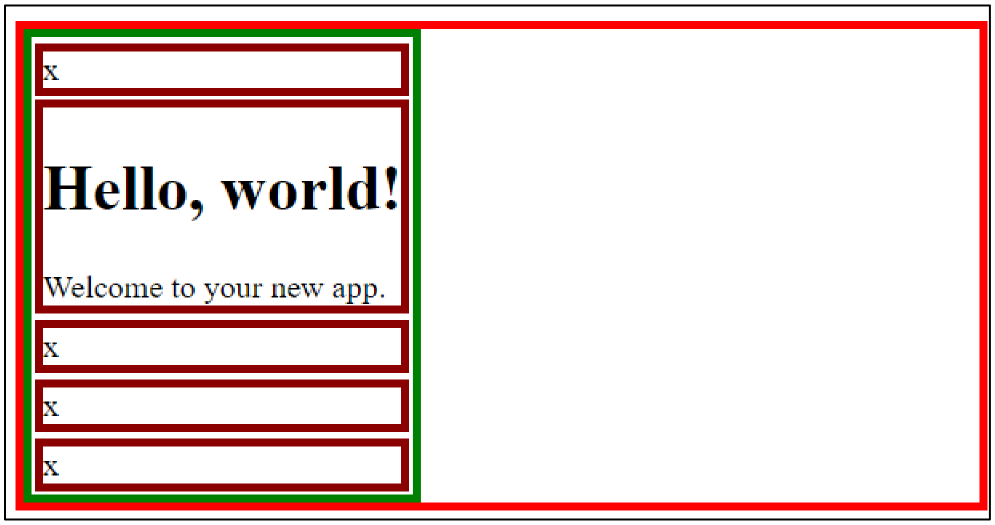
The object at this time is to see the individual areas. To clarify this, let's increase the border size to 4px, create a table, and create five rows to contain the components. It turns out the <tr> does recognize styles, so we add a <div> inside each <tr> to show the geography of each table row. We display an "x" in each row to view all the rows.

```
<div class="container">
    <table class="tableheader">
        <tr class="BrandingTopMenu"><div class="tablerowdiv">x</div></tr>
        <tr class="Body">
            <div class="tablerowdiv">
                @Body
            </div>
        </tr>
        <tr class="BottomLeftRight"><div class="tablerowdiv">x</div></tr>
        <tr class="StatusBar"><div class="tablerowdiv">x</div></tr>
        <tr class="Footer"><div class="tablerowdiv">x</div></tr>
    </table>
</div>
```

We also added three classes to the css to support the table headers, table rows, and table divisions:

```
.tableheader {
    margin: 0px;
    padding: 0px;
    border: solid 4px green;
}

.tablerowdiv {
    margin: 0px;
    padding: 0px;
    border: solid 4px darkred;
}
```

When we run the project we see:


x

Hello, world!

Welcome to your new app.

x

x

x

As you can see, the red border is the container, and the green border is the table. The dark red border is from the <div> in each <tr>. The reason for the extra white space around the <tr> elements is unknown, and I was unable to get that white space to go away. I suspect it is a legacy issue with table rows.

Add the Components

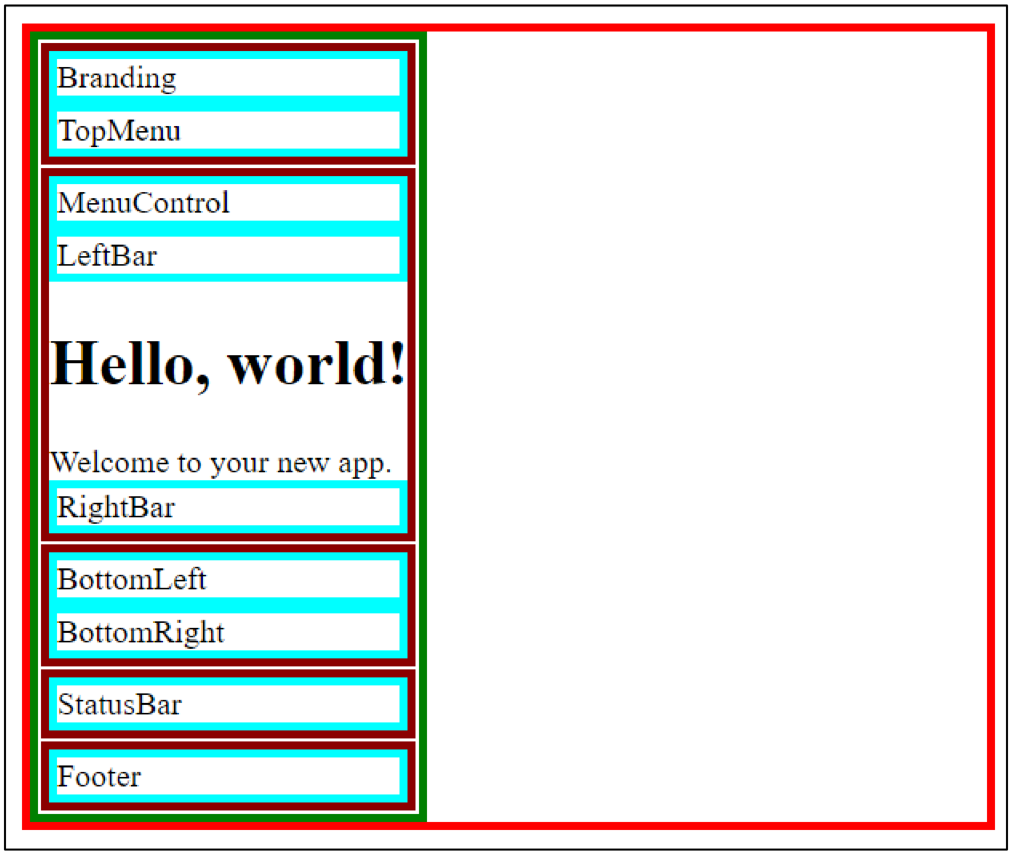
To identify the display area taken by each component, let's enhance each component with a <div>, and remove the <h3> tag around the component name as follows:

```
<div style="margin: 0px; padding: 0px; border: solid 4px aqua;">
    Branding
</div>
```

After adding the components into the appropriate table row the MainLayout.razor contains:

```
<div class="container">
    <table class="tableheader">
        <tr class="BrandingTopMenu"><div class="tablerowdiv"><Branding /><TopMenu /></div></tr>
        <tr class="Body">
            <div class="tablerowdiv">
                <MenuControl /><LeftBar />@Body<RightBar />
            </div>
        </tr>
        <tr class="BottomLeftRight"><div class="tablerowdiv"><BottomLeft /><BottomRight /></div></tr>
        <tr class="StatusBar"><div class="tablerowdiv"><StatusBar /></div></tr>
        <tr class="Footer"><div class="tablerowdiv"><Footer /></div></tr>
    </table>
</div>
```

And when we run the project:


Branding
TopMenu
MenuControl
LeftBar
Hello, world!
Welcome to your new app.
RightBar
BottomLeft
BottomRight
StatusBar
Footer

At this stage, we have all of the components displaying, but when you review the layout, the positioning of the components in the table rows is incorrect.

Arrange the Components in the Rows

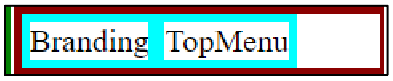
Another common layout style is the “wrapper”:

```
.wrapper {
    display: flex;
    flex-flow: row wrap;
    font-weight: normal;
    text-align: left;
    align-content: stretch;
}
```

Branding and TopMenu

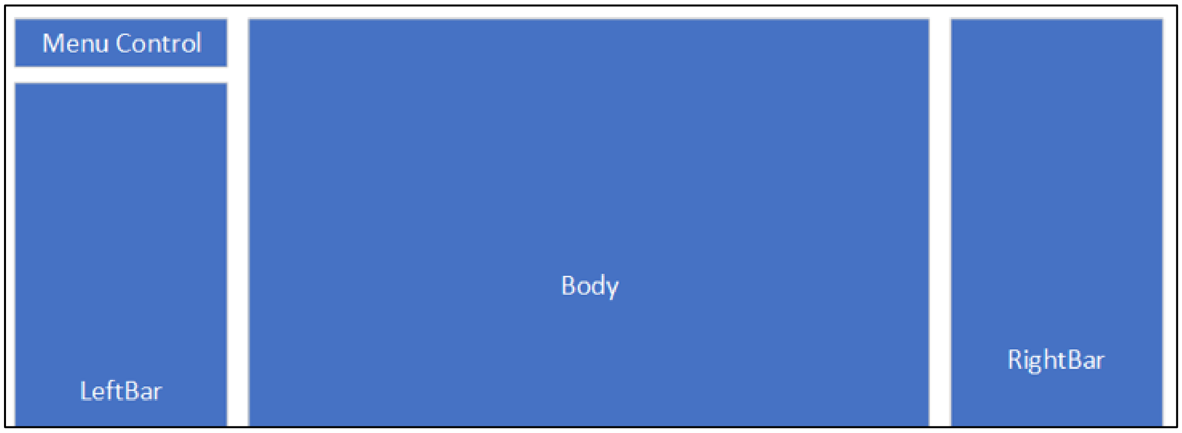
Branding and TopMenu should be horizontally next to each other. We accomplish this by adding the wrapper class to the “Top Menu” row:

```
        <tr class="/BrandingTopMenu">
            <div class="tablerowdiv wrapper"><Branding /><TopMenu /></div>
        </tr>
```


Branding TopMenu

MenuControl, LeftBar, @body, and RightBar

This takes a little more structure to get what we are looking for:



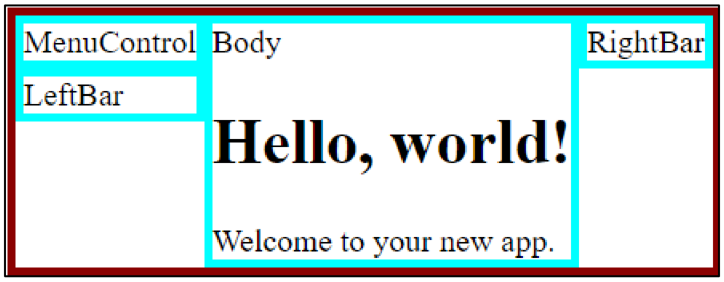
We add the “wrapper” class.

We add a <div> for the Menu Control and Left Bar components so they can stack.

We add another <div> for @Body and Right Bar with style="display:inline-flex; float:right;" to get the body and Right Bar to set beside each other.

Then we set up separate <div> for @Body and the Right Bar. We don’t need separate <div> here at this time, but we will use them later.

```
        <tr class="Body">
            <div class="tablerowdiv wrapper">
                <div><MenuControl /><LeftBar /></div>
                <div style="display:inline-flex; float:right;">
                    <div>@Body</div>
                    <div><RightBar /></div>
                </div>
            </div>
        </tr>
```



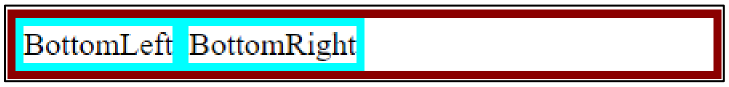
Hello, world!

Welcome to your new app.

BottomLeft and BottomRight

Here we add the “wrapper” styles the same as we did with Branding and Top Menu:

```
        <tr class="BottomLeftRight">
            <div class="tablerowdiv wrapper"><BottomLeft /><BottomRight /></div>
        </tr>
```



StatusBar

Nothing is needed here currently.

Footer

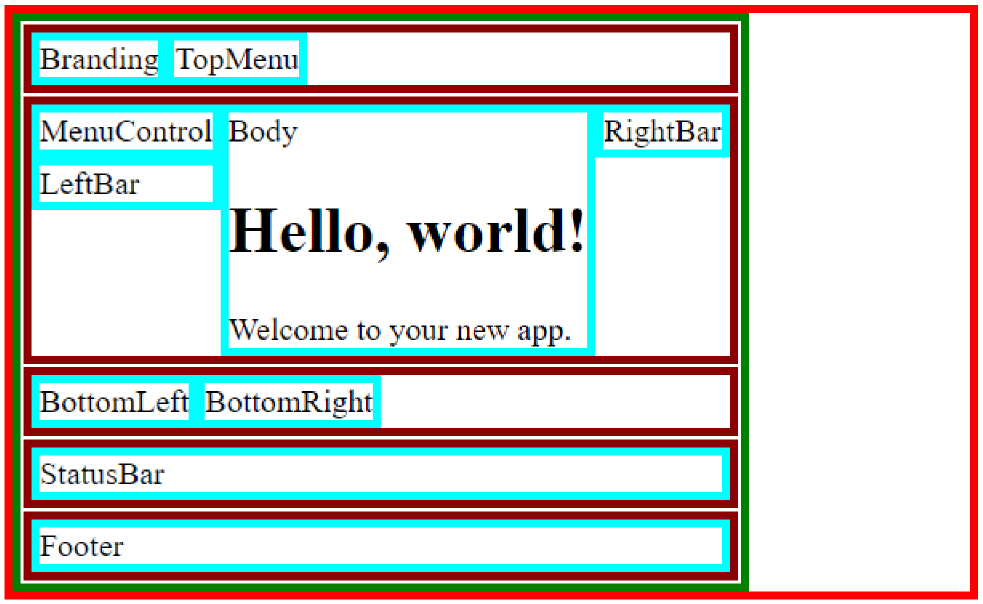
Nothing is needed here currently.

Final Arrangement

Final MainLayout component and body arrangement container division:

```
<div class="container">
    <table class="tableheader">

        <tr class="BrandingTopMenu">
            <div class="tablerowdiv wrapper"><Branding /><TopMenu /></div>
        </tr>

        <tr class="Body">
            <div class="tablerowdiv wrapper">
                <div><MenuControl /><LeftBar /></div>
                <div style="display:inline-flex; float:right;">
                    <div>@Body</div>
                    <div><RightBar /></div>
                </div>
            </div>
        </tr>

        <tr class="BottomLeftRight">
            <div class="tablerowdiv wrapper"><BottomLeft /><BottomRight /></div>
        </tr>

        <tr class="StatusBar"><div class="tablerowdiv"><StatusBar /></div></tr>

        <tr class="Footer"><div class="tablerowdiv"><Footer /></div></tr>

    </table>
</div>
```

The components and body are arranged as follows:

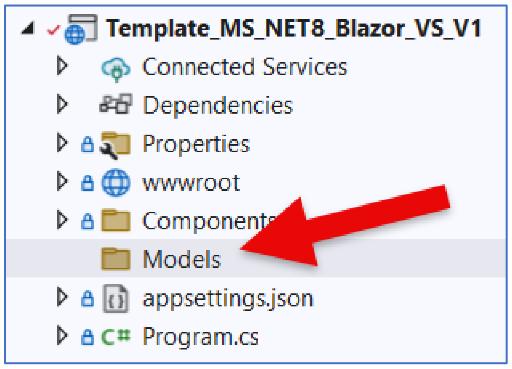


Data Models

For this walk-through, the data model is implemented in models with “mockup” data. We need data to control the components, and when we start building out the application, we will be able to enhance the component data model to help with the application. We also need data to define and manage the menus and the styles to provide administrator-managed and user-selected themes.

Create the “Models” directory to the project

<Right-Click> the project node and select <Add> -> <New folder> and change the name to “Models”.

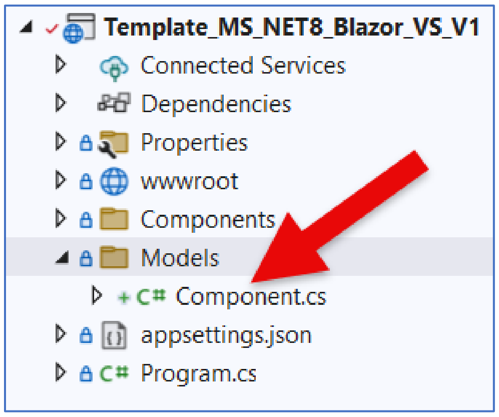


Component Data Model

Few applications will need all of the components in this template, and in some instances, they need to be turned on and off dynamically. We use a data model to support this behavior:

Create the Component Class

<Right-Click> the "Models" node and select <Add> -> <Class>, enter name "Component", and select <Add>.



The default class code includes:

```
namespace Template_MS_NET8_Blazor_VS_V1.Models
{
    public class Component
    {
    }
}
```

Define model Attributes

Let's add some attributes to the "Component" class:

```
    public class Component
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int ComponentId { get; set; }
        public string? Name { get; set; }
        public string? Description { get; set; }
        public bool IsVisible { get; set; }
    }
```

IsVisible supports an option not to display the component.

Define the Components class and mockup data

First, we create a collection of "Component" records (entities, rows, records, class instances, items). The AI seems to like records or items, so I will try to call them "records" in this document.

```
        public Collection<Component> ComponentsCollection { get; set; } = new Collection<Component>();
```

Second, we compose a class constructor to load the mock data into the collection when the class is created. This automatically happens when referenced in Blazor, so we don't need to create a separate class; all components can share the same instance. We make this class available to the component directly with "Dependency Injection" described later.

```
        public Components()                                 // constructor
        {
            MockData(ComponentsCollection);
        }
```

Third, we populate the mock data rows with all components visible and enabled.

```
        public void MockData(Collection<Component> c)
        {
            c.Add(new Component { Name = "BottomLeft", Description = "BottomLeft", IsVisible = true });
            c.Add(new Component { Name = "BottomRight", Description = "BottomRight", IsVisible = true });
            c.Add(new Component { Name = "Branding", Description = "Branding", IsVisible = true });
            c.Add(new Component { Name = "Footer", Description = "Footer", IsVisible = true });
            c.Add(new Component { Name = "LeftBar", Description = "LeftBar", IsVisible = true });
            c.Add(new Component { Name = "MenuControl", Description = "MenuControl", IsVisible = true });
            c.Add(new Component { Name = "RightBar", Description = "RightBar", IsVisible = true });
            c.Add(new Component { Name = "StatusBar", Description = "StatusBar", IsVisible = true });
            c.Add(new Component { Name = "TopMenu", Description = "TopMenu", IsVisible = true });
        }
```

Summary

Now, our Components class data section is complete and we have a container where we can store our components attributes, change them as needed, and transfer this mock data to a persisted storage at a later time.

```
    public class Components
    {
        public Collection<Component> ComponentsCollection { get; set; } = new Collection<Component>();

        public Components()                                 // constructor
        {
            MockData(ComponentsCollection);
        }

        public void MockData(Collection<Component> c)
        {
            c.Add(new Component { Name = "BottomLeft", Description = "BottomLeft", IsVisible = true });
            c.Add(new Component { Name = "BottomRight", Description = "BottomRight", IsVisible = true });
            c.Add(new Component { Name = "Branding", Description = "Branding", IsVisible = true });
            c.Add(new Component { Name = "Footer", Description = "Footer", IsVisible = true });
            c.Add(new Component { Name = "LeftBar", Description = "LeftBar", IsVisible = true });
            c.Add(new Component { Name = "MenuControl", Description = "MenuControl", IsVisible = true });
            c.Add(new Component { Name = "RightBar", Description = "RightBar", IsVisible = true });
            c.Add(new Component { Name = "StatusBar", Description = "StatusBar", IsVisible = true });
            c.Add(new Component { Name = "TopMenu", Description = "TopMenu", IsVisible = true });
        }
    }
```

Style Attribute Data Model

Building out the Style Attribute Data Model follows the same pattern as the build-out for the Component Data Model.

Create the StyleAttribute class

Follow the same steps described in the Component Data Model with the class "StyleAttribute." I am using "StyleAttribute" here instead of just "Style" to remove confusion.

The "StyleAttribute" class initially looks like:

```
    public class StyleAttribute
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int StyleAttributeId { get; set; }
        public string? Theme { get; set; }
        public string? Section { get; set; }
        public string? SubSection { get; set; }
        public string? Filter { get; set; } = "";
        public bool HighLight { get; set; } = false;
        public string? Key { get; set; }
        public string? Value { get; set; }
    }
```

As we discuss menu items in a future section, some class members are included for future use.

The first implementation of "StyleAttributes" includes configuring "Programs.cs" and "MainLayout.razor" for dependency injection of the StyleAttributes class. Follow the pattern set in the "Component Data Model" section.

We are going to change the border style on the "container" to "`solid 2px red`" by adding the following "StyleAttribute" record:

```
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "Container", Key = "border",
Value = "solid 2px red" });
```

You will see a "Theme" attribute: This will be used in the future to allow for multiple-theme support.

The "Section" and "SubSection" provide data classification. As you can see, the entry above is for the "MainLayout" section and "Container" subsection, meaning it is used by the "MainLayout" component and the "Container" division.

For many style specifications, we can make a call directly to the "GetSettings" method implemented in the "StyleAttributes" injected class:

```
public string? GetSettings(string theme, string section
    , string subsection, string filter = "", bool highlight = false)
{
    string? result = "";
    foreach (var item in StyleAttributesCollection.
                Where(i => (i.Filter == filter || i.Filter == "")
                && (i.Theme == theme)
                && (i.Section == section)
                && (i.SubSection == subsection)
                && (i.HighLight == false)))
    {
        result += item.Key + ":" + item.Value + "; ";
        if (highlight == true) // add highlight settings
        {
            foreach (var highlightitem in StyleAttributesCollection.
                Where(i => (i.Filter == filter || i.Filter == "")
                && (i.Theme == theme)
                && (i.Section == section)
                && (i.SubSection == subsection)
                && (i.HighLight == true)))
            {
                result += highlightitem.Key + ":" + highlightitem.Value + "; ";
            }
        }
    }
    return result;
}
```

This relatively complex method retrieves the data records based on theme, section, subsection, filter, and highlight. It accumulates the key and value attributes in a formatted HTML style and adds any highlight style elements.

Menu Data Model

TBD

Dependency Injection

Dependency Injection makes the public members of a class available within the component.

To make the Components data model class behavior available to other components:

1. Add this "using" to "Programs.cs":

```
using Template_MS_NET8_Blazor_VS_V1.Models;
```

2. Add this "AddSingleton" to the "builder" before "builder.Build();"

```
builder.Services.AddSingleton<Components
```

Implement Data-Driven Behavior

To implement dynamic UI behavior, we need to make calls into C# from within the HTML and use the data record values to provide the necessary attributes (typically HTML "Style" data) to modify the HTML behavior.

Component Behavior

The implementation allows for changing the "Visibility State" of each Component. To implement visibility, we set the "Display" style to "none" to remove the component and back to "flex" to show the component.

C# code call in HTML to implement visibility

This is our first implementation of C# code in HTML.

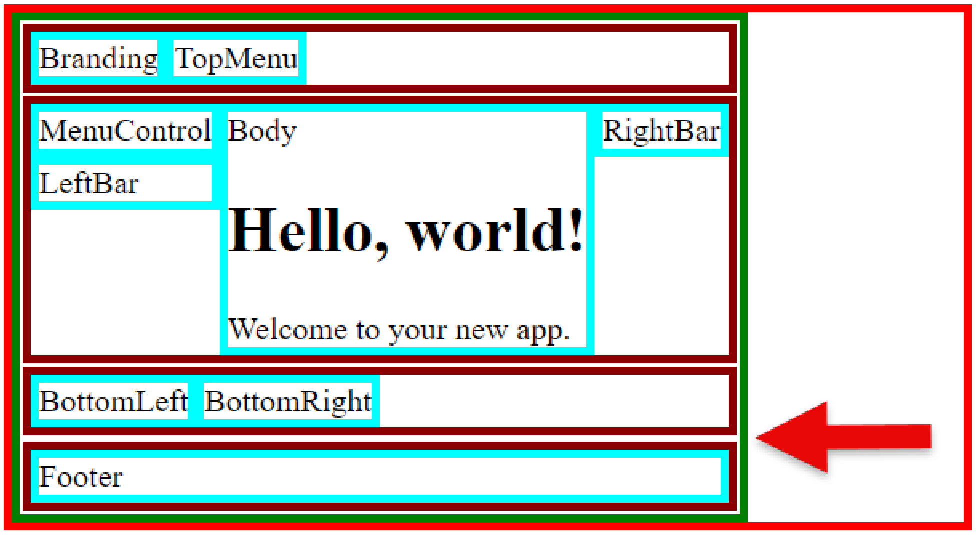
For this specific example, we are going to "hide" the StatusBar. Currently the "tablerowdiv" class is specified. We are going to add a C# generated "style" to add the needed "display" attribute. Currently the StatusBar row codes looks like:

```
        <tr class="StatusBar"><div class="tablerowdiv"><StatusBar /></div></tr>
```

We need to change it to the following to hide the StatusBar:

```
        <tr class="StatusBar"><div class="tablerowdiv" style="display:none"><StatusBar /></div></tr>
```

Make this change and run the project and notice the StatusBar component no longer displays:



Now we need to make the “style” dynamic by calling C#:

```
        <tr class="StatusBar"><div class="tablerowdiv" style="@ComponentStyle"><StatusBar /></div></tr>
```

And add a @code section to feed the “style”:

```
@code{

    private string ComponentStyle = "display:none;";

}
```

Run the project and observe the StatusBar component is missing.

Add a method to MainLayout to retrieve Component Style

First, let’s add the component name to the HTML:

```
<tr class="StatusBar"><div class="tablerowdiv" style="@ComponentStyle("StatusBar")"><StatusBar /></div></tr>
```

Second, enhance the ComponentStyle method to call a method (next step) to retrieve the dynamically created style text:

```
    private string ComponentStyle(string ComponentName)
    {
        return components.componentStyle(ComponentName);
    }
```

Dependency Injection for the Components class

Add the following directive to MainLayout to make the Components class directly accessible within MainLayout:

```
@inject Models.Components components
```

Add a method to the Components class to return the appropriate “style”:

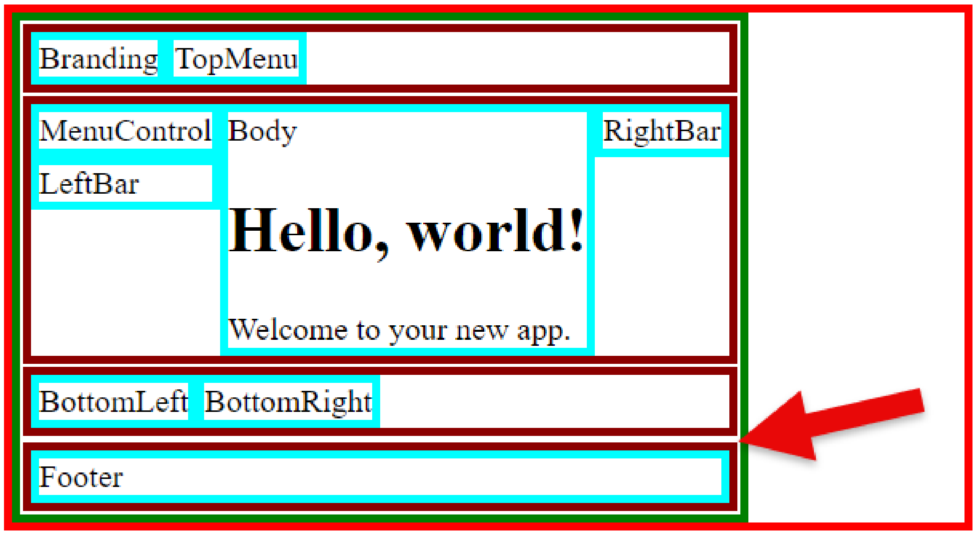
```
public string componentStyle(string ComponentName)
{
    int index = ComponentsCollection.Select((c, i) => new { c, i }).First(x => x.c.Name == ComponentName).i;
    Component c = ComponentsCollection[index];
    return c.IsVisible ? "display:flex" : "display:none";
}
```

Change data rows in the Component data model

To turn off the StatusBar, change the Component mockup IsVisible attribute data for the StatusBar record:

```
c.Add(new Component { Name = "StatusBar", Description = "StatusBar", IsVisible = false });
```

Run the project and observe if the StatusBar component is not displayed:



Propagate @ComponentStyle(call to other components in the HTML:

As you can see in the following code, we have added the @ComponentStyle(call to all of the MainLayout components. We have also added some additional <div> elements to provide the separation we need to implement a different style for each component.

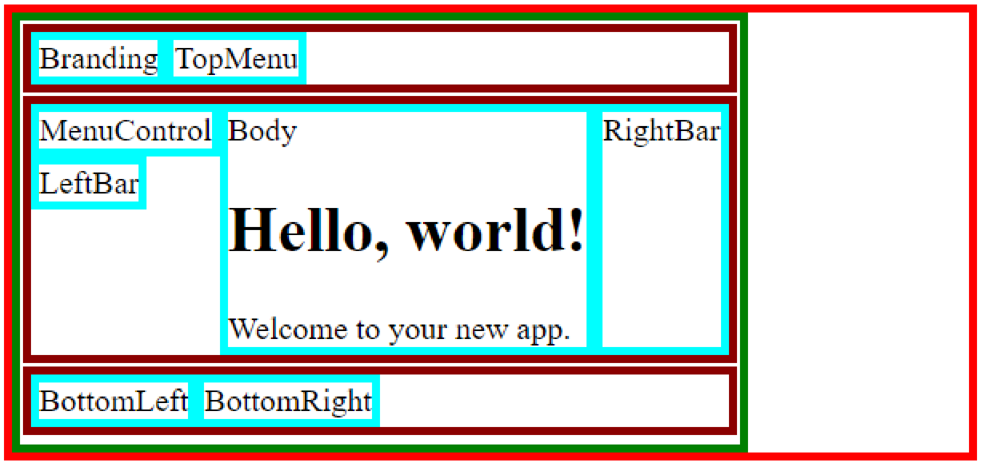
```
<div class="container">
    <table class="tableheader">

        <tr class="BrandingTopMenu">
            <div class="tablerowdiv wrapper">
                <div>
                    <div style="@ComponentStyle("Branding")"><Branding /></div>
                    <div style="@ComponentStyle("TopMenu")"><TopMenu /></div>
                </div>
            </div>
        </tr>

        <tr class="Body">
            <div class="tablerowdiv wrapper">
                <div style="@ComponentStyle("MenuControl")"><MenuControl /></div>
                <div style="@ComponentStyle("LeftBar")"><LeftBar /></div>
                <div style="display:inline-flex; float:right;">
                    <div>@Body</div>
                    <div style="@ComponentStyle("RightBar")"><RightBar /></div>
                </div>
            </div>
        </tr>

        <tr class="BottomLeftRight">
            <div class="tablerowdiv wrapper">
                <div style="@ComponentStyle("BottomLeft")"><BottomLeft /></div>
                <div style="@ComponentStyle("BottomRight")"><BottomRight /></div>
            </div>
        </tr>

        <tr class="StatusBar"><div class="tablerowdiv" style="@ComponentStyle("StatusBar")"><StatusBar
/></div></tr>

        <tr class="Footer"><div class="tablerowdiv" style="@ComponentStyle("Footer")"><Footer /></div></tr>

    </table>
```

```
</div>
```

Now, we control the visibility of all navigation components from data.

As a test, let’s set the visibility of the Footer component to “false” and run the project:

```
c.Add(new Component { Name = "Footer", Description = "Footer", IsVisible = false });
```



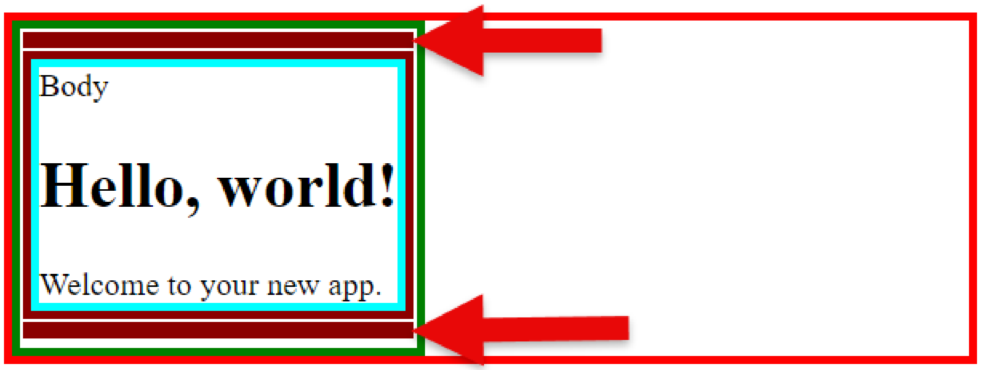
Notice that the “StatusBar” and “Footer” components are no longer displayed.

Notice that the “LeftBar” component size is smaller than the “MenuControl” component. This was caused by the additional <div> elements where the size of these two components is now decoupled. This is a good thing, as we will see later.

Cleanup unnecessary table-row borders

Turn off all components

Now that we have implemented data-driven component visibility, let’s turn off all components and see what we get:



Notice the table-row border (implemented as a <div>) shows for the rows containing two components. We want to make these go away.

Add method to remove unnecessary border

We can create a method to identify if a border is needed if all components in the row are not visible:

Add the following method to “Components.cs”:

```
public string borderStyle(string ComponentName1,string ComponentName2)
{
    int index1 = ComponentsCollection.Select((c, i) => new { c, i }).First(x => x.c.Name == ComponentName1).i;
    int index2 = ComponentsCollection.Select((c, i) => new { c, i }).First(x => x.c.Name == ComponentName2).i;
    return ComponentsCollection[index1].IsVisible == true
```

```
        || ComponentsCollection[index1].IsVisible == true
        ? "border: 4px solid darkred" : "border: none";
}
```

Add the following method to “MainLayout.razor”:

```
    public string borderStyle(string ComponentName1, string ComponentName2)
    {
        return components.borderStyle(ComponentName1, ComponentName2);
    }
```

Add the highlighted code to the “BrandingTopMenu” row:

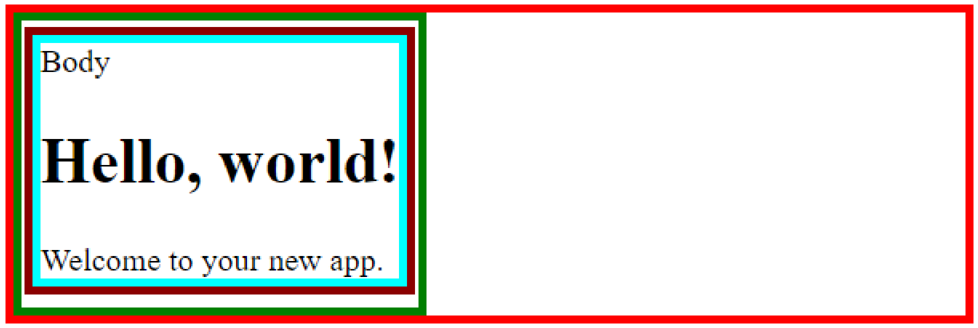
```
        <tr class="BrandingTopMenu">
            <div class="tablerowdiv wrapper" style="@borderStyle("Branding","TopMenu")">
                <div style="@ComponentStyle("Branding")"><Branding /></div>
                <div style="@ComponentStyle("TopMenu")"><TopMenu /></div>
            </div>
        </tr>
```

Add the highlighted code to the “BottomLeftRight” row:

```
        <tr class="BottomLeftRight">
            <div class="tablerowdiv wrapper" style="@borderStyle("BottomLeft","BottomRight")">
                <div style="@ComponentStyle("BottomLeft")"><BottomLeft /></div>
                <div style="@ComponentStyle("BottomRight")"><BottomRight /></div>
            </div>
        </tr>
```

Verify Behavior

Run project and validate extra borders no longer display:

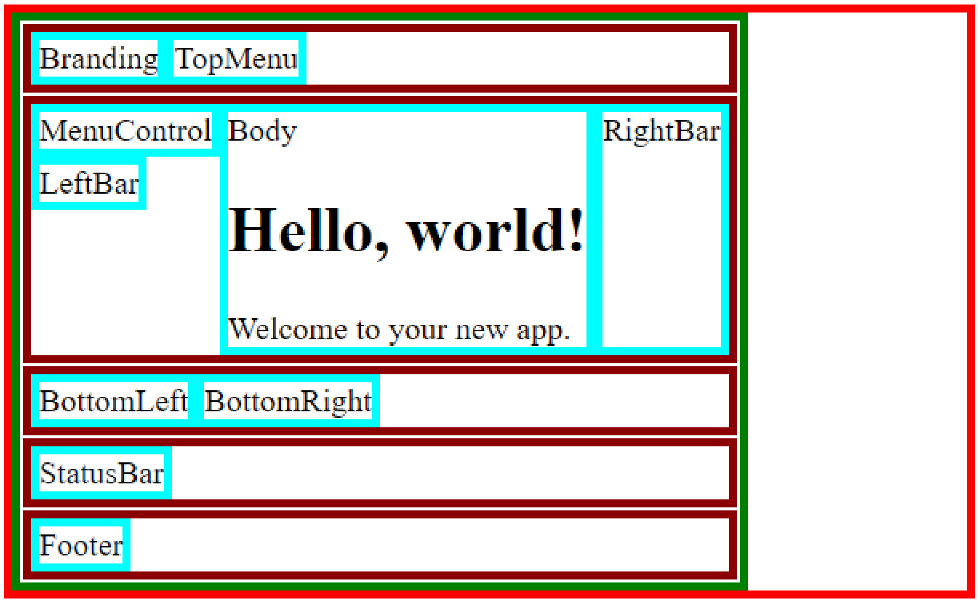

Body

Hello, world!

Welcome to your new app.

Turn on all components

Turn on all components and run the project to validate the data-driven component visibility additions:



Appearance Behavior for Components

In this section, we are going to enhance the MainLayout.razor component to use the StyleAttributes data model to allow the administrator to add and modify user-selectable “Themes.”

MainLayout.razor Component

We are going to enhance the HTML for the “container <div>”, “table”, “table row <div>”, and all the “components” within the MainLayout component to use administrator-created data-driven styles.

For illustration purposes, I am not removing the border style from the “MainLayout.razor.css” class definitions to illustrate that the dynamically inserted “style” specifications override the class definitions.

MainLayout Container <div>

The first implementation of “StyleAttributes” includes configuring “Programs.cs” and “MainLayout.razor” for dependency injection of the StyleAttributes class. Follow the pattern set in the “Component Data Model” section.

We are going to change the border style on the “container” to “`solid 2px red`” by adding the following “StyleAttribute” record (Already added in the “Style Attribute Data Model” section.

StyleAttribute.cs

```
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "Container", Key = "border",
Value = "solid 2px red" });
```

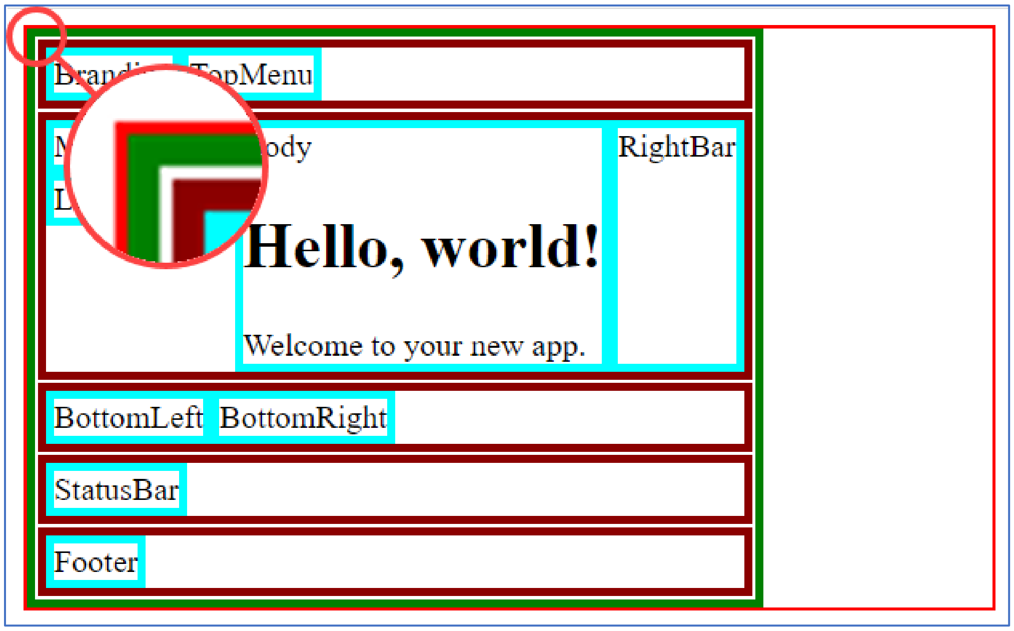
MainLayout.razor

Implementation in “MainLayout.razor” becomes quite simple:

```
<div class="container" style="@styleAttributes.GetSettings("default","MainLayout","Container")">
```

Example

Run the project and observe the changes:



Notice the "container" border is now "`solid 2px red`".

MainLayout TableHeader and TableRowDiv

We are going to reduce the size of the borders to 2px.

StyleAttribute.cs

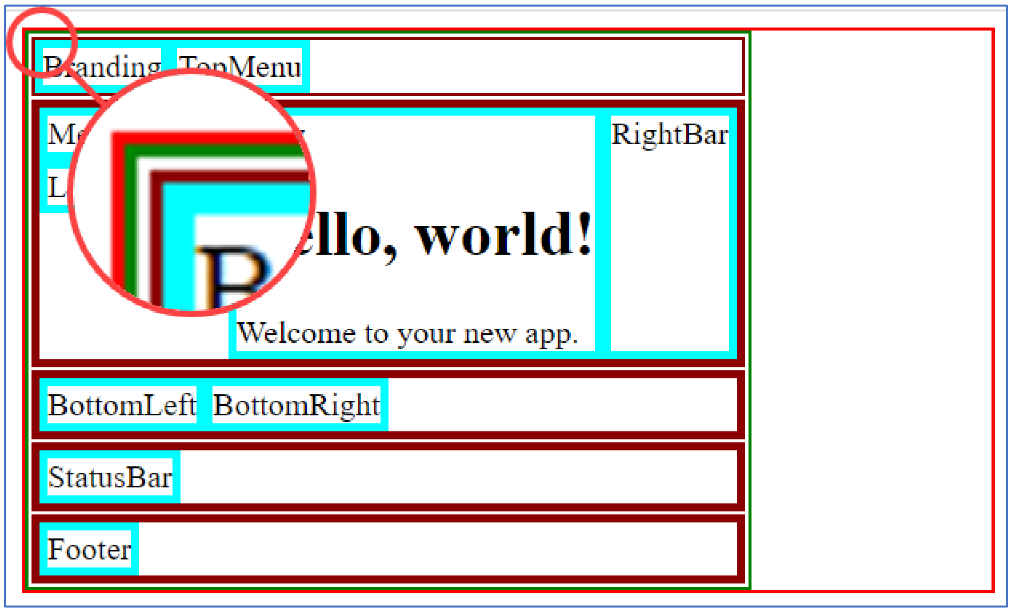
```
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "TableHeader", Key = "border",
Value = "solid 2px green" });
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "TableRowDiv", Key = "border",
Value = "solid 2px darkred" });
```

MainLayout.razor

```
<table class="tableheader" style="@styleAttributes.GetSettings("default","MainLayout","TableHeader")">

    <tr class="BrandingTopMenu">
        <div class="tablerowdiv wrapper" style="@(borderStyle("Branding","TopMenu")
                                          + @styleAttributes.GetSettings("default","MainLayout","TableRowDiv"))">
            <div style="@ComponentStyle("Branding")"><Branding /></div>
            <div style="@ComponentStyle("TopMenu")"><TopMenu /></div>
        </div>
    </tr>
```

Notice: the new code added to the "tablerowdiv" section includes a parenthesis ("(") following the first "@" sign. This parenthetical allows the "+" sign to concatenate the new style attributes with the previously retrieved ones. There are two border specifications in this "style=" specification, and the last one in the string wins.

Example



Observe the red “container” border is now 2px wide followed by a green “tableheader” 2px border. These borders are followed by what appears to be a 2px row white border we have been unable to manage, followed by the “tablerowdiv” 2px border, and the 4px border of the Branding component.

MainLayout Branding and TopMenu

Let’s set the Branding border to 2px aqua and the TopMenu border to 2px darkblue.

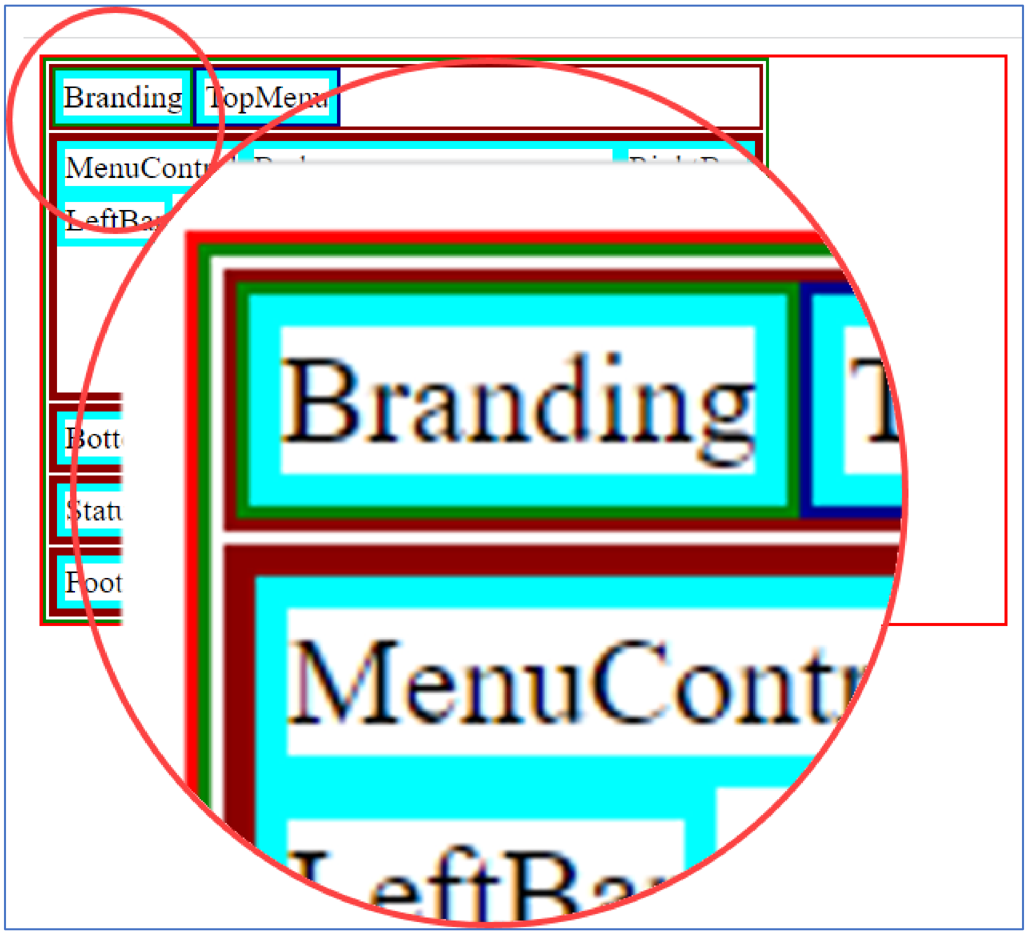
StyleAttribute.cs

```
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "Branding", Key = "border",
Value = "solid 2px Green" });
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "TopMenu", Key = "border",
Value = "solid 2px darkblue" });
```

MainLayout.razor

```
<tr class="BrandingTopMenu">
    <div class="tablerowdiv wrapper" style="@(borderStyle("Branding","TopMenu")
                                              + @styleAttributes.GetSettings("default","MainLayout","TableRowDiv"))">
        <div style="@(ComponentStyle("Branding")
                  + @styleAttributes.GetSettings("default","MainLayout","Branding"))"><Branding /></div>
        <div style="@(ComponentStyle("TopMenu")
                  + @styleAttributes.GetSettings("default","MainLayout","TopMenu"))"><TopMenu /></div>
    </div>
</tr>
```

Example



Another lesson learned: Originally, we assigned the component borders to 4px aqua inside the component, but in this example, we have added an administrator-configurable border for the component from within “MainLayout.razor.” So, in the above example, you will notice the 2px tablerowdiv dark red border followed by the 2px green border around “Branding” and the 2px dark blue border around “TopMenu,” followed by the 4px aqua border from within the components.

MainLayout MenuControl, LeftBar, Body, RightBar

Now we are ready to add configurable borders around MenuControl, LeftBar, Body, and RightBar:

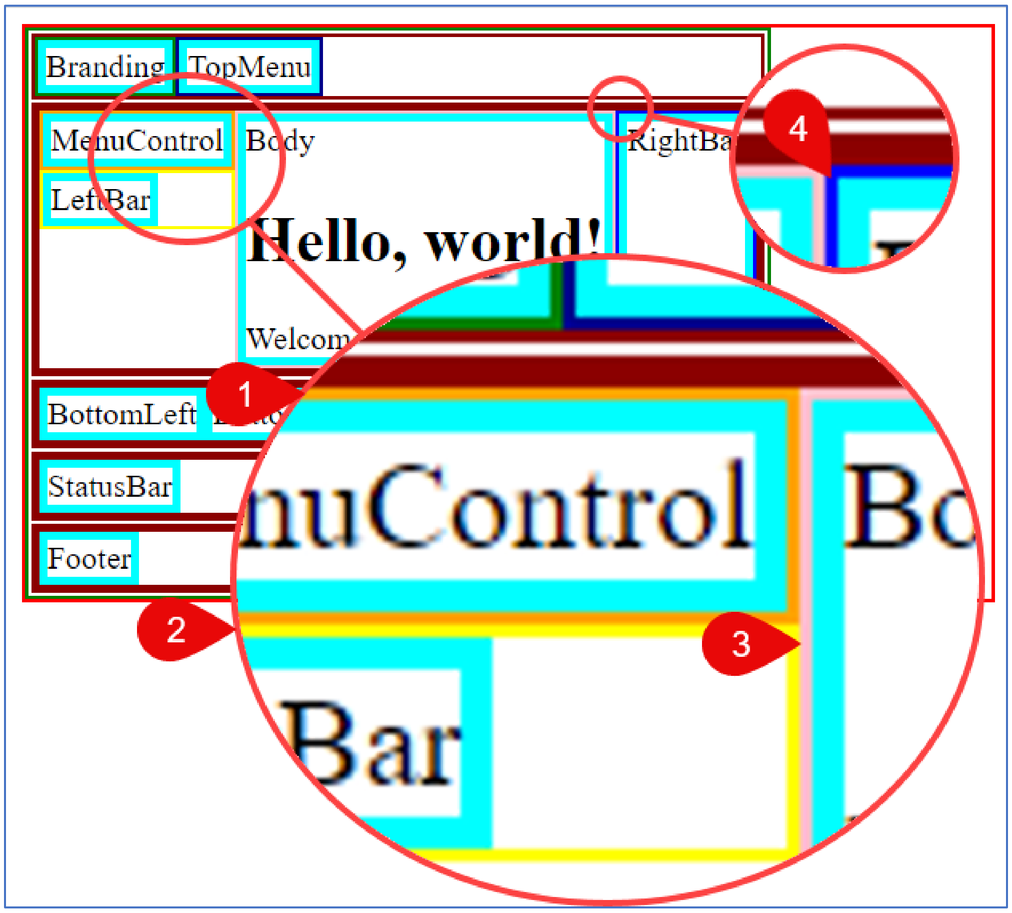
StyleAttribute.cs

```
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "MenuControl", Key = "border",
Value = "solid 2px orange" });
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "LeftBar", Key = "border",
Value = "solid 2px yellow" });
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "Body", Key = "border", Value =
"solid 2px pink" });
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "RightBar", Key = "border",
Value = "solid 2px blue" });
```

MainLayout.razor

```
<tr class="Body">
    <div class="tablerowdiv wrapper">
        <div>
            <div style="@(ComponentStyle("MenuControl")
                        + @styleAttributes.GetSettings("default","MainLayout","MenuControl"))">
                        <MenuControl /></div>
            <div style="@(ComponentStyle("LeftBar")
                        + @styleAttributes.GetSettings("default","MainLayout","LeftBar"))">
                        <LeftBar /></div>
        </div>
        <div style="display:inline-flex; float:right;">
            <div style="@styleAttributes.GetSettings("default","MainLayout","Body")">
                @Body</div>
            <div style="@(ComponentStyle("RightBar")
                        + @styleAttributes.GetSettings("default","MainLayout","RightBar"))">
                        <RightBar /></div>
        </div>
    </div>
</tr>
```

Example



1. Orange border around MenuControl.
2. Yellow border around LeftBar.
3. Pink border around Body.
4. Blue border around RightBar.

MainLayout BottomLeft and BottomRight

Add borders to BottomLeft and BottomRight components.

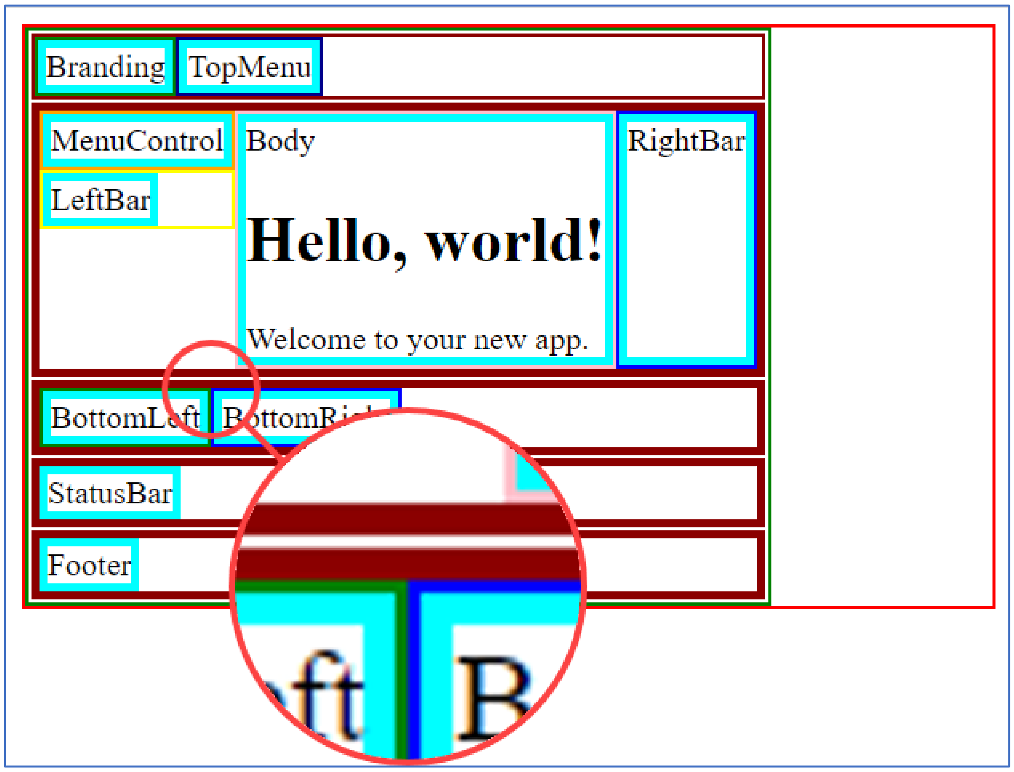
StyleAttributes.cs

```
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "BottomLeft", Key = "border",
Value = "solid 2px green" });
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "BottomRight", Key = "border",
Value = "solid 2px blue" });
```

MainLayout.razor

```
<tr class="BottomLeftRight">
    <div class="tablerowdiv wrapper" style="@borderStyle("BottomLeft","BottomRight")">
        <div style="@(ComponentStyle("BottomLeft")
                        + @styleAttributes.GetSettings("default","MainLayout","BottomLeft"))">
                        <BottomLeft /></div>
            <div style="@(ComponentStyle("BottomRight")
                        + @styleAttributes.GetSettings("default","MainLayout","BottomRight"))">
                        <BottomRight /></div>
    </div>
</tr>
```

Example



Notice green border around BottomLeft and blue border around BottomRight

MainLayout StatusBar and Footer

Add borders to StatusBar and Footer

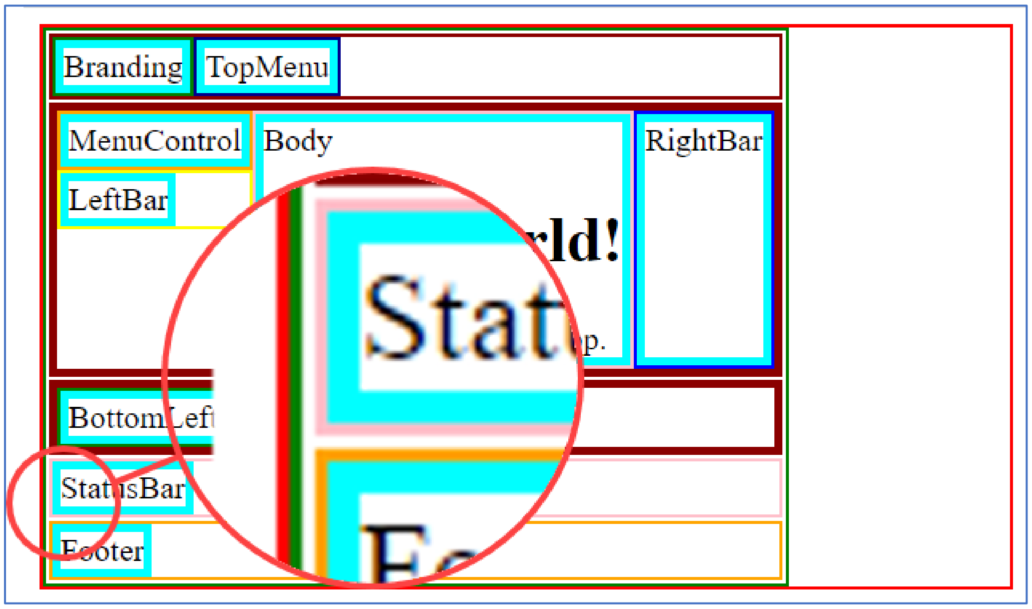
StyleAttributes.cs

```
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "StatusBar", Key = "border",
Value = "solid 2px pink" });
sa.Add(new StyleAttribute { Theme = "default", Section = "MainLayout", SubSection = "Footer", Key = "border", Value
= "solid 2px orange" });
```

MainLayout.razor

```
<tr class="StatusBar">
    <div class="tablerowdiv" style="@(ComponentStyle("StatusBar")
                                   + @styleAttributes.GetSettings("default","MainLayout","StatusBar"))">
                                   <StatusBar /></div>
</tr>

<tr class="Footer">
    <div class="tablerowdiv" style="@(ComponentStyle("Footer")
                                   + @styleAttributes.GetSettings("default","MainLayout","Footer"))">
                                   <Footer /></div>
</tr>
```

Example



Observe the pink border around StatusBar and the orange border around Footer.

What did we do?

7. We created a Visual Studio-style layout showing how to layout several component scenarios.
8. We created a data-driven mockup to manage the visibility of the components.
9. We created a data-driven mockup for administrator-managed and user-selectable styles for themes.
10. We enhanced MainLayout.razor to apply data-driven styles to the components.
11. We described in detail each step during the derivation of this layout so developers can enhance it as needed.

What is next?

1. We will enhance this template using only Microsoft components for the Menus.
2. We will then enhance the template and menus to support secrets.
3. Then, we will enhance the template to support user authentication and authorization.
4. Then, we will add a simple application.
5. Then, we will deploy to Azure.