Blazor Server

Building an Application

Project Structure and Layouts

Components

2023

Outline

- Creating a Blazor application
- Key Components
- Organizing files using feature folders
- Defining Layout
- Routable Components
- Example

Creating the Application

- In the terminal, run the following command to create your app:

```
dotnet new blazorserver -o app --no-https -f net7.0
```

- The dotnet watch command will build and start the app, and then update the app whenever you make code changes. You can stop the app at any time by selecting Ctrl+C.

```
cd app
dotnet watch
```

Key components of a Blazor application

- _Host.cshtml
  - The root page of the app
  - When any page of the app is initially requested, this page is rendered and returned in the response.
  - The Host page specifies where the root App component (App.razor) is rendered.
- Program.cs
  - The app's entry point that sets up the ASP.NET Core host and contains the app's startup logic, including service registrations and request processing pipeline configuration
- App.razor
  - The root component of the app that sets up routing using the Router component. The Router component intercepts browser navigation and renders the page that matches the requested address.

Key components of a Blazor application

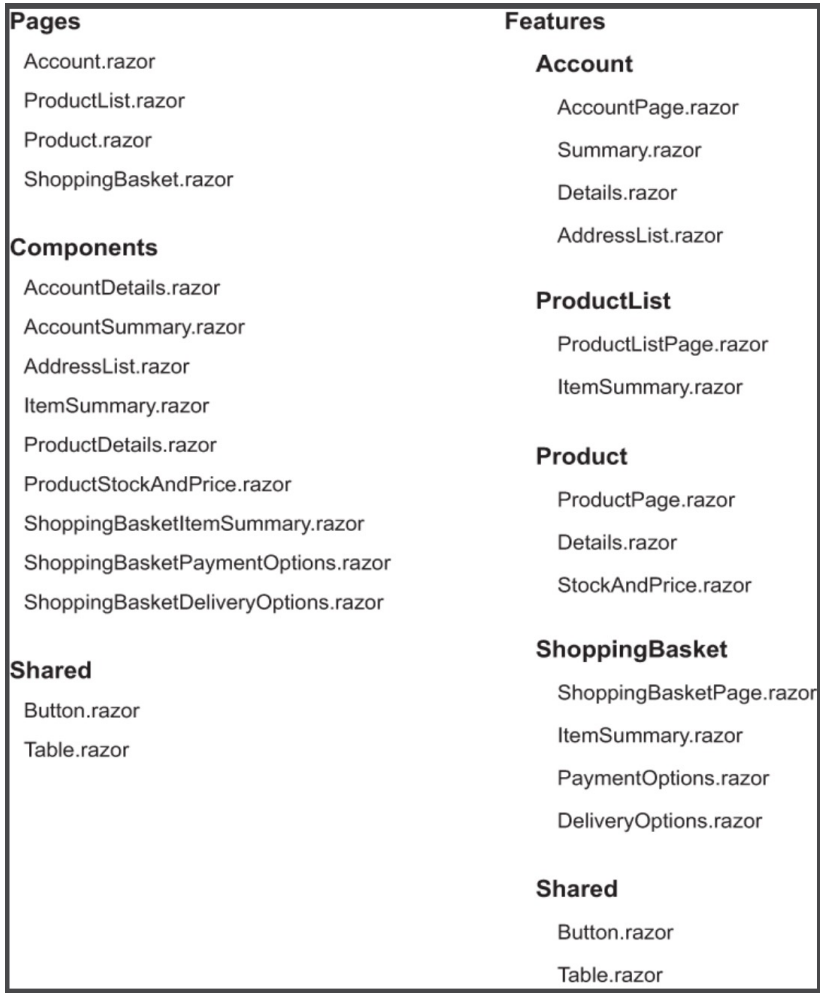
- wwwroot folder and _Imports.razor
  - All ASP.NET Core applications have a wwwroot folder, which is used to store public static assets.
  - This is the place where you can put things such as images, CSS files, JavaScript files, or any other static files you need
  - Its job is to store using statements. The benefit is that those using statements are made available to all the components in the file's directory and any subdirectories.

Organizing files using feature folders

- By default, the app structure used by the template divides files by responsibility.
- There's a Pages folder for routable components, and there's a Shared folder for anything that is used in multiple places or is a global concern.
- This kind of separation doesn't scale well and makes adding or changing functionality much more difficult, as files end up being spread out all over the place.
- Instead, we're going to use a structure called feature folders to organize our application.

Organizing files using feature folders

- When using feature folders, all the files relating to that feature are stored in the same place.
- When you go to work on a particular feature, all the files you need are in the same place, making everything easier to understand and more discoverable.
- Every time you add a new feature to the app, you just add a new folder and everything goes in there. You can also arrange each feature with subfeatures if they contain a lot of files.



Organizing files using feature folders

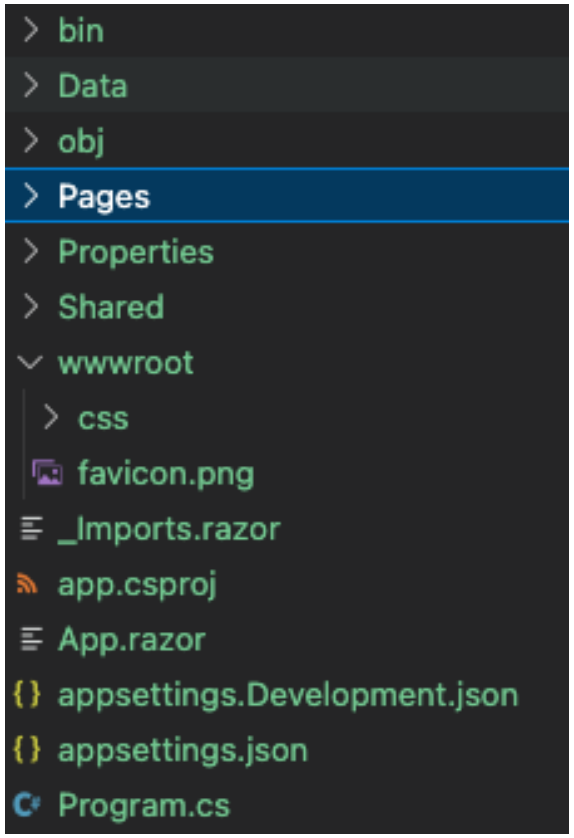
- You should put any files that relate to that feature in the folder—C# classes, TypeScript files, CSS files, anything at all.
- Static assets such as images are the only exception to this. These need to be placed in the wwwroot folder; otherwise they will not be available at run time.
- However, you can mirror your feature folder structure in the wwwroot folder if you wish.

Organizing files using feature folders

- The other useful little thing to do when using this organization system with Blazor is to append any routable component with the word Page.
- When a feature has several other components in it, it’s almost impossible to identify the routable component easily.

Organizing files using feature folders

Original structure

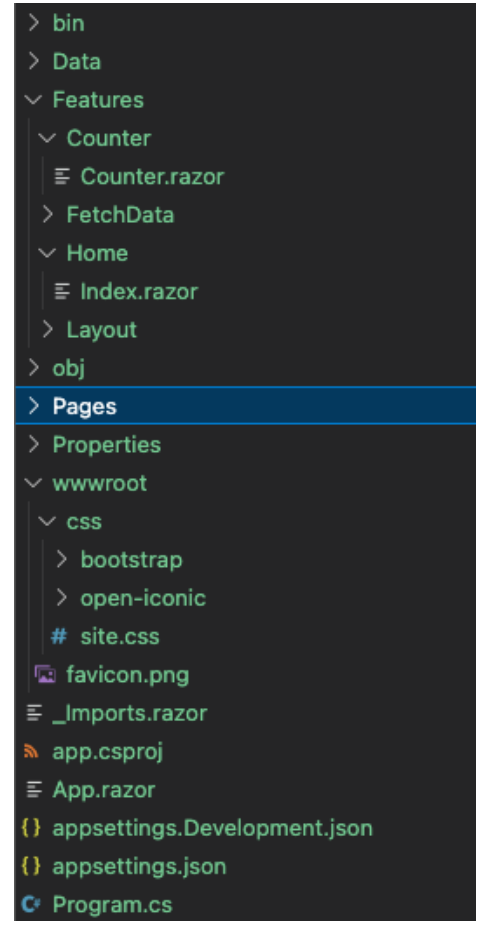


Feature Folder

One folder for feature

Pages is preserved for holding _Host.cshtml and Error.cshtml

Remove shared folder and adjust references in _imports.razor



Setting stylling

- The Blazor templates ship with a CSS framework called Bootstrap.
- You can place images in wwwroot in a folder called images.
- You can add some custom styles to the site.css file.
- By adding the styles here, they will affect the whole application.
- These styles will customize the look of some common elements, and are placed in the wwwroot > css folder.

Setting stylling

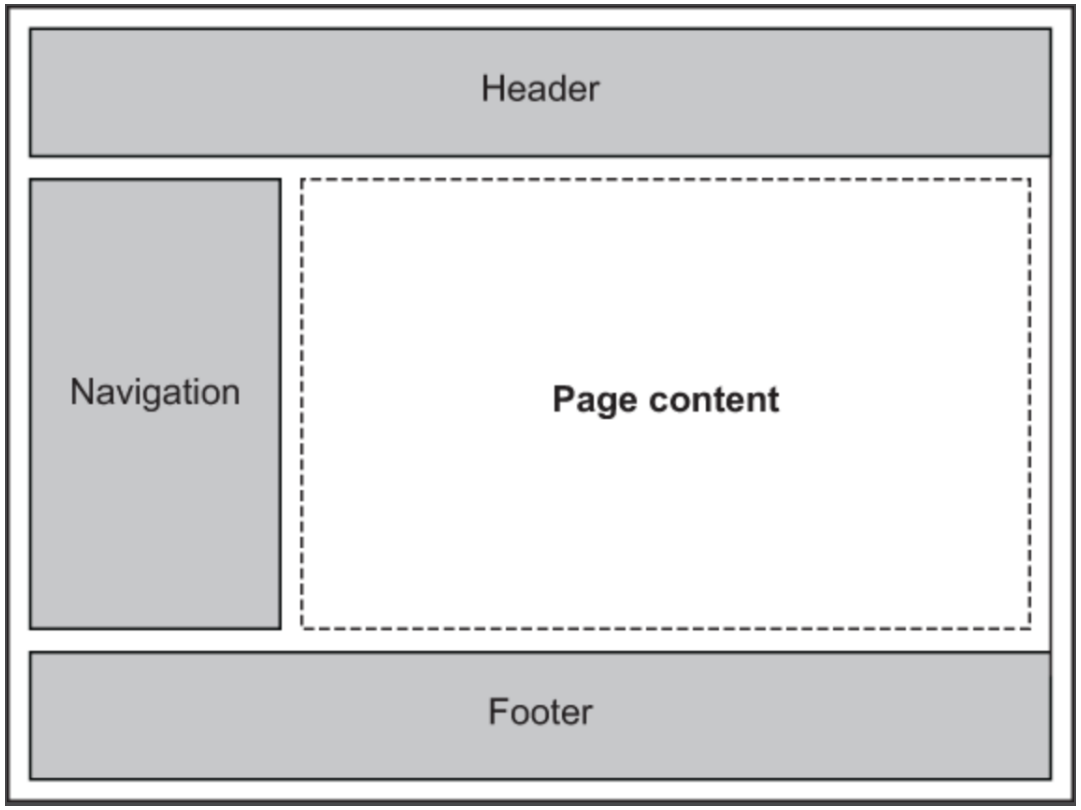
- You can make some adjustments to the _Host.cshtml page.
- First, we'll add a reference to Bootstrap Icons (https://icons.getbootstrap.com/).
- In order to use it, we'll add the following line to the head element of the index.html page in wwwroot:

```
@page "/"
@using Microsoft.AspNetCore.Components.Web
@namespace app.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="app.styles.css" rel="stylesheet" />
    <link rel="icon" type="image/png" href="favicon.png"/>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap-"icons@1.4.1/font/boo
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
</head>
```

Defining Layout

- The concept of a layout allows us to define common UIs, which is required by multiple pages
- Things such as the header, footer, and navigation menu are all examples of things you might put in your layout.
- We also add a reference to a parameter called Body where we want page content to be rendered.
- This comes from a special base class that all layouts in Blazor must inherit from called LayoutComponentBase.

Defining Layout



In Blazor, the default layout is defined within the Router component, which can be found in App.razor

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
    <NotFound>
        <PageTitle>Not found</PageTitle>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Defining Layout

- If you want to use a different layout on certain pages, you can specify an alternative by applying the @layout directive.
- This goes at the top of the page, and you pass the name of the component you wish to use.
- For example, if we had an alternative layout called AdminLayout, our layout directive would look like this: @layout AdminLayout.

Defining Layout

- We're going to update the MainLayout component.

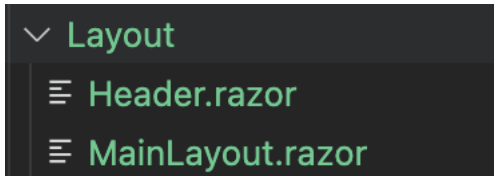
```
@inherits LayoutComponentBase

<main class="container mt-5 mb-5">
    @Body
</main>
```

Defines the component as a layout component

Marks the location where page content is rendered in the layout

Defining Layout

- The Header will be defined in a separate component: Header.razor




```
<nav class="navbar mb-5 shadow">
    <a class="navbar-brand" href="/">
        <img src="/images/logo.png">
    </a>
</nav>
```

https://gist.github.com/brunobmo/d121e1d18fac84fcc1081cc527fb9876

- We can now add that to the MainLayout:

```
@inherits LayoutComponentBase

<Header />

<main class="container mt-5 mb-5">
  @Body
</main>
```

https://gist.github.com/brunobmo/a0b57ba04f60177b61be73fdcdf45f83

Routable Components

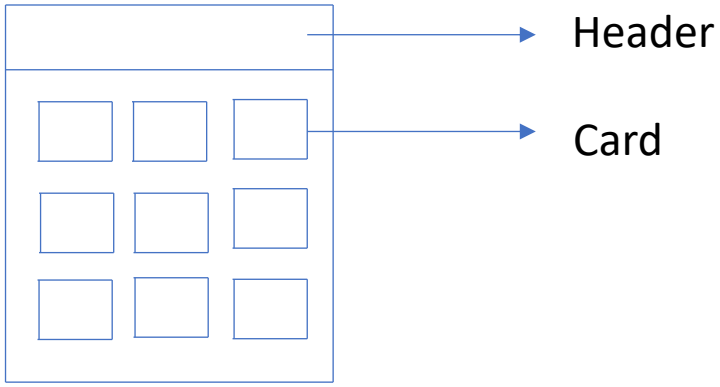
- To make a component routable, we need to use the @page directive and a route template that specifies the route it will be responsible for:

```
@page "/"

<PageTitle>Index</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for
you?" />
```

When a route template contains only a forward slash (/), it tells the router that this is the root page of the application.

Creating a page

- Let’s create the concept of “Card” to create a page with the following structure:



Creating a page

```
namespace app.Features.Home;

public class Card
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public string Description { get; set; } = "";
    public string Image { get; set; } = "";
}
```

https://gist.github.com/brunobmo/cca963ea4e3f717a603fcdbc6e7997db

Creating a page

- Let’s create some sample data.
- We will do an http call to load data from the backend.


```
[
  {
    "id": 1,
    "name": "Card 1",
    "description": "Card 1 description",
    "image": "(…)"
  },
  {
    "id": 2,
    "name": "Card 2",
    "description": "Card 2 description",
    "image":"(…)"
  }
]
```


https://gist.github.com/brunobmo/8623e8b9744356b960fcc9f5f968f9ba

Creating a page

- We’re going to load the data using the HttpClient, but to use it we need to get an instance of it using dependency injection.
- Blazor makes this easy by providing an inject directive: @inject [TYPE] [NAME], where [Type] is the type of the object we want and [Name] is the name we’ll use to work with that instance in our component.
- Under the page directive, add @inject HttpClient Http, which will give us an instance of the HttpClient to work with;

Creating a page

- The inject directive:
  - The inject directive allows us to quickly and easily inject instances of objects, registered with the service container in Program.cs, into our components.
  - So, if we use the Inject directive for http client, it ends up compiled to this:

```
[Inject]
public HttpClient Http { get; set; }
```

Creating a page

- Let's change the Index.Razor page (inside Home folder) to this:

```
@page "/"
@inject HttpClient Http

<PageTitle>Index</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />

@code{
    private IEnumerable<Card>? _cards;
}
```

The Inject directive is used to get instances of objects from the dependency injection container.

The Private field holds cards data

https://gist.github.com/brunobmo/7143d6093dc6275055f750cc6de0ad10

Creating a page

- For using the Http client you need to:
  - Install the Nugget package:
    - dotnet add package System.Net.Http --version 4.3.4
  - Regist the servisse in program.cs file:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;
using app.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();
builder.Services.AddHttpClient();

(...)
```

Creating a page

- Now that we have somewhere to store our test data, we can make the call to retrieve it.
- A great place to do this kind of thing is the OnInitialized life cycle method.
- This method is provided by ComponentBase—which all Blazor components inherit from—and it's one of three primary life cycle methods.
- OnInitialized is run only once in the component's lifetime, making it perfect for loading initial data;

Creating a page

- To retrieve the data from the JSON file, we can make a GET request just like we would if we were reaching out to an API.
- As the file is in the wwwroot folder, it will be available as a static asset at run time, just like the CSS file.

Creating a page

- Let's change the Index.razor:

```
@inject HttpClient Http
@inject NavigationManager NavigationManager

(...)

@code {
    private IEnumerable<Card>? _cards;
    protected override async Task OnInitializedAsync(){
        try{
            string domainName = NavigationManager.Uri;
            _cards = await Http.GetFromJsonAsync<IEnumerable<Card>>(domainName+"cards/cards.json");
        }
        catch (HttpRequestException ex){
            Console.WriteLine($"There was a problem loading amiibos data: {ex.Message}");
        }
    }
}
```

Note: Inject is used to manage dependencies between objects, while Using is used to manage the lifecycle of IDisposable objects.

https://gist.github.com/brunobmo/dc1a1e666ec0d72111feabd8aefd70c9

Creating a page

- For showing data, we can use a simple if statement in our markup to check the value of the _cards field.
- If it's null, then we can surmise that the data is still being loaded, excluding an error scenario, of course.
- If the value is not null, then we have some data, and we can go ahead and display it (see the following listing).

Creating a page

Index.razor

```
@page "/"
(...)
@if (_cards == null) {
    <p>Loading cards...</p>
}
else{
    <div class="grid">
      @foreach (var card in _cards) {
        <div class="card shadow" style="width: 18rem;">
          <img src="@card.Image" class="card-img-top" alt="@card.Name">
          <div class="card-body">
            <h5 class="card-title">@card.Name</h5>
            <h6 class="card-subtitle mb-3 text-muted">
              <span class="oi oi-map-marker"></span>
              @card.Name
            </h6>
        (...)
        </div>
      }
    </div>
}
(...)
```

https://gist.github.com/brunobmo/cec57870da873ad48e289375722c5399

Creating a page

- Let’s modify the previous code to encapsulate it all in a componente.
- This would make the code in the HomePage component much easier to read.
- Create a new component called Card.razor in the Home feature folder.
- Then replace the boilerplate code with the markup for the card from the Index page

Creating a page

CardComponent.razor

```
<div class="card shadow" style="width: 18rem;">
   <img src="@card.Image" class="card-img-top" alt="@card.Name">
       <div class="card-body">
         <h5 class="card-title">@card.Name</h5>
          <h6 class="card-subtitle mb-3 text-muted">
           <span class="oi oi-map-marker"></span>
              @card.Name
          </h6>
       (...)
       </div>
     }
   </div>
}
(...)
```

Now we have a problem. How do we get access to the current card data? The answer is parameters

Creating a page

- We can pass data into components via parameters.
- Think of these as the public API for a component, and they work one way, from parent to child.
- We can define them in the code block by creating a public property and decorating it with the Parameter attribute.
- We pass data into them from the parent using attributes on the component tag.

Creating a page

- In addition to using the Parameter attribute, we've also added another attribute called EditorRequired.
- We can use it to indicate that a parameter is required.
- If we try to use the Card component now, without passing a card to the card parameter, we'll get a warning.

CardComponent.razor

```
(...)
@code {
   [Parameter, EditorRequired]
    public Card card { get; set; } = default!;
}
```

https://gist.github.com/brunobmo/179839ff0f1c0a6e15153d09a1fb2dc0

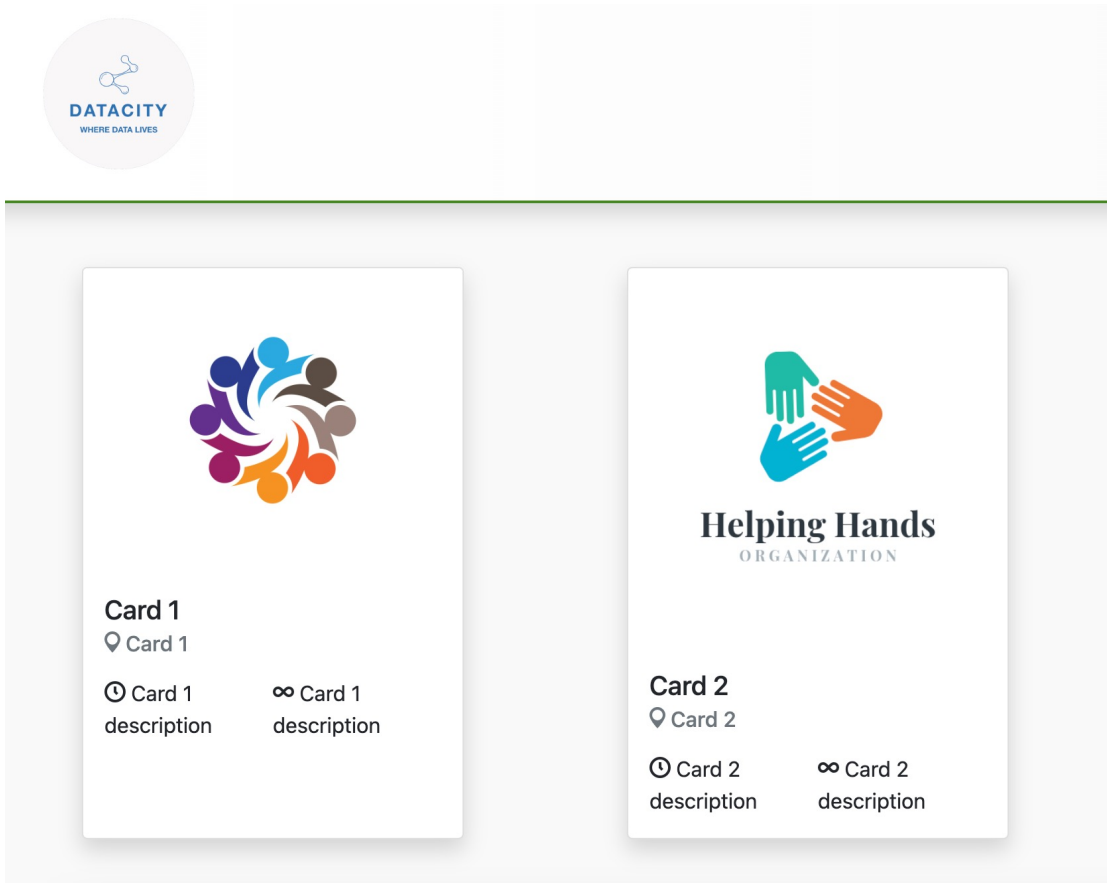
Creating a page

- Let's update Index.razor

```
@if (_cards == null) {
    <p>Loading cards...</p>
}
else{
    <div class="grid">
     @foreach (var card in _cards) {
       <Card card="card" />
     }
    </div>
}
```

https://gist.github.com/brunobmo/fd3321a13ea263a7be44764115ebb4bb

Final Result



Resume

- Blazor applications use a host page that contains the HTML element where the Blazor app will be rendered
- App.razor is the default root component and contains the Router component. All other components will be rendered as children of App.
- Feature folders can offer a number of benefits when organizing the files in your application.
  - Everything that's related to a feature is in one place, making updates and maintenance easier.

Resume

- Layout components are a great way to define common UI, which would be repeated on every page, such as headers and navigation menus.
- Values can be passed into components via parameters, which can be thought of as the API for a component.
- Parameters must be public properties; they cannot be private.

Blazor Server

Building an Application

Project Structure and Layouts

Components

2023