Blazor Server

Routing

2023

Introducing client-side routing

- Client-side routing differs substantially from traditional navigation in server-based web applications.
- To navigate to another page in traditional multipage apps, a request is made to the server for the new page.
- The new page is then downloaded to the browser, and the browser renders it.
- With SPAs, generally speaking, all of the pages reside on the client and navigating between them is handled by a client-side router.

Blazor's router

- In Blazor, the router is just another component, and you can find it inside the App component (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>
```

The router uses reflection to scan for page components. The AppAssembly parameter is used to tell the router where to scan.

The Found template is where page components that match a requested route are loaded.

The NotFound template is shown when the router can't find a match for the requested route in its routing table

Blazor's router

- When a Blazor app first loads, the Router component uses reflection to scan the application's assemblies to find routable components, or call page components.
- These are components that have a special directive declared in them called @page.
- The @page directive allows us to specify what route the component will be loaded for.
- The router then stores the type of the component and the route it handles in a routing table.
- The router then listens for navigation events

Blazor's router

- When a link is clicked, a navigation event is triggered.
- Blazor has infrastructure that lives in the JavaScript world, and one of the things that code does is intercept various events, including navigation events.
- The URL that the link points to is passed to a JavaScript service called NavigationManager;
- The final step the service takes is to raise an event that triggers some JavaScript interop.
- This event is picked up by a service with the same name, NavigationManager, that lives in the C# world

Blazor's router

- When the C# NavigationManager receives the event, it updates its URI property.
- This property stores the current URL so components can access it if required.
- It then triggers an event called LocationChanged.
- Blazor's router subscribes to this event, and when it fires, the router checks the URI property of the NavigationManager against its routing table to find a match.
- If one is found, then the component is loaded; otherwise a NotFound template is rendered.

Defining page components

- Page components are regular components that declare a specific directive—the @page directive. It has two parts, the directive name and the route template, and when declared looks like this: @page "/my-awesome-page
- The route template is the section in quotes. This defines the URL that the component will handle—it must always start with a forward slash (/);
- It's perfectly fine to have a single component declare multiple @page directives and handle multiple routes

Defining page components

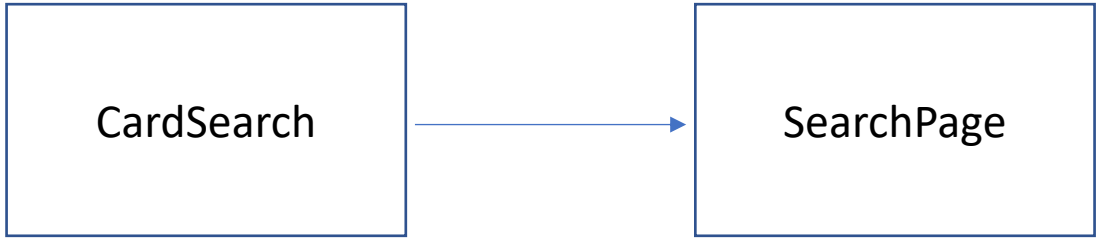
- Let’s add a new page component to the application.
- Add a new Razor component in the Features > Home folder called SearchPage.razor with the code

```
@page "/search”

<PageTitle>Search Cards</PageTitle>

<h3>Search results</h3>
```

https://gist.github.com/brunobmo/948afc962a8ad478f5718428e18670a8

Navigating between pages programmatically

- In Blazor, programmatic navigation is achieved via the NavigationManager.NavigateTo() method.
- We need to create a new component in the Home feature folder called CardSearch.razor.
- This component will house the search box and logic for redirecting to the SearchPage.



Navigating between pages programmatically

```
@inject NavigationManager NavManager

<div class="jumbotron">
    <h1 class="display-4 text-center">Welcome</h1>
    <p class="lead text-center">Find cards using our search!
</p>
    <p class="mt-4">
        <input @onkeydown="SearchForCard"
               @bind="_searchTerm"
               @bind:event="oninput" type="text"
       placeholder="Search for a card..." class="form-control form-
control-lg" />
    </p>
</div>

@code {
  private string _searchTerm = "";
  private void SearchForCard(KeyboardEventArgs args){
    if (args.Key != "Enter") return;
      NavManager.NavigateTo($"/search/{_searchTerm}");
    }
}
```

An instance of the NavigationManager is injected using the @inject directive.

The SearchForCard method is called every time a keydown event is fired.

The @bind directive allows two-way binding in Blazor. Here we're binding the text the user inputs to the _searchTerm field.

Update the _searchTerm field whenever the oninput event fires, essentially when a new character is entered

Return if the key pressed wasn't the Enter key.

Use NavigationManager.NavigateTo to programmatically navigate to the search page, passing the search term entered by the user.

CardSearch.razor

https://gist.github.com/brunobmo/0a739def732a37ecb5db7975fc525e54

Navigating between pages programmatically

- We are using the binding directive (@bind)
- Instead of the default onchange event, we're using the oninput event, which updates the _searchTerm field every time a new character is typed into the input.
- Now, when the SearchForCard method is called, the _searchTerm field is populated correctly.

Navigating between pages programmatically

- We’re also going to add some scoped styles for the CardSearch component.
- Add a new CSS file in the Home folder called CardSearch.razor.css, and add the code shown in the following listing.

```
.jumbotron {
    background: none;
}

.jumbotron input {
    border: 2px solid var(--brand);
}
```

https://gist.github.com/brunobmo/07afa40dd08bac67d96efaf0c9376516

Navigating between pages programmatically

- We just need to add our new CarSearch component to the HomePage.
- The following listing shows a subsection of the Index component where the CarSearch should be added.

```
@if (_cards == null)
{
  <p>Loading cards...</p>
}
else
{
   <CardSearch />
   <CardDetails card="_selectedCard" />
   <div class="grid">
     @foreach (var card in _cards)
     {
        <CardComponent card="card" OnSelected="HandleCardSelected" />
     }
   </div>
}
```

https://gist.github.com/brunobmo/9d8891e74958214831314512fc708459

Passing data between pages using route parameters

- When navigating between pages, there are times we want to pass arbitrary data as part of the URL;
- We can do this by using a special feature of route templates called template parameters.
- They are placeholders for a value that will be supplied later.
- They are paired with a component parameter that matches the name of the route parameter.
- When the component is executed, the value in the segment of the route defined by the route parameter is passed into the component parameter so we can access it in code.

Passing data between pages using route parameters

- The following listing shows the SearchPage component updated with a route parameter to capture the search term:

```
@page "/search/{SearchTerm}"

<h3 class="mt-5 mb-4">Search results for "@SearchTerm"</h3>

@code {
    [Parameter]
    public string SearchTerm { get; set; } = default!;
}
```

Route parameters are defined using curly braces in a route segment.

A component parameter matching the name of the route parameter is required to capture its value.

Passing data between pages using route parameters

- Next we can add the logic so the search page displays matching Cards.

Breadcrumbs allow navigation back to the home page

```
@page "/search/{SearchTerm}"
@inject HttpClient Http
@inject NavigationManager NavManager

<nav aria-label="breadcrumb">
   <ol class="breadcrumb">
     <li class="breadcrumb-item">
       <a href="/">Home</a>
     </li>
     <li class="breadcrumb-item active" aria-current="page">Search</li>
   </ol>
</nav>

<h3 class="mt-5 mb-4">Search results for "@SearchTerm"</h3>

@if (_searchResults == null){
   <p>Loading search results...</p>
}else{
 <CardDetails card="_selectedCard" />
 <div class="grid">
    @foreach (var card in _searchResults){
       <CardComponent card="card" OnSelected="HandleCardSelected" />
    }
 </div>
}
```

Passing data between pages using route parameters

- Next we can add the logic so the search page displays matching Cards.

```
@code {
  private IEnumerable<Card>? _searchResults;
  private Card? _selectedCard;
  [Parameter]
   public string SearchTerm { get; set; } = default!;
   protected override async Task OnInitializedAsync(){
     try{
       string domainName = NavManager.Uri;
       var allCards = await Http.GetFromJsonAsync<IEnumerable<Card>>("http://localhost:5167/cards/cards.json");
      _searchResults = allCards!.Where(x => x.Name.Contains(SearchTerm,StringComparison.CurrentCultureIgnoreCase) ||
x.Description.Contains(SearchTerm, StringComparison.CurrentCultureIgnoreCase));
     }
     catch (HttpRequestException ex) {
       Console.WriteLine($"There was a problem loading trail data: {ex.Message}");
     }
  }
   private void HandleCardSelected(Card card){
     _selectedCard = card;
     StateHasChanged();
  }
}
```

When the component is loaded, it will get all the cards from the dummy data file and find any that have a name or location that contains the search term.

Technically, the call to GetFromJsonAsync returns a null. However, we have specific test data so we can safely ignore the potential null using the null forgiving operator.

https://gist.github.com/brunobmo/dc741af32a23ce3ffd2248c338f6a6dd

Handling multiple routes with a single component

- It’s possible to have a single component be responsible for multiple routes.
- We’ll add a second route to the SearchPage component that will contain the max Id filter.
- If a filter is entered, we will redirect the user to the same page using the second route.

```
@page "/search/{SearchTerm}"
@page "/search/{SearchTerm}/maxid/{MaxId:int}"

@inject HttpClient Http
@inject NavigationManager NavManager
(...)
```

Shows the original @page directive and matching component parameter

Shows the new @page directive and matching component parameter

Handling multiple routes with a single component

- There is a significant difference in the definition of the second @page directive's route template;
- Where the MaxId route parameter is defined, there is some additional syntax: :int.
  - This is called a route constraint.

Full list of constraints: http://mng.bz/d2xg

Handling multiple routes with a single component

- We'll create the search filter as a new component called SearchFilter.razor in the Home feature folder:

The value entered by the user is bound to the _maxId field.

```
@inject NavigationManager NavManager

<div class="filters">
    <label for="maxId">Max Length</label>
    <input id="maxId" type="number" class="form-control" @bind="_maxId" />
    <button class="btn btn-outline-primary" @onclick="FilterSearchResults">Filter</button>
    <button class="btn btn-outline-secondary" @onclick="ClearSearchFilter">Clear</button>
</div>
```

Clearing an existing filter is handled by the ClearSearchFilter

Clicking the Filter button executes the FilterSearchResults method.

Handling multiple routes with a single component

- We'll create the search filter as a new component called SearchFilter.razor in the Home feature folder:

```
@code {
  private int _maxId;

  [Parameter, EditorRequired]
  public string SearchTerm { get; set; } = default!;

  private void FilterSearchResults() => NavManager.NavigateTo($"/search/{SearchTerm}/maxid/{_maxId}");

  private void ClearSearchFilter()
  {
      _maxId = 0;
    NavManager.NavigateTo($"/search/{SearchTerm}");
  }
}
```

To filter the search result, we navigate to the second route we defined for the component.

To clear the filter, we navigate to the original route

https://gist.github.com/brunobmo/218381f5853d771b4bfa8e721c9bec02

Handling multiple routes with a single component

- The SearchFilter component uses an HTML input to record the desired max id from the user.
- When the Filter button is clicked, the NavigationManager NavigateTo method is used to redirect the user to the second route we added to the SearchPage component.
- To do this, the SearchFilter component needs to know the search term, so we're specifying that as a component parameter to be supplied by the SearchPage.
- To clear a filter, the Clear button redirects the user to the original route.

Handling multiple routes with a single component

- Let’s add a new CSS file called SearchFilter.razor.css into the Home feature folder with the styles shown in the following listing.

```
.filters {
display: flex;
margin-bottom: 20px;
align-items: baseline;
justify-content: flex-end;
}

.filters label {
text-transform: uppercase;
margin-right: 10px;
}

.filters input {
margin-right: 20px;
width: 100px;
}

.filters button:first-of-type {
margin-right: 10px;
}
```

https://gist.github.com/brunobmo/5df833901336090253c4fce050d78a79

Handling multiple routes with a single component

- With the styles in place, we just need to reference the SearchFilter component in the SearchPage.
- We'll add it just under the page header:

```
@page "/search/{SearchTerm}"
@page "/search/{SearchTerm}/maxlength/{MaxLength:int}"

@inject HttpClient Http
@inject NavigationManager NavManager

<h3 class="mt-5 mb-4">Search results for "@SearchTerm"</h3>
<SearchFilter SearchTerm="@SearchTerm" />

(...)
```

Handling multiple routes with a single component

- The last task we have is to implement the filtering functionality (SearchPage).

```
@page "/search/{SearchTerm}"
@page "/search/{SearchTerm}/maxid/{MaxId:int}"
@inject HttpClient Http
@inject NavigationManager NavManager

<h3 class="mt-5 mb-4">Search results for "@SearchTerm"</h3>
<SearchFilter SearchTerm="@SearchTerm" />

<nav aria-label="breadcrumb">
   <ol class="breadcrumb">
      <li class="breadcrumb-item"><a href="/">Home</a></li>
      <li class="breadcrumb-item active" aria-current="page">Search</li>
  </ol>
</nav>
@if (_searchResults == null){
    <p>Loading search results...</p>
}
else{
   <CardDetails card="_selectedCard" />
   <div class="grid">
     @foreach (var card in _searchResults){
        <CardComponent card="card" OnSelected="HandleCardSelected" />
     }
   </div>
}
```

Handling multiple routes with a single component

- The last task we have is to implement the filtering functionality.

```
@code {
    private IEnumerable<Card>? _searchResults;
    private Card? _selectedCard;
    private IEnumerable<Card> _cachedSearchResults = Array.Empty<Card>();
   [Parameter]
    public string SearchTerm { get; set; } = default!;
    [Parameter]
    public int? MaxId { get; set; }

    protected override async Task OnInitializedAsync(){
      try{
          var allCards = await Http.GetFromJsonAsync<IEnumerable<Card>>("http://localhost:5167/cards/cards.json");
        _searchResults = allCards!.Where(x => x.Name.Contains(SearchTerm, StringComparison.CurrentCultureIgnoreCase) ||
        x.Description.Contains(SearchTerm, StringComparison.CurrentCultureIgnoreCase));
        _cachedSearchResults = _searchResults;
      }catch (HttpRequestException ex){
          Console.WriteLine($"There was a problem loading cards data: {ex.Message}");
       }
    }

(continues)
```

Stores a copy of the unfiltered search results

Check for cached search results and a filter value; if both are present, then filter the results.

Handling multiple routes with a single component

- The last task we have is to implement the filtering functionality.

```
protected override void OnParametersSet(){
    if (_cachedSearchResults.Any() && MaxId.HasValue){
      _searchResults = _cachedSearchResults.Where(x => x.Id <= MaxLength.Value);
    }else if (_cachedSearchResults.Any() && MaxId is null){
       _searchResults = _cachedSearchResults;
    }
}
(...)
```

Check for cached search results and a filter value; if both are present, then filter the results.

If there are cached search results but no filter, then reset the results to the unfiltered set.

https://gist.github.com/brunobmo/14e3de112295f8dc9a9d392cdbb9a473

Handling multiple routes with a single component

- When we enter a filter, the URL is updated but the SearchPage component is not destroyed and re-created.
- When changing routes, Blazor performs a diff just like it would with any other UI update.
- In our case, we're navigating to the same component that is already rendered, so nothing in the UI needs to change.

Setting query-string values

- The query string starts with a question mark (?), then comes the key-value pair separated by an equals sign (=).
- Query strings can be a good option when you want to work with multiple optional values.
- Example:
  - www.blazor.net?blazor=awesome
- With query strings, we can include as many or as few key-value pairs.

Setting query-string values

- We’re going to add an additional filter to our search that allows the user to filter cards based on their date
- We’ll call this max date.
- This means that users will be able to filter results based on either max id, max date, or both.

Setting query-string values

- We'll start in the SearchFilter component and add a new field to store the max time as well as a new label and HTML input to record it;
- Let's add a new property to the Card class: `public DateTime? Date { get; set; }`
- Let's add some data to the JSON file with the following structure:`"date": "2001-01-01T09:00:00Z"`

Setting query-string values

- In the SearchPage.razor change the route added to:

```
@page "/search/{SearchTerm}"
@page "/search/{SearchTerm}/maxid/{MaxId:int}/maxdate/{MaxDate:datetime}"
```

- Route matching only works when all segments are present, so if the user selects only maxId, what do we do about maxDate?
- We could give it a default value and update our logic to check for zero/null. Or we could add another route to the page that contained max id but not max date.
- Either of these would solve the issue, but what happens if we added another and another?

Setting query-string values

- In the SearchFilter component add a new field to store the date as well as a new label and HTML input to record it:

```
@inject NavigationManager NavManager

<div class="filters">
   <label for="maxLength">Max Length</label>
   <input id="maxLength" type="number" class="form-control" @bind="_maxId" />
   <label for="maxDate">Max Date</label>
   <input id="maxDate" type="date" class="form-control" @bind="_maxDate" />
   <button class="btn btn-outline-primary" @onclick="FilterSearchResults"> Filter </button>
   <button class="btn btn-outline-secondary" @onclick="ClearSearchFilter">Clear</button>
</div>

@code {
private int _maxId;

private DateTime? _maxDate;

(...)
```

The new HTML input is bound to the _maxDate field.

https://gist.github.com/brunobmo/7d398ec0ed7f82ee70dfbfc0708a3aa5

Setting query-string values

- With the ability to record the max date from the user, we can update the FilterSearchResults method to add our filters as query-string values instead of route parameters.
- We'll take advantage of the new query-string helpers introduced in .NET 6 to do this.

Setting query-string values

- In Searchfilter.razor, let's replace the FilterSearchResults to:

```
private void FilterSearchResults()
{
   var uriWithQuerystring = NavManager.GetUriWithQueryParameters(new Dictionary<string, object?>(){
     [nameof(SearchPage.MaxId)] = _maxId == 0 ? null : _maxId,
     [nameof(SearchPage.MaxDate)] = _maxDate
    });

    NavManager.NavigateTo(uriWithQuerystring);
}
```

Constructs a URI containing the key-value pairs provided as a query string

If the value of a key is null, the method will omit the entry from the query string.

Navigates to the URI with the query string

https://gist.github.com/brunobmo/d027870eddaeed8b3f0ad7f574c861ce

Setting query-string values

- The GetUriWithQueryParameters method takes a dictionary, and depending on the value of a key, it will include or omit that value from the query string.
- In the code, if either _maxId or _maxDate are 0/null, we don't want to include that entry on the query string.
- By setting their value to null, they will be ignored when the query string is built.

Setting query-string values

- We must do is add the new _maxDate field to the ClearSearchFilter method in the SearchFilter.razor:

```
private void ClearSearchFilter()
{
    _maxId = 0;
    _maxDate = null;
    NavManager.NavigateTo($"/search/{SearchTerm}");
}
```

Retrieving query-string values using SupplyParameterFromQuery

- Before we can run the app, we need to update the SearchPage.
- You'll notice that when we added the key-value pairs to the dictionary we passed into the GetUriWithQueryParameters method, we used nameof()to define the name of the keys—referencing two properties on the SearchPage
- We're going to use parameters decorated with a special attribute called SupplyParameterFromQuery.
- When we create a parameter on a component and add this attribute, Blazor will attempt to set the value of the property based on a query string with a matching name.
- This is why using the nameof() technique is so useful: it ensures that the key name used in the query string always matches that of the destination parameter.

Retrieving query-string values using SupplyParameterFromQuery

- In the SearchPage, change the parameters:

```
(...)
[Parameter, SupplyParameterFromQuery]
public int? MaxId { get; set; }

[Parameter, SupplyParameterFromQuery]
public DateTime? MaxDate { get; set; }
(...)
```

Retrieving query-string values using SupplyParameterFromQuery

- Now that we can receive the values from the query string, we can do something with them.
- We're going to add a new method, UpdateFilters, that will filter the search results based on the values of MaxId and MaxDate

Retrieving query-string values using SupplyParameterFromQuery

- Add the following method to the SearchPage:

```
private void UpdateFilters()
{
    var filters = new List<Func<Card, bool>>();
    if (MaxId is not null && MaxId > 0){
        filters.Add(x => x.Id <= MaxId);
    }
    if (MaxDate is not null){
        filters.Add(x => x.Date <= MaxDate);
    }
    if (filters.Any()){
      _searchResults = _cachedSearchResults.Where(card => filters.All(filter => filter(card)));
    }else{
      _searchResults = _cachedSearchResults;
    }
}
```

The filters variable will hold a list of lambda expressions based on which search filters are present.

If a max Id filter is defined, add the lambda to filter it to the filters list.

If a max date filter is defined, add the lambda to filter it to the filters list.

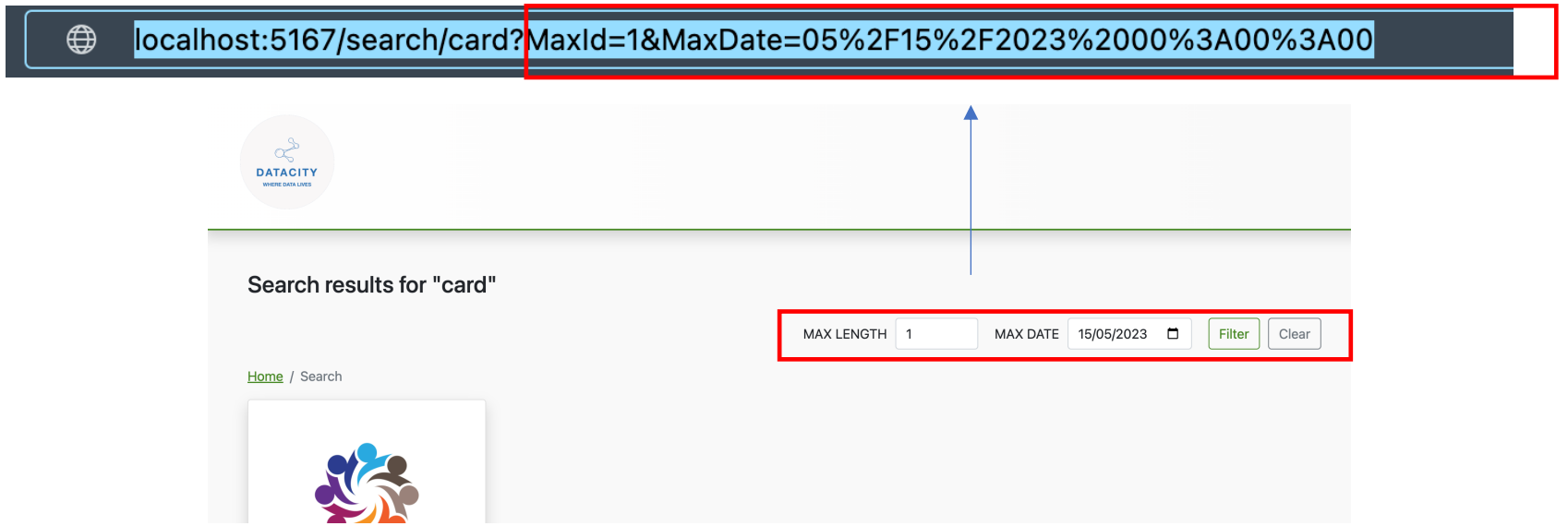
https://gist.github.com/brunobmo/60fb0cfdd94b02214f4595b4da14aaff

Retrieving query-string values using SupplyParameterFromQuery

- We now need to call the UpdateFilters method at the appropriate time.
- As query-string values are handled in the same way as any other parameters, when they change, the OnParametersSet life cycle method will be called.
- This means that we can update the current OnParametersSet implementation to just call the UpdateFilters method.
- In SearchPage change OnParametersSet to:

```
protected override void OnParametersSet() => UpdateFilters();
```

Retrieving query-string values using SupplyParameterFromQuery



Retrieving query-string values using SupplyParameterFromQuery

Challenge:

- If you copy the URL and open the same address in another tab, you'll notice the values in the search filter inputs are not being set to those in the query string, even though the results are showing correctly.
- How can we fix that?

Summary

- Navigation in Blazor is handled by a client-side router, which is a component.
- By default, the router component resides in the App component at the top of the app component tree.
- Pages in Blazor are just components that contain a special directive called @page

Summary

- It is possible to define multiple page directives on a single component and have it load for more than one route.
- The page directive requires a route to be defined, which the component will handle.
- Reflection is used by the router to find pages; these are then stored in a table in memory by the router.
- When a route is requested, the router looks up which component handles the requested route.
- If a match for the route isn't found, then the markup defined in the router's NotFound template is displayed.

Summary

- Developers can trigger navigation programmatically using the NavigationManager service.
- The NavigationManager class exposes an event called LocationChanged, which is triggered whenever a navigation occurs. This can be subscribed to by developers to run custom actions.
- Simple data, such as IDs, can be passed between pages via route parameters.

Blazor Server

Routing

2023