Blazor Server

Creating more reusable components

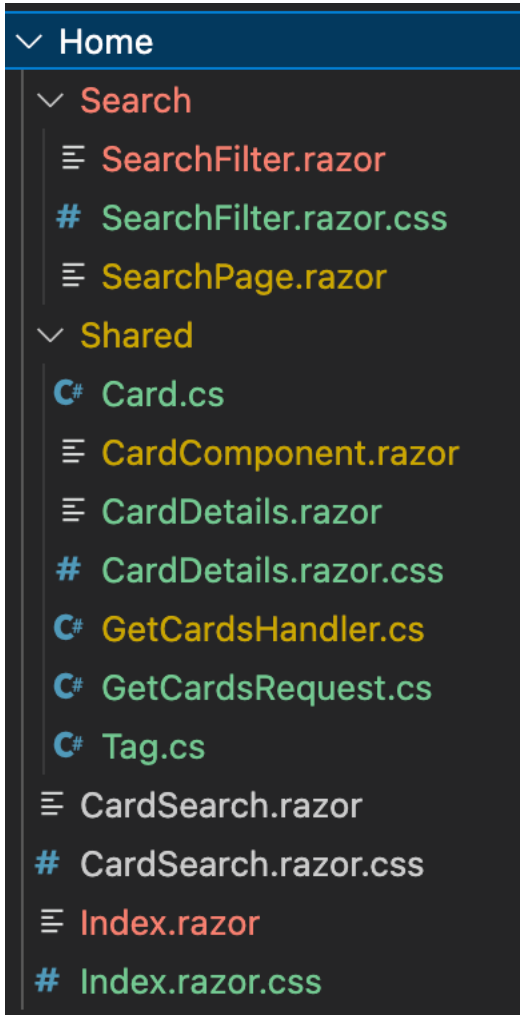
2023

Introduction

- The first step is to complete some parts of our website:
  - Return all cards to the Home Page
  - Create links to access the pages created before
  - “Wire things”

Refactor Home Folder Structure

- Currently, there are a lot of files in the Home feature. This isn't a problem if they all belong there—but that isn't the case here.
- By identifying subfeatures and moving them to their own folders, we can keep larger features organized and easy to navigate.



Returning all cards from Mongo database

- Currently, the Home and Search Page components load card data from a local file called card-data.json, which is in the Web project's wwwroot folder.
- We will start by creating a new folder: Shared inside Homefolder.
- Inside this folder, we will add a new class called GetCardRequest.cs.

Refactor Home Folder Structure

- By adding those subfolders, we've introduced extra namespaces.
- This is because, by default, Blazor components will use the folder structure they reside in to generate their namespace.
- This will cause some build errors. So, we're going to add a new component called _Imports.razor to the root of the Home folder with the following line:

```
@using app.Features.Home.Shared
```

GetCardsRequest

```
public record GetCardsRequest() : IRequest<GetCardsRequest.Response>{
   public static readonly string RouteTemplate = "find";
   public record Response(IEnumerable<Card> cards);

   public record Card(
     string _Id,
     int Id,
     string Name,
     string Description,
     string? Image,
     DateTime? Date,
     int TimeInMinutes,
     List<Tag> Tags
  );

  public record Tag(int Id, string Name);
}
```

It is similar to other requests but we refactored the set of class variables. Notice the “static readonly”.

We are changing some things to put the connection details in the appsettings.json file.

https://gist.github.com/brunobmo/e18e1d1aa437bdfff49b5480f6e15f42

Adding the GetCardsHandler class

```
public class GetCardsHandler : IRequestHandler<GetCardsRequest, GetCardsRequest.Response>{
  private readonly HttpClient _httpClient;
  private IConfigurationRoot config;

  public GetCardsHandler(HttpClient httpClient){
    _httpClient = httpClient;
    var configuration = new ConfigurationBuilder().AddJsonFile($"appsettings.json");
    config = configuration.Build();
  }

  public async Task<GetCardsRequest.Response> Handle(GetCardsRequest request,CancellationToken cancellationToken){
    var requestMessage = new HttpRequestMessage(HttpMethod.Post,config.GetValue<string>("MongoConnectionString:BaseURL")
+ GetCardsRequest.RouteTemplate);
  (...)
  var mongoJson = config.GetValue<string>("MongoConnectionString:RequestStructure");
  mongoJson = mongoJson.Replace("{body}", "");
(...)
```

The IConfiguration interface is used to read Settings and Connection Strings from AppSettings.

A template string is used to reuse code

https://gist.github.com/brunobmo/3b5ca2ffe6f09e7646ed14e2e3cf9219

Adding the GetCardsHandler class

- Appsettings.json

```
{
  (...)
  "MongoConnectionString": {
    "BaseURL": "https://data.mongodb-api.com/app/data-docuz/endpoint/data/beta/action/",
    (...)
    "RequestStructure": "{\"collection\": \"{collection}\",\"database\":
\"{database}\",\"dataSource\": \"{dataSource}\" {body}}"
  }
(...)
}
```

Updating HomePage.razor and SearchPage.razor

- With the GetCardsHandler in place, we can update the Home and Search pages to get cards from the API.
- To do this, we'll update them to dispatch a GetCardsRequest via MediatR.

Updating HomePage.razor and SearchPage.razor

- Let's update Index page:

```
(...)
@if(_cards.Any()){
  <div class="mb-4">
    <p class="font-italic text-center">Do you have an awesome card you'd like to share? <a href="add-
card">Add it here</a>.</p>
  </div>
  <div class="grid">
    @foreach (var card in _cards){
      <CardComponent card="card" OnSelected="HandleCardSelected" />
    }
  </div>
}else{
  <div class="no-cards">
    <h3 class="text-muted font-weight-light"> We currently don't have any cards,
      <a href="add-card">why not add one?</a>
    </h3>
  </div>
}
```

https://gist.github.com/brunobmo/896d8cc9f497c66ea3b2f9722a2edd22

Updating HomePage.razor and SearchPage.razor

- To finish things off, we will add a couple of styles to the home page (Index.razor.css).

```
.no-cards {
  text-align: center;
  margin-top: 100px;
}
.no-cards h3 {
  font-size: 1.3rem;
  font-weight: 200;
  color: #000007;
}
```

https://gist.github.com/brunobmo/adaf8d8d1ebf58b2acf43d78765b79b7

Exercises

- Change Search page to load data from Mongo database
- Change Card component and CardDetails to show images correctly
- Change Requests and Handlers according to what we did in GetCardsHandler.

Defining templates

- Templates are a powerful tool when building reusable components.
- They allow us to specify chunks of markup to be provided by the consumer, which we can then output wherever we wish.
- We have already used some basic templating when we built the FormSection and FormFieldSet components in the previous chapters.
- In those components, we defined a parameter with a type of RenderFragment and a name of ChildContent:

```
[Parameter]
public RenderFragment ChildContent { get; set; }
```

Defining templates

- We'll be enhancing the Index page with a component that allows the user to toggle the layout between a grid and a table
- The ViewSwitcher component allows the user to toggle between a card view and a table view of the available cards.
- To make this component as reusable as possible, we don't want to hardcode the markup for either the grid or the table view.
- Instead, we want to define these as templates that allow the consumer of the component to define these areas for themselves

Defining templates

- Let's look at the initial markup for the ViewSwitcher component (Under Features > Home > Shared)

```
(...)
<div class="btn-group">
   <button @onclick="@(() => _mode = ViewMode.Grid)" title="Grid View" type="button" class="btn
@(_mode == ViewMode.Grid ? "btn-secondary" : "btn-outline-secondary")">
     <i class="bi bi-grid-fill"></i>
   </button>
   <button @onclick="@(() => _mode = ViewMode.Table)" (…)</button>
   </div>
</div>
@if (_mode == ViewMode.Grid){
  @GridTemplate
}
else if (_mode == ViewMode.Table){
  @TableTemplate
}
(...)
```

Defining templates

- Let's look at the code for the ViewSwitcher component (Under Features > Home > Shared)

```
@code {
private ViewMode _mode = ViewMode.Grid;

[Parameter, EditorRequired]
public RenderFragment GridTemplate { get; set; } = default!;

[Parameter, EditorRequired]
public RenderFragment TableTemplate { get; set; } = default!;

private enum ViewMode { Grid, Table }
}
```

https://gist.github.com/brunobmo/b6d046ded8f02dfb1d0b7dcb5f54cde0

Defining templates

- We're also going to add some styling for the component.
- We'll add a new file called ViewSwitcher.razor.css and add the following code.

```
.grid {
    display: grid;
    grid-template-columns: repeat(3, 288px);
    grid-column-gap: 123px;
    grid-row-gap: 75px;
}
table {
    width: 100%;
    margin-bottom: 1rem;
    color: #212529;
    border-collapse: collapse;
}
(…)
```

https://gist.github.com/brunobmo/468d2e3872cbe44cd407878143d94239

Defining templates

- What is all we need for now.
- Let’s jump over to Index.razor and implement ViewSwitcher.
- We’re going to replace the current code that renders the grid of cards

Defining templates

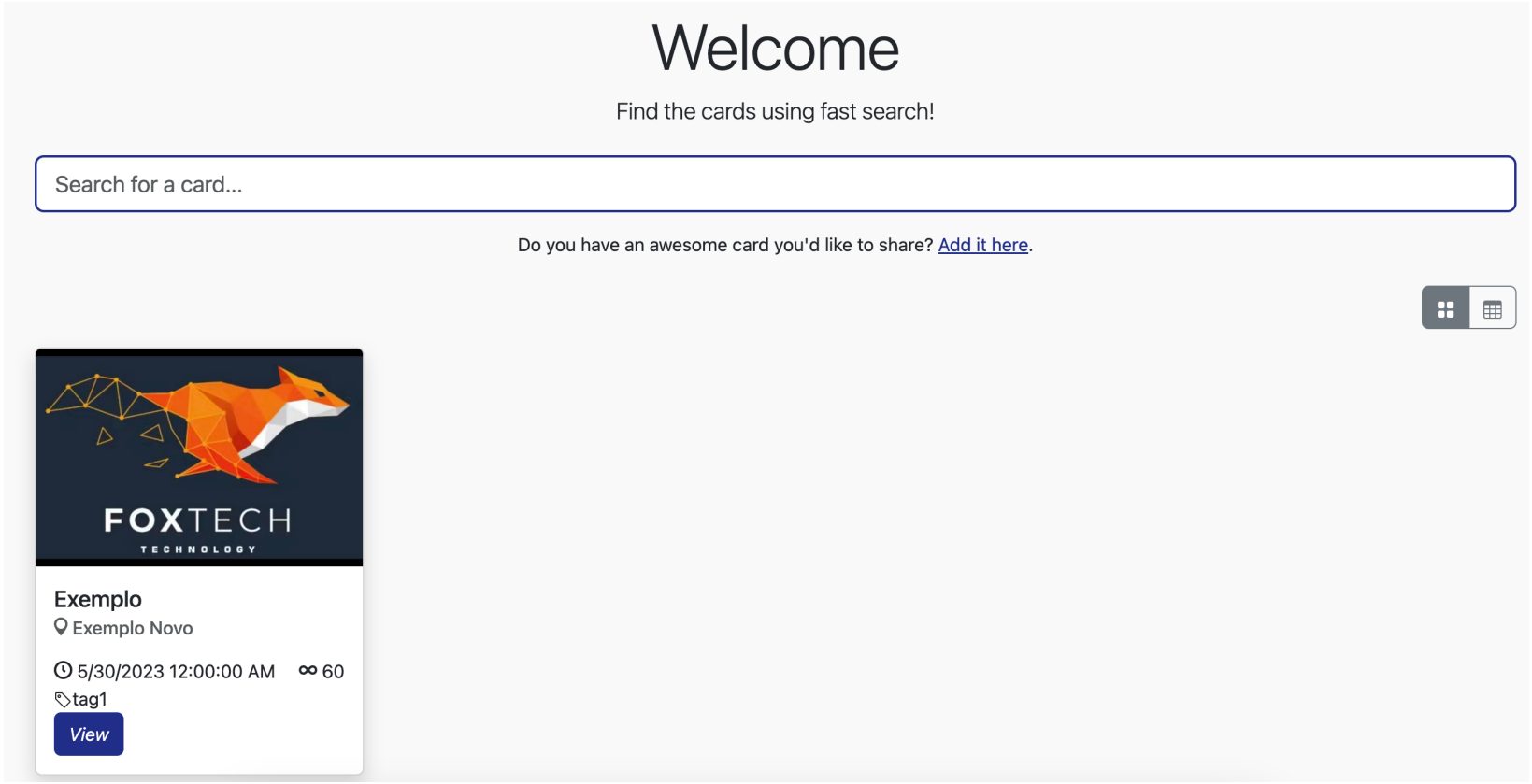
```
<ViewSwitcher>
  <GridTemplate>
     <div class="grid">
      @foreach (var card in _cards){<CardComponent card="card" OnSelected="HandleCardSelected" />}
     </div>
  </GridTemplate>
  <TableTemplate>
    <tbody>
      @foreach (var card in _cards){
      <tr>
         <th scope="col">@card.Name</th>
         (...)
        <button @onclick="@(() => HandleCardSelected(card))" title="View" class="btn btn-primary">
           <i class="bi bi-binoculars"></i>
        </button>
        <button @onclick="@(() => NavigationManager.NavigateTo($"/edit-card/{card._Id}"))"
title="Edit"} (...)

     </tbody>
   </table>
  </TableTemplate>
</ViewSwitcher>
```

https://gist.github.com/brunobmo/6132549638746e3b612a993934ca4a89

Defining templates

- To specify the markup for a particular template, we define child elements that match the name of the parameter.
- In our case, that is GridTemplate and TableTemplate.
- The markup we've defined above for the GridTemplate and TableTemplate will be output by ViewSwitcher where we specified the @GridTemplate and @TableTemplate expressions.

Defining templates



Enhancing templates with generics

- Currently, our component allows us to define the markup for the table and grid views and for the user to toggle between them.
- Right now, we must define a lot of markup in the Index when we're using the component.
- We're defining a div with a class of .grid around a foreach block in the grid template.
- Then for the table template, we're providing the entire markup for the table

Enhancing templates with generics

- As we know we're going to be displaying a grid or a table, we can bake some of the boilerplate markup into the component.
- Then when we use the component, we only have to specify the markup and data specific to that usage.
- To do this, we will introduce generics into our ViewSwitcher component.

Enhancing templates with generics

```
@typeparam TItem
(...)
@if (_mode == ViewMode.Grid){
  <div class="grid">
  @foreach (var item in Items){
    @GridTemplate(item)
  }
  </div>
}
else if (_mode == ViewMode.Table){
  <table><thead><tr>@HeaderTemplate</tr></thead>
    <tbody>
     @foreach (var item in Items){
       <tr>@RowTemplate(item)</tr>
     }
    </tbody>
  </table>
}
```

A type parameter is specified using the typeparam directive

We now only require the header cells to be specified when using the component, rather than all the markup for the head of the table.

Defining RenderFragments with a type parameter allows the consumer to use properties of that type when defining a template.

Enhancing templates with generics

```razor
@code {
    private ViewMode _mode = ViewMode.Grid;

    [Parameter, EditorRequired]
    public IEnumerable<TItem> Items { get; set; } = default!;
    [Parameter, EditorRequired]
    public RenderFragment<TItem> GridTemplate { get; set; } = default!;
    [Parameter, EditorRequired]
    public RenderFragment HeaderTemplate { get; set; } = default!;
    [Parameter, EditorRequired]
    public RenderFragment<TItem> RowTemplate { get; set; } = default!;

    private enum ViewMode { Grid, Table }

}
```

The component now accepts a list of items to be displayed.

https://gist.github.com/brunobmo/86541a0a88a96701e52035a0daca6520

Enhancing templates with generics

- Let's change Index.razor

```
<ViewSwitcher Items="_cards" Context="card">
  <GridTemplate>
   <CardComponent card="card" OnSelected="HandleCardSelected" />
  </GridTemplate>
  <HeaderTemplate>
   <th>Name</th>
   (...)
   </HeaderTemplate>
   <RowTemplate>
     <th scope="col">@card.Name</th>
     (...)
      <button @onclick="@(() => HandleCardSelected(card))" title="View" class="btn btn-primary">
(...)</button>
     <button @onclick="@(() => NavigationManager.NavigateTo($"/edit-card/{card._Id}"))"
title="Edit"
class="btn btn-outline-secondary">(...)</button>
     (...)
   </RowTemplate>
</ViewSwitcher>
```

The context parameter is renamed at the component level.

The list of cards is now passed into the ViewSwitcher rather than having to define foreach loops in the templates.

The GridTemplate is now cleaner, as we no longer need to define the grid and a foreach loop.

In the template that uses RenderFragment<T>, we can now access properties of the object through a variable called context. This allows loads of flexibility when building our markup.

The header template allows us to define the columns our table needs, but without all the boilerplate we had before.

https://gist.github.com/brunobmo/64b980a533ab302b3d72e22ec631d2cf

Sharing components with Razor class libraries

- We’re going to cover how we can share components between applications.
- It could be that you’re looking to build up a library of common components you use across all your applications.
- The way to share Blazor components is via a Razor class library (RCL)

Sharing components with Razor class libraries

- Let’s add a new RCL to our applications so we can learn how they work.
- Using the command line, navigate to the folder that contains the solution file.
- Then run the following commands:
  - dotnet new razorclasslib -o app.ComponentLibrary
  - dotnet sln add app.ComponentLibrary\app.ComponentLibrary.csproj
  - dotnet add app\app.csproj reference app.ComponentLibrary\app.ComponentLibrary.csproj

Sharing components with Razor class libraries

- As with most project types in .NET, the RCL comes with some example files.
- We don’t need these, so you can delete them—except _Imports.razor.
- It also comes with a directory called wwwroot.
- If we had any static assets to ship with our RCL, such as images, CSS, or JavaScript files, we could add them here and reference them in the host project.
- But as we don’t, go ahead and delete it as well.
- The RCL should now contain just the _Imports.razor file.

Sharing components with Razor class libraries

- Now that our library is ready, we can move the ViewSwitcher.razor and ViewSwitcher.razor.css files over from the blazor server Project;
- We can now switch back to the blazor server project and add a new using statement to the main _Imports.razor in the root of the project. Add the following line to the file:
  - @using app.ComponentLibrary

Exercises

- Show only cards that can be available considering TimeInMinutes, the amount of minutes the card will be available.
- Allow Card edit in grid view using a specific button (use bootstrap buttons).

Summary

- Templates are defined by creating parameters with a type of RenderFragment.
- Defining a template with the name ChildContent will capture all the markup entered between the start and end tags of a component.
- Templates can be generically typed.

Summary

- Generically typed templates require an object of type T to be passed into them when they are invoked.
- When providing markup for a template that is generically typed, the properties of T are available to be used in the markup via a parameter called context.
- The context parameter can be renamed to aid readability via the Context attribute.

Summary

- The context parameter can be renamed on a specific template or at the component level. If done at the component level, all generic templates in the component inherit the new name.
- Components are shared using a Razor class library.
- Razor class libraries can be packaged and shipped via NuGet, as with other .NET libraries.
- When using scoped CSS in an RCL, it’s automatically bundled and included in a host Blazor application.

Blazor Server

Creating more reusable components

2023