Blazor Server

Forms and Validation – Beyond the basics

2023

Customizing validation CSS classes

- CSS frameworks such as Bootstrap (https://getbootstrap.com), Materialize (https://materializecss.com), or Bulma (https://bulma.io) all have predefined classes for valid and invalid input states.
- Blazor allows us to use these classes—instead of the default ones it provides—by specifying them in a custom FieldCssClassProvider.

Creating a FieldCssClassProvider

Create the: BootstrapCssClassProvider.cs file:

```
using Microsoft.AspNetCore.Components.Forms;

namespace app.Validation;
public class BootstrapCssClassProvider : FieldCssClassProvider
{
  public override string GetFieldCssClass(EditContext editContext, in FieldIdentifier fieldIdentifier){
    var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();
    if (editContext.IsModified(fieldIdentifier)){
      return isValid ? " is-valid" : "is-invalid";
    }
    return isValid ? "" : "is-invalid";
  }
}
```

https://gist.github.com/brunobmo/f541401900fcb7b9a328bf584c825910

Customizing validation CSS classes

- When deriving from FieldCssClassProvider, we need to override the GetFieldCssClass method.
- This method takes an EditContext and a FieldIdentifier that represents the field in the form we're getting CSS classes for.
- EditContext is the brain of the form and keeps track of the state of each field in the form.
- We can use the GetValidationMessages method on the EditContext to check if there are any validation messages for the current field.
- If there are, then we know the field is currently not valid and we can set the isValid variable accordingly, or vice versa

Customizing validation CSS classes

- Next, we can use the IsModified method on the EditContext to check if the field has been edited by the user in any way.
- For a field to be modified, the user must have typed something or changed a selection.
- Even typing into an empty field and then removing all the characters returning it to its originally empty state would class the field as modified.

Using custom FieldCssClassProviders with EditForm

- To use BootstrapCssClassProvider with the EditForm component, we need to plug it in. To do this, we use the EditContext.
- Internally, the EditForm creates an EditContext instance using that model.
- However, we can create an EditContext ourselves and pass that to the EditForm component instead of the model

Using custom FieldCssClassProviders with EditForm

We pass the EditContext instance we create to the EditForm rather than passing it to the model directly.

- Let's change AddCard.razor:

```
<EditForm EditContext="_editContext" OnValidSubmit="SubmitForm">
(...)

@code {
    private EditContext _editContext = default!;
(...)

protected override void OnInitialized()
{
  _editContext = new EditContext(_card);
  _editContext.SetFieldCssClassProvider(new BootstrapCssClassProvider());
}
(...)
```

Shows the new private field for our EditContext instance

Configures the EditContext to use our new BootstrapCssClassProvider

https://gist.github.com/brunobmo/7983b1679c4e1bb1a4d96f16f455b5aa

Using custom FieldCssClassProviders with EditForm

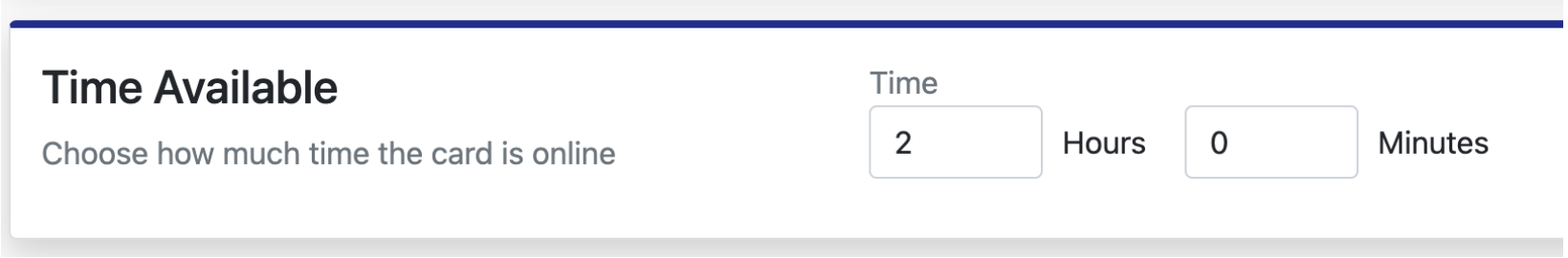
- Before we move on, we just need to tidy up our CSS.
- We will remove the following classes from wwwroot > css:

```
input.invalid,
textarea.invalid,
select.invalid {
  border-color: red;
}

input.valid.modified,
text.valid.modified,
select.valid.modified {
   border-color: green;
}
```

Building custom input components with InputBase

- Let’s create a new feature to the cards, allowing the user to define how much time a card will appear in the home page (like a “flash sale”);
- This is a great opportunity for a custom input component:



Building custom input components with InputBase

- The Blazor team has included a base type: InputBase<T>
- All we need to do is provide the UI and implementation for a method called TryParseValueFromString

Inheriting from InputBase<T>

- The first thing we need to do is create a new component in our ManageCards feature called InputTime.razor.

```
@inherits InputBase<int>

<div class="input-time">
  <div>
    <input class="form-control" type="number" min="0" @onchange="SetHourValue" value="@_hours" />
    <label>Hours</label>
  </div>
  <div>
     <input class="form-control" type="number" min="0" max="59" @onchange="SetMinuteValue" value="@_minutes" />
     <label>Minutes</label>
  </div>
</div>

@code {
  private int _hours;
  private int _minutes;
  protected override bool TryParseValueFromString(string? value, out int result, out string
validationErrorMessage)=> throw new NotImplementedException();
}
```

The inherits directive allows us to specify InputBase<T> as a base class for our component. The type parameter must match the type of the form model property the component will bind to.

When using InputBase<T>, we must provide an implementation for the TryParseValueFromString method. However, with our design, this method won't be called.

https://gist.github.com/brunobmo/f2ce50fd52db626485c5dc96517f81f0

Inheriting from InputBase<T>

- The first thing we need to do is create a new component in our ManageCards feature called InputTime.razor.
- In the code block, we have provided an implementation for the TryParseValueFromString method.
- This method must be implemented by any component derived from InputBase<T>.
- Its job is to convert a string value to the type that the component is bound to on the form model.
- However, depending on how you build a custom input component, this method may not ever get called.

Inheriting from InputBase<T>

- The base class provides two properties to update the model value:
  - CurrentValueAsString
  - CurrentValue
- We could bind that input directly to the CurrentValueAsString property.
- We'd need to use this property, as HTML inputs only work with string values.
- When a value was entered in the input, it would set CurrentValueAsString, Blazor call TryParseValueFromString method

Inheriting from InputBase<T>

- Let's change InputTime.razor:

```
@inherits InputBase<int>
<div class="input-time">
  <div>
    <input class="form-control" type="number" min="0" @onchange="SetHourValue" value="@_hours" />
    <label>Hours</label>
  </div>
  <div>
    <input class="form-control" type="number" min="0" max="59" @onchange="SetMinuteValue" value="@_minutes" />
    <label>Minutes</label>
  </div>
</div>
```

Inheriting from InputBase<T>

- Let's change InputTime.razor:

```
@code {
  private int _hours;
  private int _minutes;
  (...)
  private void SetHourValue(ChangeEventArgs args){
    int.TryParse(args.Value?.ToString(), out _hours);
    SetCurrentValue();
  }
  private void SetMinuteValue(ChangeEventArgs args){
    int.TryParse(args.Value?.ToString(), out _minutes);
    SetCurrentValue();
  }
  private void SetCurrentValue() => CurrentValue = (_hours * 60) + _minutes;

protected override void OnParametersSet(){
    if (CurrentValue > 0){
      _hours = CurrentValue / 60;
      _minutes = CurrentValue % 60;
    }
  }
}
```

Using the ChangeEventArgs, this method extracts the new value entered by the user and converts it to an integer and sets the _hours field. It then calls SetCurrentValue.

The _hours and _minutes fields are converted to a total minutes value, and then the CurrentValue property is set to that value.

Loading existing values

https://gist.github.com/brunobmo/b1081f7cde598bfa5319569d3a52656a

Inheriting from InputBase<T>

- Until now, when we've bound to HTML inputs, we've used the bind directive.
- However, that method wouldn't be optimal in this scenario.
- We need to perform some actions every time either the hour or minute values change.
- While we could use the bind directive with a property and do the work inside the setter method, that would still require us to have a private backing field.

Inheriting from InputBase<T>

- Note that CurrentValue property comes from the base class.
- By setting this property, all the logic for triggering validation and updating the model value will be run
- The final piece to the component involves loading an existing value—for example, when the component is being used to edit an existing record.
- For this we will use the OnParametersSet life cycle method

Styling the custom component

- We will add a new CSS file into the Features > ManageCard folder called InputTime.razor.css

```css
.input-time {
  display: flex;
}
.input-time div {
  display: flex;
  align-items: center;
  margin-right: 20px;
}
.input-time div input {
  width: 90px;
  margin-right: 10px;
}
.input-time div label {
  margin-bottom: 0;
}
```

https://gist.github.com/brunobmo/a01648686a504ea2fc96db40f9dd8d77

Styling the custom component

- The last piece of styling we need to configure is for validation.
- Another nice feature of using InputBase is that it provides us with a property called CssClass that outputs the correct validation classes based on our field's state.
- We're going to reference the CssClass property in the class attribute on both of our input elements:

```
<input class="form-control @CssClass" type="number" min="0" @onchange="SetHourValue"
value="@_hours" />
```

the quotation mark is necessary in this case to indicate that the value of the CssClass variable should be treated as a string when binding it to the "class" attribute.

Using the custom input component

- Let’s add a new section to the form (AddCard.razor):

```
(...)
<FormSection Title="Time Available" HelpText="Choose how much time the card is online">
  <FormFieldSet Width="col-5">
    <label for="cardTime" class="font-weight-bold text-secondary">Time</label>
    <InputTime @bind-Value="_card.TimeInMinutes" id="cardTime" />
    <ValidationMessage For="@(() => _card.TimeInMinutes)" />
  </FormFieldSet>
</FormSection>
(...)
```

https://gist.github.com/brunobmo/aee9e14ee35af35c083b68a50f094291

Using the custom input component

- Let's update CardValidator class:

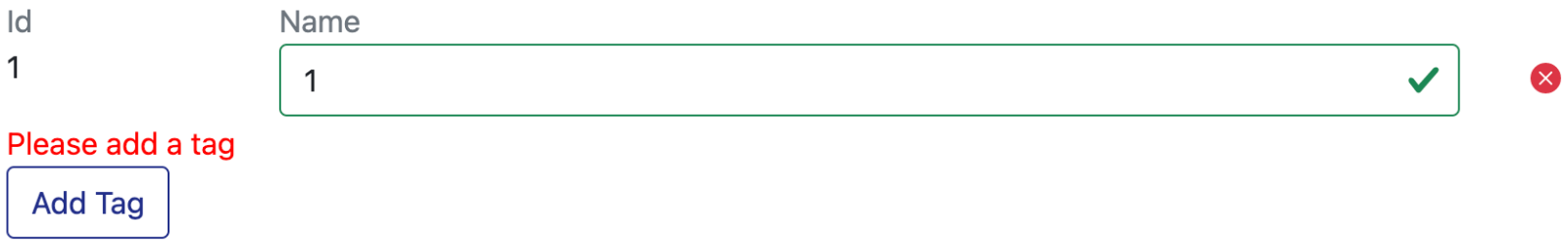
```
RuleFor(x => x.TimeInMinutes).GreaterThan(0).WithMessage("Please enter a time");
```

- And Card class with a new field:

```
public int TimeInMinutes { get; set; }
```

Exercise

- Test everything
- If we click submit a message: “Please add a tag” message appear. It doesn’t disappear when we insert a tag



Working with files

- We’re going to start by adding the InputFile component to our form:

```
<FormFieldSet Width="col-6">
  <label for="cardImage" class="font-weight-bold text-secondary">Image</label>
  <InputFile OnChange="LoadCardImage" class="form-control-file" id=”cardImage" accept=".png,.jpg,.jpeg" />
</FormFieldSet>
```

https://gist.github.com/brunobmo/a9bc976bbba305253e404ece280dc460

Working with files

- The most important point to notice is that the InputFile component doesn't use the bind directive as the other input components do.
- Instead, we must handle the OnChange event it exposes.
- Just as with file uploading in regular HTML forms, we can provide a list of file types we want the user to be able to upload using the accept attribute.
- Under the hood, the InputFile component renders an HTML input element with a type of file. The accept attribute is passed down to this element when the component renders.

Working with files

- Now that we have the InputFile component in place, we need to add the LoadCardImage method to the code block.

```
(...)
private IBrowserFile? _cardImage;
(...)
private void LoadCardImage(InputFileChangeEventArgs e) => _cardImage = e.File;
```

Uploading files when the form is submitted

- We’re going to extend this logic to check if an image has been selected and make an additional call to upload it.
- Starting with the SubmitForm method, we’re going to update the existing code

Uploading files when the form is submitted

```
private async Task SubmitForm(){
    var response = await Mediator.Send(new AddCardRequest(_card));
    if (response.insertedId == ""){
      _errorMessage = "There was a problem saving your card.";
      _submitSuccessful = false;
      return;
    }

    if (_cardImage is null){
      _submitSuccessful = true;
      ResetForm();
      return;
      var newImageName = await ProcessImage(response.insertedId, _cardImage);
}
```

Mongo Request for store Card

Checks if a card image has been selected

If no image is selected, reset the form.

Call ProcessImage method passes in the card ID returned from the previous API call.

https://gist.github.com/brunobmo/1d985cf1cdf1c6e35cf677df33962dc4

Uploading files when the form is submitted

- If a card has been selected, then we call the ProcessImage method.
- This method takes the card ID returned from the AddCardRequest.

```
private async Task<string> ProcessImage(string cardId, IBrowserFile? _cardImage)
    string imageUploadResponse = await UploadImage.HandleUpload(cardId, _cardImage);
    if (string.IsNullOrWhiteSpace(imageUploadResponse)){
      _errorMessage = "Your card was saved, but there was a problem uploading the image.";
      return;
    }

    _submitSuccessful = true;
    ResetForm();
    return imageUploadResponse;

}
```

https://gist.github.com/brunobmo/ccc6b44fa53d3ed1527e4e2396a4a0ac

Building the upload logic

- The final piece to the add is handling upload.
- This will go under Features > ManageCards.
- First, we will add a package: dotnet add package SixLabors.ImageSharp --version 3.0.1
- We'll use this package to resize the uploaded image to the correct dimensions for our app.

Building the upload logic

- Second, we'll use a folder images in the wwwroot of the API project.
- This is where we'll store all the cards images that are uploaded.

Building the upload logic

- Let's create UploadCardImage.cs to handle upload logic:

```
public class UploadImage {
   public static async Task<string> HandleUpload(string cardId, IBrowserFile? file, CancellationToken
cancellationToken = default){
     if (file.Size == 0) {
       return "No image found.";
     }
     var filename = $"{Guid.NewGuid()}.jpg";
     var saveLocation = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot/images", filename);
     var resizeOptions = new ResizeOptions {
        Mode = ResizeMode.Pad,
        Size = new Size(640, 426)
     };
     var image = await Image.LoadAsync(file.OpenReadStream());
     image.Mutate(x => x.Resize(resizeOptions));
     await image.SaveAsJpegAsync(saveLocation, cancellationToken: cancellationToken);
     return filename;
   }
}
```

Creates a new filename for the uploaded image that is safe to use in the application

Specifies the save location for the file

Using ImageSharp, resize the uploaded image to the correct dimensions and save it to the filesystem

https://gist.github.com/brunobmo/fea29f8d148396eb6492c3514ddc8d03

Building the upload logic

- Now we need to update Mongo Database to update the image with the generated name;
- Let's create UpdateCardImageRequest:

```
public record UploadCardImageRequest(string cardId, string image) : IRequest<UploadCardImageRequest.Response>
{
   public const string RouteTemplate = "https://data.mongodb-api.com/app/data-
docuz/endpoint/data/beta/action/updateOne";
   (...)
   public record Response{
      public int modifiedCount { get; set; }
      public Response(int modifiedCount){
         this.modifiedCount = modifiedCount;
      }
}
```

https://gist.github.com/brunobmo/9369891ce1084bcd1b0c3700aab622a9

Building the upload logic

- Thus, we will create the UpdateCardImageRequestHandler for handling the request to MongoDB to update image url after the new name generation:

```
public class UpdateCardImageHandler : IRequestHandler<UploadCardImageRequest, UploadCardImageRequest.Response>{
(...)
  public async Task<UploadCardImageRequest.Response> Handle(UploadCardImageRequest request,CancellationToken
cancellationToken){
     var requestMessage = new HttpRequestMessage(HttpMethod.Post, UploadCardImageRequest.RouteTemplate);
requestMessage.Headers.Add("api-key", AddCardRequest.ApiKey);

     var mongoJson =
        "{\"c ollection\":\""
        + AddCardRequest.Collection
        + "\","
        + "\"database\":\""
        + AddCardRequest.Database
        + "\",\"dataSource\":\""
        + AddCardRequest.DataSource
        + "\",\"filter\": { \"_id\": { \"$oid\": \""+ request.cardId +"\" } },"
        + "\"update\": { \"$set\": { \"Image\":\""+ request.image +"\""
        + "}}}";

(...)
```

https://gist.github.com/brunobmo/858ae17a9efbb4304eeeaaab1e716d37

Building the upload logic

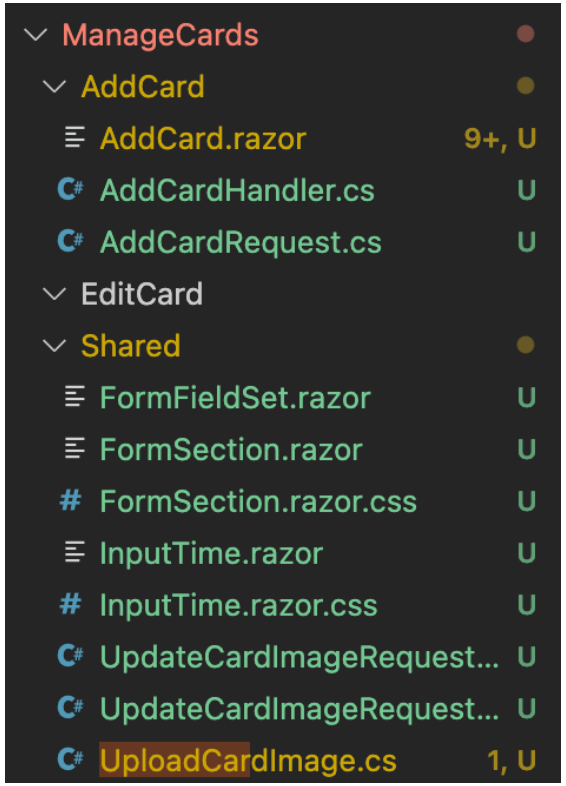
- Exercises:
  - Update AddCard.razor to include this logic, i.e. insert the record and then update it with new image name
  - Put image upload mandatory
  - Create a loading message after form being submitted

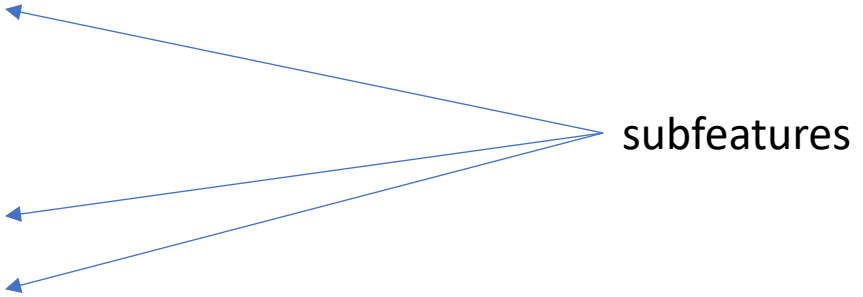
Updating the form to allow editing

- The final piece of work we're going to do is refactor our form so it can handle both adding and editing of cards.
- To do this, we'll extract the form from AddCard.razor and make it into a standalone component.
- We can share it with AddCard.razor and a new page we'll add called EditCard.razor.

Updating the form to allow editing

- First, let's reorganize our ManageCards folder structure (fix errors this change generate):





Separating the card form into a standalone component

- The first task we'll tackle is separating out the card form from the AddCard and making it into its own component, capable of handling both adding and editing.
- We will refactor the code inside AddCard, creating a new component that allow the encapsulation of the form behaviour.
- Add the CardForm.razor

```
<EditForm EditContext="_editContext" OnValidSubmit="SubmitForm">
(...)
</EditForm>
```

Separating the card form into a standalone component

- Add the CardForm.razor

```
@code {
  private EditContext _editContext = default!;
  private Card _card = new Card();
  private IBrowserFile? _cardImage;
  private Boolean activeSpinner = false;
  [Parameter]
  public Func<Card, IBrowserFile?, Task> OnSubmit { get; set; }

  protected override void OnInitialized(){(...)}
  public void ResetForm(){
   _card = new Card();
   _cardImage = null;
  }

  private void LoadCardImage(InputFileChangeEventArgs e){(...)}

  private async Task SubmitForm(){
    await OnSubmit(_card, _cardImage);
}
```

The OnSubmit parameter defines an event (delegate) that passes the data entered in the form to the handler specified by the consuming component.

Note that the ResetForm method is public. This will be called by the consuming component to reset the form, if required.

The handler for the EditForm's OnValidSubmit event will invoke the CardForm's OnSubmit event. This allows the handler to decide how to persist the data from the form.

Separating the card form into a standalone component

- We've added a component parameter that defines a component event—On-Submit.
- When the EditForm's OnValidSubmit event is invoked, the SubmitForm method is run.
- This, in turn, calls the OnSubmit event passing in the card data from the form, as well as the image, if one has been selected.
- It's worth noting here that we're not using the EventCallback<T>. This is because we want to manually control when StateHasChanged is called in the handler.

Refactoring AddCarPage.razor

- We will update the markup section of the AddCard with the following code:

```
@inject IMediator Mediator
@using app.Features.ManageCards.Shared;

@page "/add-card"

<PageTitle>Add Card</PageTitle>

(...)
@if (_submitSuccessful){
    (...)
}
else if (_errorMessage is not null){
    (...)
}

<CardForm @ref="_cardForm" OnSubmit="SubmitNewCard" />
```

The original EditForm component is replaced with the new CardForm component. Blazor's @ref directive is used to capture a reference to the component. This will be used to invoke the ResetForm method. We also provide a OnSubmit handler.

Refactoring AddTrailPage.razor

- Before we jump into the code block, we have an opportunity to extract some reusable code out into components.
- The markup that shows the success and error alerts will also be needed on the edit card page when we add it a little later.
- This will save us from duplicating a fair amount of code, and if we ever want to update our success and error alerts, we can do it in a single place.

Refactoring AddTrailPage.razor

- Create a new component in the Shared folder called SuccessAlert.razor.

```
<div class="alert alert-success" role="alert">
    <svg xmlns="http://www.w3.org/2000/svg" width="18" height="18"
      fill="currentColor" class="bi bi-check-circle-fill" viewBox="0 0 16 16">
    <path fill-rule="evenodd" d="M16 8A8 8 0 1 1 0 8a8 8 0 0 1 16
 0zm-3.97-3.03a.75.75 0 0 0-1.08.022L7.477 9.417 5.384 7.323a.75.75 0 0
 0-1.06 1.06L6.97 11.03a.75.75 0 0 0 1.079-.02l3.992-4.99a.75.75 0 0
 0-.01-1.05z" />
    </svg>
    @Message
</div>
@code {
    [Parameter, EditorRequired]
    public string Message { get; set; } = default!;
}
```

https://gist.github.com/brunobmo/cabaa224ae080f7ba72e8f97e53e70e9

Refactoring AddTrailPage.razor

- We'll add a new component called ErrorAlert.razor and add the following code

```
<div class="alert alert-danger" role="alert">
  <svg xmlns="http://www.w3.org/2000/svg" width="18" height="18"
    fill="currentColor" class="bi bi-x-circle-fill" viewBox="0 0 16 16">
    <path fill-rule="evenodd" d="M16 8A8 8 0 1 1 0 8a8 8 0 0 1 16
0zM5.354 4.646a.5.5 0 1 0-.708.708L7.293 8l-2.647 2.646a.5.5 0 0 0
.708.708L8 8.707l2.646 2.647a.5.5 0 0 0 .708-.708L8.707 8l2.647-2.646a.5.5
0 0 0-.708-.708L8 7.293 5.354 4.646z" />
  </svg>
  @Message
</div>
@code {
  [Parameter, EditorRequired]
  public string Message { get; set; } = default!;
}
```

https://gist.github.com/brunobmo/a0dd89dc41b235c3b09368ee94cd3692

Refactoring AddTrailPage.razor

- We can now update the AddCard to use the new components:

```
(...)
@if (_submitSuccessful)
{
<SuccessAlert Message="Your card has been added successfully!" />
}
else if (_errorMessage is not null)
{
<ErrorAlert Message="@_errorMessage" />
}
(...)
```

https://gist.github.com/brunobmo/6b90af8cbf88f20a1c90aef4324bbd31

Refactoring AddTrailPage.razor

- With that small refactor out of the way, we can get back to updating the code block for AddCard.razor.

```
@code {
    private bool _submitSuccessful;
    private string? _errorMessage;
    private CardForm _cardForm = default!;
    private async Task SubmitNewCard(Card _card, IBrowserFile? _cardImage){
      var response = await Mediator.Send(new AddCardRequest(_card));
      if (response.insertedId == "" || _cardImage is null){
        _errorMessage = "There was a problem saving your card.";
        _submitSuccessful = false;
        StateHasChanged();
       return;
     }
     var newImageName = await ProcessImage(response.insertedId, _cardImage);
     var responseImageUpdate = await Mediator.Send(new UploadCardImageRequest(response.insertedId,
newImageName));
(continues...)
```

If there was an error saving the card, manually call StateHasChanged to update the UI with the error message.

Shows a manual call to StateHasChanged to trigger an update of the UI

Refactoring AddTrailPage.razor

- With that small refactor out of the way, we can get back to updating the code block for AddCard.razor.

```
if (responseImageUpdate.modifiedCount == 0){
    _errorMessage = "There was a problem saving your image.";
    _submitSuccessful = false;
    _cardForm.ResetForm();
    StateHasChanged();
    return;
  }
  _cardForm.ResetForm();
}
private async Task<string> ProcessImage(string cardId, IBrowserFile? _cardImage){
  string imageUploadResponse = await UploadImage.HandleUpload(cardId, _cardImage);
  if (string.IsNullOrWhiteSpace(imageUploadResponse)){
    _errorMessage = "Your card was saved, but there was a problem uploading the image.";
    return "";
  }
  _submitSuccessful = true;
  StateHasChanged();
  return imageUploadResponse;
}
}
```

If we are here, we've attempted to upload a card image. We trigger a UI update to show the result of that operation.

https://gist.github.com/brunobmo/544c0bcc8721c1549b49be08bbd0781d

Adding the edit card feature

- Before we start making any changes to enable editing, we need to make some changes:
  - We must update the Card class—specifically, to handle updating the card image.
  - We need to add two new requests for our edit functionality: EditCardRequest and GetCardRequest.

Adding the edit card feature

- When editing a card, we will need to be able to display the card's current image, if it has one.
- We will also need to give the user the ability to remove the image, update it, or leave it unchanged.
- To enable these scenarios, we'll update the class with two additional properties and a new enum.

Adding the edit card feature

```
public class Card{
 (...)
 public string Image { get; set; } = "";
 public ImageAction ImageAction { get; set; }
}

public enum ImageAction
{
  None,
  Add,
  Remove
}
(...)
```

Image holds the filename of an existing image.

ImageAction allows us to set what operation to perform on the trail image when updating the trail.

Contains the various operations that can be performed on an image

https://gist.github.com/brunobmo/937f66b42e2c9a5fc5a6e66018007c1f

Adding the edit card feature

- We're going to add GetCardRequest. We'll add this in the new EditCard folder we just created.

```
public record GetCardRequest(string cardId) : IRequest<GetCardRequest.Response>
{
   public const string RouteTemplate = "https://data.mongodb-api.com/app/data-
docuz/endpoint/data/beta/action/findOne";
   (…)
   public record Response(Card card);
   public record Card(
     int Id,
     string Name,
     string Description,
     string? Image,
     DateTime? Date,
     int TimeInMinutes,
     IEnumerable<Tag> tags);
     public record Tag(int Id, string Name, string Description);
}
```

The record contains a single property that holds the ID of the card to retrieve.

The request returns a response that contains all the information needed by the Card form.

https://gist.github.com/brunobmo/adc2ed8a2a81d0f083e6db198fe14570

Adding the edit card feature

- The other request we need to add is the EditCardRequest. This will be called when the form is submitted once the user has finished editing.

```
public record EditCardRequest(Card card) : IRequest<EditCardRequest.Response>
{
  public const string RouteTemplate = "https://data.mongodb-api.com/app/data-
docuz/endpoint/data/beta/action/UpdateOne";
  (...)

  public record Response(bool IsSuccess);
}

public class EditCardRequestValidator : AbstractValidator<EditCardRequest>
{
   public EditCardRequestValidator(){
     RuleFor(x => x.card).SetValidator(new CardValidator());
   }
}
```

Shows the edited card data stored in the Card property on the record

To validate the card, we reuse the CardValidator that lives with the Card. This ensures we're only using one set of validation rules whether we're adding or editing a trail.

https://gist.github.com/brunobmo/ee6af33f32e2636e4faa5d90f44b01fd

Adding the edit card feature

- Let's create the handlers (GetCardHandler):

```
public class GetCardHandler : IRequestHandler<GetCardRequest, GetCardRequest.Response{
    private readonly HttpClient _httpClient;
    (...)
    public async Task<GetCardRequest.Response> Handle(GetCardRequest request, CancellationToken
cancellationToken){
        (...)
         var mongoJson = (...)
        + "\",\"filter\": { \"Id\": "+ request.cardId +" }}";
        (...)
        if (response.IsSuccessStatusCode){
          var responseJson = await response.Content.ReadAsStringAsync();
          var jsonObject = JsonConvert.DeserializeObject<dynamic>(responseJson);
          GetCardRequest.Card c =
    JsonConvert.DeserializeObject<GetCardRequest.Card>(jsonObject.document.ToString());
          return new GetCardRequest.Response(c);
        }else{
          return new GetCardRequest.Response(null);
        }
    }
}
```

https://gist.github.com/brunobmo/4244e9c5e503a069e7284c1f7bfe2d28

Adding the edit card feature

- Let's create the handlers (EditCarHandler):

```
public class EditCardHandler : IRequestHandler<EditCardRequest, EditCardRequest.Response>{
(...)
  public async Task<EditCardRequest.Response> Handle((...))
    var mongoJson = (...)
    + "\",\"filter\": { \"_id\":"+ request.card.Id + "},"
    + "\"update\": { \"$set\": { \"Name\":\""+ request.card.Name +"\","
    + "\"Description\":\""+ request.card.Description +"\","
    + "\"Image\":\""+ request.card.Image +"\","
    + "}}}";
    (...)
   if (response.IsSuccessStatusCode){
     var responseJson = await response.Content.ReadAsStringAsync();
     var jsonObject = JsonConvert.DeserializeAnonymousType(responseJson,new { matchedCount = 0,
modifiedCount = 0 });
     return new EditCardRequest.Response(jsonObject.modifiedCount == 0 ? 0 : jsonObject.modifiedCount);
   }else{
      return new EditCardRequest.Response(0);
   }
  }
}
```

https://gist.github.com/brunobmo/c1ed50e21f6080a0259536672fbde31a

Adding the edit card feature

- Now, i are going to create the EditPage component (inside EditCardFolder):

```
@page "/edit-card/{CardId}"
@inject IMediator Mediator

<PageTitle>Edit Card</PageTitle>

<nav aria-label="breadcrumb">(...)</nav>

@if (_isLoading){
  <p>Loading card...</p>
}else{
  <h3 class="mt-5 mb-4">Editing trail: @_card.Name</h3>

@if (_submitSuccessful){
   <SuccessAlert Message="Your card has been edited successfully!" />
}
else if (_errorMessage is not null){
   <ErrorAlert Message="@_errorMessage" />
}

<CardForm Card="_card" OnSubmit="SubmitEditCard" />
}
```

As we need to load the card being edited from the API, we're going to show a loading message until the card is available.

The CardForm is referenced as in the AddCard. However, this time it also provides a handler for the OnSubmit event. We're also passing in the card to be edited.

Adding the edit card feature

- Now let's take a look at the code block

```
@code {
private bool _isLoading, _submitSuccessful;
private string? _errorMessage;
private Card _card = new Card();
[Parameter]
public string CardId { get; set; }
protected override async Task OnInitializedAsync()
{
    _isLoading = true;
    var response = await Mediator.Send(new GetCardRequest(CardId));
    if (response.card is not null){
      _card.Name = response.card.Name;
      (...)
     _card.Tags.Clear();
     _card.Tags.AddRange(response.card.Tags.Select(t => new Tag { Id = t.Id, Name = t.Name }));
    }
    else{_errorMessage = "There was a problem loading the card.";}
    _isLoading = false;
}
```

If the card is returned, its details are copied into a local field, which is passed to the CardForm.

Adding the edit card feature

```
private async Task SubmitEditCard(Card card, IBrowserFile? image){
  var response = await Mediator.Send(new EditCardRequest(card));
  if (response.modifiedCount == 0){
   _submitSuccessful = false;
   _errorMessage = "There was a problem saving your card.";
  }else{
     _card.Name = card.Name;
     (...)
     _card.Tags.AddRange(card.Tags.Select(t => new Tag { Id = t.Id, Name = t.Name }));
     _submitSuccessful = true;

     (continues...)
}
```

If there was an error saving the card, an error message is shown

Do not associate the image to local field to preserve the old one if replaced

Any updates made to the card instance from the form are applied to the card

Adding the edit card feature

```
if (card.ImageAction == ImageAction.Add){
    var newImageName = await ProcessImage(card, image!);
    if(!string.IsNullOrWhiteSpace(card.Image)){
      UploadImage.RemoveImage(_card.Image);
    }
    _card.Image= newImageName;
    var responseImageUpdate = await Mediator.Send(new UploadCardImageRequest(_card._Id,
newImageName));
    _submitSuccessful = newImageName.Equals("") || responseImageUpdate.modifiedCount == 0 ? false :
true;
    }else if (card.ImageAction == ImageAction.Remove){
      //not used for this scenario since image in mandatory
      UploadImage.RemoveImage(card.Image);
      _card.Image = null;
     }
}
StateHasChanged();
```

If the user updated the card image, ProcessImage is called to upload the new image

When user replace a image

If a new image was selected, update the local cards Image property with the new filename.

If the user removed the image, the Image property is cleared.

https://gist.github.com/brunobmo/4c98e77694478f64f42ae4ec6d9d572f

Adding the edit card feature

```
private async Task<string> ProcessImage(int cardId, IBrowserFile cardImage){
  string imageUploadResponse = await UploadImage.HandleUpload(cardImage);

  if (string.IsNullOrWhiteSpace(imageUploadResponse)){
    _errorMessage = "Your card was saved, but there was a problem uploading the image.";
    return "";
  }
  _card.Image = imageUploadResponse;
  return imageUploadResponse;
}
```

https://gist.github.com/brunobmo/6646158c61fe7b556612deeaa94230c5

Adding the edit card feature

- If the request was successful, then the card instance is updated with the values of the card from the form
- We do this because when we call StateHasChanged at the end of the method, the form will lose any changes made.
- This happens because the CardForm is a child of the EditFormPage and calling StateHasChanged will re-render the EditCardPage.
- This also re-renders the CardForm and provides a fresh copy of any parameters being passed in.
- As we're passing in the card from the EditCardPage, this will overwrite any changes entered by the user.

Updating CardForm to Handle Editing

- To handle editing, we need to complete some updates to the CardForm component.
- We need to add a parameter that allows a card to be passed in for editing.
- We also need to add some logic that will handle setting the ImageAction

Updating CardForm to Handle Editing

- Update CardForm with:

```
[Parameter]
public Card? Card { get; set; }
```

The Card parameter will allow an existing card to be passed into the form

- And with OnParametersSet method:

```
protected override void OnParametersSet(){
  _editContext = new EditContext(_card);
  _editContext.SetFieldCssClassProvider(new BootstrapCssClassProvider());
  if (Card != null){
   _card.Name = Card.Name;
   _card.Description = Card.Description;
   _card.Image = Card.Image;
   _card.TimeInMinutes = Card.TimeInMinutes;
   _card.Tags.Clear();
   _card.Tags.AddRange(Card.Tags.Select(t => new Tag { Id = t.Id, Name = t.Name }));
  }
}
```

OnInitialized is replaced with OnParametersSet. This will be called whenever an update happens to the object passed in via the Card parameter. We need this so we can update or remove the image after the SubmitEditCard handler in the EditCard runs.

https://gist.github.com/brunobmo/4dbf417d3de45ca479500a32cae0e997

Updating CardForm to Handle Editing

- Our final job in the CardForm component is to handle setting the ImageAction property. In CardForm:

```
<FormFieldSet Width="col-6">
    <label for="cardImage" class="font-weight-bold text-secondary">Image</label>
    @if (string.IsNullOrWhiteSpace(_card.Image) || _card.Image.Equals("-"))
        <InputFile OnChange="LoadCardImage" class="form-control-file" id="cardImage"
accept=".png,.jpg,.jpeg" />
        <ValidationMessage For="@(() => _card.Image)" />
    }else{
        <div class="card bg-dark text-white">
          <img src="images/@_card.Image" />
            <div class="card-img-overlay">
                <button class="btn btn-primary btn-sm" @onclick="RemoveCardImage">Remove</button>
            </div>
        </div>
    }
</FormFieldSet>
```

If the card doesn't have an image, render the InputFile component, allowing the user to select one. "-" character is a dummy value to indicate that an image was uploaded.

If the card has an image, display it along with a button to remove it.

https://gist.github.com/brunobmo/bf11661788417be1be578fa32da93646

Updating CardForm to Handle Editing

- Our final job in the CardForm component is to handle setting the ImageAction property. In CardForm:

```
private void LoadCardImage(InputFileChangeEventArgs e)
{
  _cardImage = e.File;
  _card.Image = "_";
  _card.ImageAction = ImageAction.Add;
}

private void RemoveCardImage()
{
  _card.Image = "";
  _card.ImageAction = ImageAction.Remove;
}
```

Set the ImageAction to add when an image is selected.

This method is called when the Remove Image button is clicked. It will reset the Image property, triggering the InputFile component to show. It also marks the image to be removed on the server.

https://gist.github.com/brunobmo/753a7d509a9998d6dc92e39d00245af9

Use _id Mongo identifier

- Until now, we are using Card id to identify the Card. However, it is not a reliable approach since the id is generated using a class variable;
- To guarantee uniqueness we need to use _id field from Mongo.
- We need to change the class Card, the route from EditCardPage.razor, CardForm, and the EditCardHandler to use _id instead CardId.
- Exercise: do it!

https://gist.github.com/brunobmo/753a7d509a9998d6dc92e39d00245af9

Summary

- While Blazor Input components output default validation class names, it is possible to customize them by providing a custom FieldCssClassProvider.
- If using a custom FieldCssClassProvider, it must be registered with the EditContext.
- Blazor provides an out-of-the-box component called InputBase<T> as a starting point for creating custom input components.
- A type parameter must be specified when inheriting from InputBase<T>. When binding model properties to the component, they must match the type parameter.
- When inheriting from InputBase<T> an implementation must be provided for the TryParseValueFromString method; however, it may not be used.

Summary

- There are two properties provided by InputBase<T> to update the model value bound to a custom input component: CurrentValueAsString and CurrentValue.
- InputFile is a component included with Blazor for working with files in forms.
- The bind directive isn't used when working with InputFile. Instead, a handler must be provided for the component's OnChange event.
- The OnChange event provides its handler with InputFileChangeEventArgs, which contains the file(s) selected by the user along with the total count of files selected.
- By default, the user can select a maximum of 10 files—selecting more will result in the component throwing an exception[…]

Blazor Server

Forms and Validation – Beyond the basics

2023