Blazor Server

Forms and Validation - Fundamentals

2023

Forms and validation

- With Blazor, it’s possible to work with HTML forms directly. However, that is not ideal.
- While collecting the data entered by the user and handling the form submit event is easy enough, one major piece of functionality is still missing: validation.
- Validation is the main reason for using the form components provided by Blazor over a standard HTML form.

Forms and validation

- The primary component is EditForm—a drop-in replacement for an HTML form element.
- Inside this, we add a validator component and various input components along with a standard HTML Submit button.
- We pass a model into the EditForm, which is an instance of a class that represents the data we want to collect with the form.
- Internally, the EditForm component constructs an EditContext—this is the brain of the form's system.

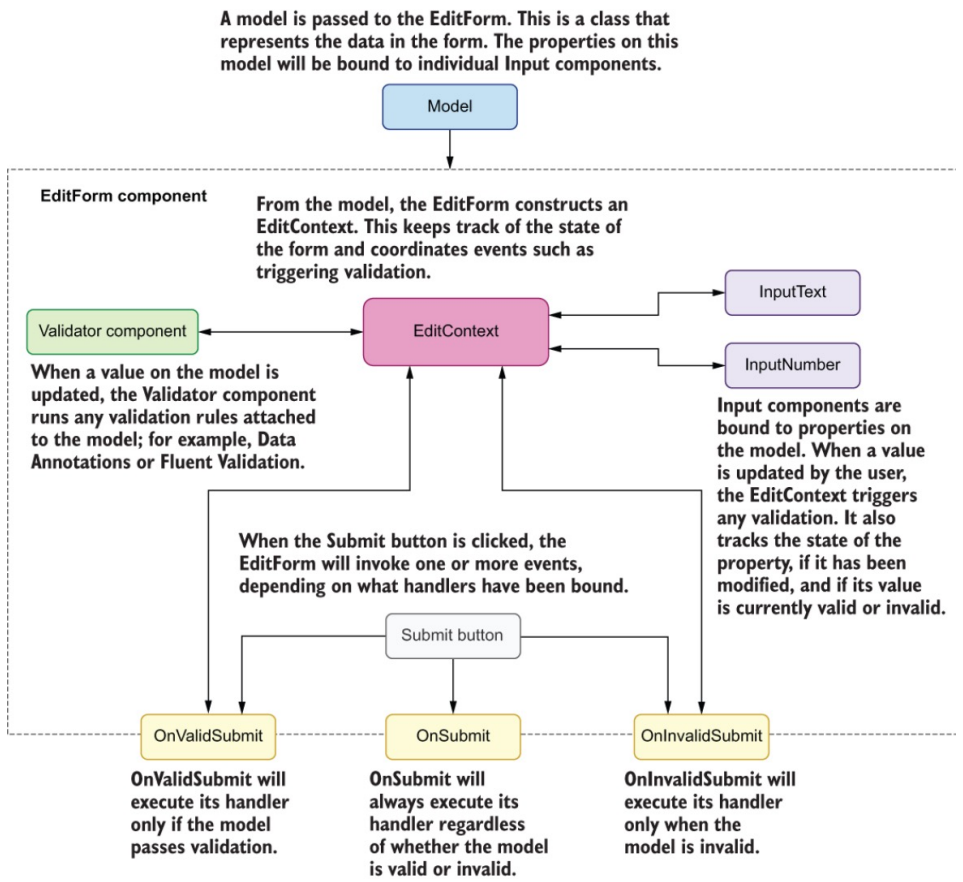
Forms and validation

- It keeps track of all the input components and the state of the model.
- Whenever a value is updated on the model, it will trigger validation via a validator component.
- Blazor ships with a validator component called DataAnnotationsValidator, which allows the validation of models using the Data Annotations.

Forms and validation

- The EditForm offers three events for handling form submits:
  - The OnSubmit event is the same as the standard submit event on an HTML form. It will be invoked whenever the Submit button is clicked, and the handler is responsible for making sure the model is valid.
  - The OnValidSubmit is triggered when the Submit button is clicked but with a key difference: the EditForm will check with the EditContext first to make sure the model is valid. The handler will be called only when the model is valid.
  - The final event is OnInvalidSubmit. As you can probably guess, this works in the opposite way to the OnValidSubmit event. It will only be invoked when the form is submitted but the model is invalid.

Forms and validation



An overview of the various components and services that make up Blazor's forms and validation system

Forms and validation

- Add a new folder called ManageCards;
- Next, add the following file: AddCard.razor;

```
@page "/add-card"

<PageTitle>Add Card</PageTitle>

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

<h3 class="mt-5 mb-4">Add a card</h3>

<EditForm Model="_card" OnValidSubmit="SubmitForm">
   <div class="mt-4 mb-5">
     <div class="row">
        <div class="offset-4 col-8 text-right">
           <button class="btn btn-outline-secondary" type="button"
@onclick="@(() => _card = new Card())">Reset</button>
           <button class="btn btn-primary" type="submit">Submit</button>
       </div>
     </div>
    </div>
</EditForm>
```

https://gist.github.com/brunobmo/2837a4cc74e4bf2770e0ae0ac0254cf0

Forms and validation

- To use the EditForm component, we need to give it two things: a model and a method to call when the form is submitted.
- The model is used internally to understand what validation rules exist and the current state of the model (i.e., whether it is valid or invalid).
- The other setup parameter we must provide is the submit handler.

Forms and validation

- We'll create a new component in the ManageCards folder called FormSection.razor.

```
<div class="card card-brand mb-4 shadow">
  <div class="card-body">
    <div class="row">
      <div class="col-4">
         <h4>@Title</h4>
         <p class="text-secondary">@HelpText</p>
      </div>
      <div class="col-8"> @ChildContent</div>
    </div>
  </div>
</div>

@code {
    [Parameter, EditorRequired]
    public string Title { get; set; } = default!;
    [Parameter, EditorRequired]
    public string HelpText { get; set; } = default!;

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

Shows the content to be displayed

https://gist.github.com/brunobmo/969746a8a0bf7add480c9ebd0abdb3e2

Forms and validation

- By defining this simple component now, we've saved a lot of repeated markup on our form, making maintenance in the future much easier.
- We also need to add a CSS file for the styles for the FormSection:

```
.card-brand {
   border-top: 4px solid var(--brand);
}
```

https://gist.github.com/brunobmo/0ee090b7abd07e1a3a6ec58fcc3f5261

Collecting data with input components

- We now turn our attention to input components.
- Out of the box, Blazor ships with component versions of the standard HTML form input elements.
- Except for InputFile, we can use any of these input components and we just need to bind them to a property on the form model using the @bind directive.

Collecting data with input components

- HTML input elements and their Blazor equivalente

HTML input control	Blazor input component
<input> or <input type="text" />	<InputText />
<textarea>	<InputTextArea />
<input type="number">	<InputNumber />
<select>	<InputSelect>
<input type = "date">	<InputDate />
<input type="checkbox"	<InputCheckbox />
<input type="radio">	<InputRadio /> and <InputRadioGroup>
<input type="file">	<InputFile />

Collecting data with input components

- Let's change the AddCard.razor:

```
<EditForm Model="_card" OnValidSubmit="SubmitForm">
   <FormSection Title="Basic Details" HelpText="This">
     <FormFieldSet Width="col-6">
        <label for="cardName" class="font-weight-bold text-secondary">Name</label>
        <InputText @bind-Value="_card.Name" class="form-control" id="cardName" />
     </FormFieldSet>
     <FormFieldSet>
        <label for="cardDescription" class="font-weight-bold text-secondary">Description</label>
        <InputTextArea @bind-Value="_card.Description" rows="6" class="form-control" id="cardDescription" />
     </FormFieldSet>
     <FormFieldSet Width="col-6">
        <label for="cardImage" class="font-weight-bold text-secondary">Image</label>
        <InputText @bind-Value="_card.Image" class="form-control" id="cardImage" />
     </FormFieldSet>
   </FormSection>
   <div class="mt-4 mb-5">
      <div class="row">
        <div class="offset-4 col-8 text-right">
          <button class="btn btn-outline-secondary" type="button" @onclick="@(() => _card = new
Card())">Reset</button>
          <button class="btn btn-primary" type="submit">Submit</button>
        </div>
      </div>
   </div>
</EditForm>
```

The new FormSection component defines the section title and help text.

The @bind directive has a slightly different syntax to what we've seen before, as when binding to a component, we must specify the parameter we're binding to.
All the Input components shipped with Blazor expose a Value parameter, hence @bind-Value.

Collecting data with input components

- Using @bind-Value="_someValue”, we are now performing two-way binding on a component rather than an HTML element.
- When binding to a component, we must specify the property on the component we wish to bind to.
- In the case of Blazor’s input components, that property is called Value—note the capital V.

Collecting data with input components

- Looking at the form components we just added, there is a lot of repetition again.
- We'll create a new component called FormFieldSet.razor in the ManageCards feature folder:

```
<div class="row">
    <div class="@Width">
      <div class="form-group">@ChildContent </div>
    </div>
</div>
@code {
   [Parameter, EditorRequired]
    public RenderFragment ChildContent { get; set; } = default!;
   [Parameter]
    public string Width { get; set; } = "col";
}
```

https://gist.github.com/brunobmo/0429d9f1f58abebb20569848dacfdc7f

Collecting data with input components

- Let's update the Basic Details section we just added to use the FormFieldSet componente in AddCard.razor.

```
<FormSection Title="Basic Details" HelpText="This information is used to...">
    (...)
</FormSection>
<div class="mt-4 mb-5">
   <div class="row">
     <div class="offset-4 col-8 text-right">
        <button class="btn btn-outline-secondary" type="button" @onclick="@(() => _card = new Card())">Reset</button>
        <button class="btn btn-primary" type="submit">Submit</button>
      </div>
   </div>
</div>

@code {
  private Card _card = new Card();
  private async Task SubmitForm()
  {
  }

}
```

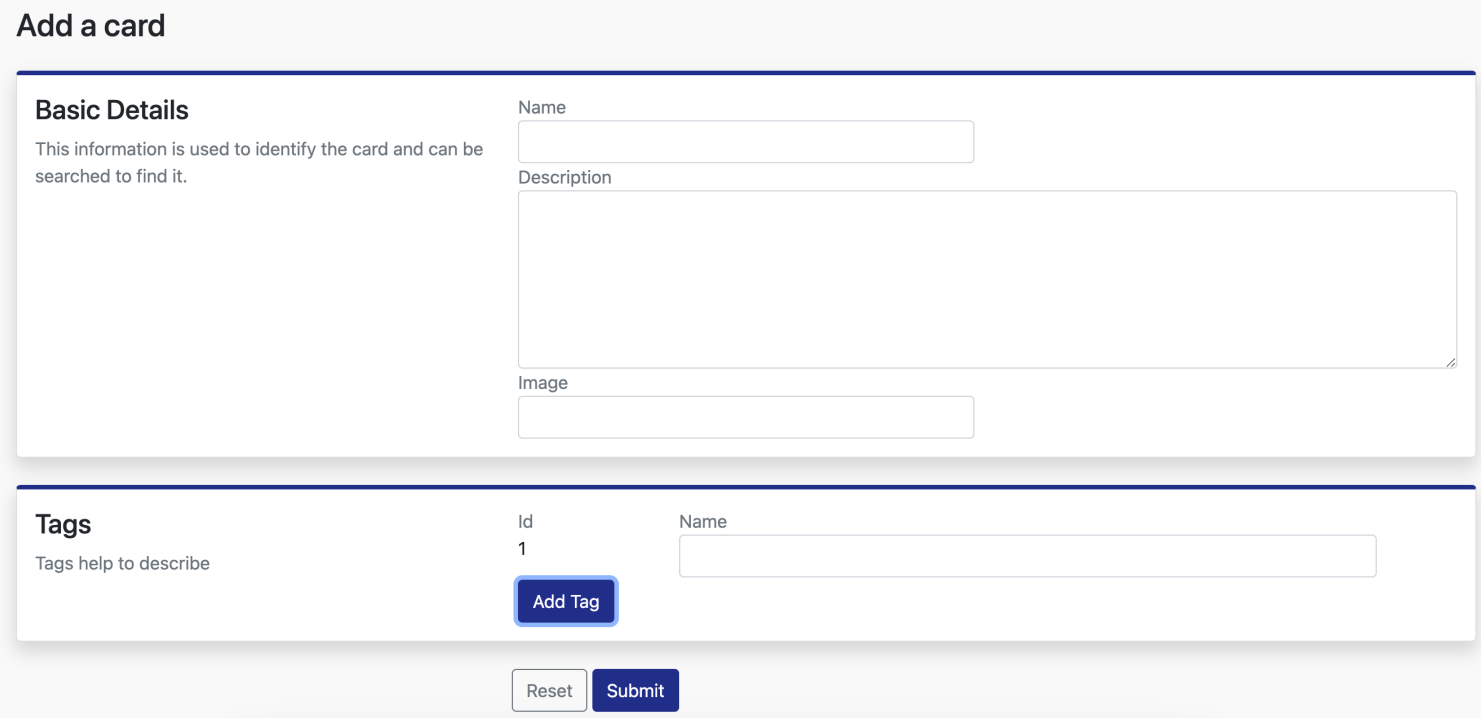
https://gist.github.com/brunobmo/0d2ecf916fac7d522a908d6f0aabb7df

Creating inputs on demand

- At some point, you will need to allow the user to create inputs on demand.
- We need to build the form in a way that allows the user to dynamically add tags as they need.

Creating inputs on demand

- At some point, you will need to allow the user to create inputs on demand.
- We need to build the form in a way that allows the user to dynamically add tags as they need.



Creating inputs on demand

- We will add the concept of tag, creating the Tag.cs file in Home folder:

```
namespace app.Features.Home{
  public class Tag{
    public int Id { get; set; }
    public string Name { get; set; } = "";
  }
}
```

https://gist.github.com/brunobmo/b6f5be0c74a7e8cbc18754eee21c2544

Creating inputs on demand

- Add data do cards.json:

```
[{
  "id": 1,
  "name": "Card 1",
  "description": "Card 1 description",
  "image": ”(...).jpg",
  "date": "2001-01-01T09:00:00Z",
  "tags": [
     {
      "id": 1,
      "name": "Tag 1"
     },
     {
      "id": 2,
      "name": "Tag 2"
     }
  ]
},
(...)
```

https://gist.github.com/brunobmo/1071ad0f2f777735e4597f69eea92e77

Creating inputs on demand

- Add a field to Card class to hold tags:

```
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; } = "";
    public DateTime? Date { get; set; }
    public List<Tag> Tags { get; set; } = new List<Tag>();
}
```

https://gist.github.com/brunobmo/505b5cba9eb0eb849f67db11033de5a0

Creating inputs on demand

- Change AddCard.razor form:

```
<FormSection Title="Tags" HelpText="Tags help to describe ">
@{
    var i = 0;
}
@foreach (var tag in _card.Tags){
  i++;
  tag.Id = i;
  <div class="row">
    <div class="col-2">
      <div class="form-group">
        <label class="font-weight-bold text-secondary">Id</label>
        <p>@tag.Id</p>
      </div>
    </div>
    <div class="col">
       <div class="form-group">
          <label for="tagDescription" class="font-weight-bold text-secondary">Name</label>
          <InputText @bind-Value="tag.Name" class="form-control" id="tagName" />
       </div>
    </div>
    (continua...)
```

Creating inputs on demand

- Change AddCard.razor form:

```
  <div class="col-1 d-flex mt-3">
    <button @onclick="@(() => _card.Tags.Remove(tag))" class="btn btn-link" type="button">
      <svg width="1em" height="1em" viewBox="0 0 16 16" class="bi bi-x-circle-fill text-danger" fill="currentColor"
xmlns="http://www.w3.org/2000/svg">
        <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>
    </button>
  </div>
</div>
}

 <div class="row">
   <div class="col">
       <button class="btn btn-outline-primary" type="button" @onclick="@(() => _card.Tags.Add(new Tag()))"> Add Tag
  </button>
   </div>
 </div>
</FormSection>
```

Clicking this button will remove a card tag from the list. Blazor will automatically re-render the UI and remove the relevant form controls for that entry.

Clicking the Add card button will add a new route instruction to the list. This triggers Blazor to iterate the collection and output the contents of the foreach loop, enabling the user to enter the details of that tag.

https://gist.github.com/brunobmo/5d8774316ac56517beb8581dd83e0834

Validating the model

- Blazor includes few components to validate forms:
  - DataAnnotationsValidator
  - ValidationSummary
  - ValidationMessage

Validating the model

- The DataAnnotationsValidator component allows Blazor forms to work with the Data Annotations validation system, which is the default for ASP.NET Core applications.
- This system works by decorating properties on a model with attributes that define the validation rules.
- For example, to make a text property required, we would do the following

```
[Required]
public string Name { get; set;}
```

Validating the model

- The ValidationSummary component displays all validation messages for a model.
- This can be useful when you want to have all the validation messages for a form grouped together in one place
- Finally, the ValidationMessage component displays a validation message for a specific property on the model.
- This allows a validation message to be displayed directly under, or next to, an input component, making it easy for the user to see where the problem is.

Validating the model

- As with most things in Blazor, there is no restriction on which validation system you can use.
- So, if Data Annotations don’t fit, you can easily swap to a different validation system.
- We are going to use Fluent Validation;
- The syntax of Fluent is more simpler for complex rules;

Configuring validation rules with Fluent Validation

- dotnet add package FluentValidation.AspNetCore
- With the package installed, we can open the Card class and add a using statement to it:

```
using FluentValidation;
```

Configuring validation rules with Fluent Validation

- To set up the validation rules, we need to define a validator class for our Card.
- After the Card class ends, add the class shown in the following slide.

Configuring validation rules with Fluent Validation

Validation classes must inherit from the AbstractValidator<T> class, T being the class to be validated.

```
public class CardValidator :AbstractValidator<Card>
{
    public CardValidator() {
        RuleFor(x => x.Name).NotEmpty().WithMessage("Please enter a name");
        RuleFor(x => x.Description).NotEmpty().WithMessage("Please enter a description");
        RuleFor(x => x.Image).NotEmpty().WithMessage("Please enter a image");
        RuleFor(x => x.Date).NotNull().WithMessage("Please enter a date");
        RuleFor(x => x.Tags).NotEmpty().WithMessage("Please add a tag");
        RuleForEach(x => x.Tags).SetValidator(new TagValidator());
    }
}
```

Validation rules are defined in the constructor of the validation class.

We need to wire up the TagValidator in the CardValidation

Validation rules are defined using a fluent syntax, hence the name. Each rule defines the property it's for, the criteria, and the error message to show if the criteria isn't met.

https://gist.github.com/brunobmo/035d0804a7bf40d59df3a87f0d685617

Configuring validation rules with Fluent Validation

- The rules we've defined will make sure that the Card is valid, but we also need to do the same for the Tag nested class.

```
public class TagValidator : AbstractValidator<Tag>
{
    public TagValidator() {
        RuleFor(x => x.Name).NotEmpty().WithMessage("Please enter a name");
    }
}
```

https://gist.github.com/brunobmo/a9a73fa3ad943e69e0b846e01de4eed7

Configuring Blazor to use Fluent Validation

- Now we have the validation rules set up for the Card, we need to tell Blazor how to understand and process them.
- To do this, we will install a package for doing that:
  - dotnet add package Blazored.FluentValidation --version 2.1.0
- It contains one component called FluentValidationsValidator, which, when included in an EditForm component, will allow the form model to be validated according to any Fluent Validation rules.

Configuring Blazor to use Fluent Validation

- Then we will add the following using statement to the _Imports.razor at the root of the project.

```
@using Blazored.FluentValidation
```

- We are now ready to add the validation components to our form.
- The first thing we'll do is tell the EditForm component that we want the model to be validated with Fluent Validation.

Configuring Blazor to use Fluent Validation

- To do this, we add the FluentValidationsValidator component somewhere between the opening and closing tags of the EditForm

```
<EditForm Model="_card" OnValidSubmit="SubmitForm">
  <FluentValidationValidator />
  <FormSection Title="Basic Details" HelpText="This information is used to ">
  (...)
```

- We now need to add a ValidationMessage component under each of the existing inputs on the form. The following listing shows the updated Basic Details section.

```
<FormFieldSet Width="col-6">
   <label for="cardName" class="font-weight-bold text-secondary">Name</label>
   <InputText @bind-Value="_card.Name" class="form-control" id="cardName" />
   <ValidationMessage For="@(() => _card.Name)" />
</FormFieldSet>
```

Configuring Blazor to use Fluent Validation

- The EditContext keeps track of the state of the form.
- When validation is executed, the input component bound to a property is updated with CSS classes that represent that property's state.
- Those classes are
  - Valid
  - Invalid
  - modified

Configuring Blazor to use Fluent Validation

- In the app.css file, in the wwwroot > css folder, we will add the styles shown in the following listing to the top of the file.

```
.validation-message {
  color: red;
}

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

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

Text for the validation-message class will be red. This class is used by the validation message component.

Any input, textarea, or select element with a class of invalid will have a red border.

Any input, textarea, or select element with a class of valid AND modified will have a border of green.

https://gist.github.com/brunobmo/92d6599dfe2d79aae2595420f65e48fd

Submitting data to the server

- For testing purposes, we are using a MongoDB database, simulating the existence of a API;
- We are going to start by creating a request that will contain the data collected by our form.
- Once this is done, we can create a handler for that request; this will be responsible for posting the data up to the API.

Submitting data to the server

- We are going to use two libraries:
  - APIEndpoints: dotnet add package Ardalis.ApiEndpoints --version 4.0.1
  - MediatR: dotnet add package MediatR --version 12.0.1

Submitting data to the server

- APIEndpoints
  - MVC Controllers are essentially an antipattern.
  - They are collections of methods that never call one another and rarely operate on the same state. They're not cohesive.
  - They tend to become bloated and to grow out of control.
  - Their private methods, if any, are usually only called by a single public method

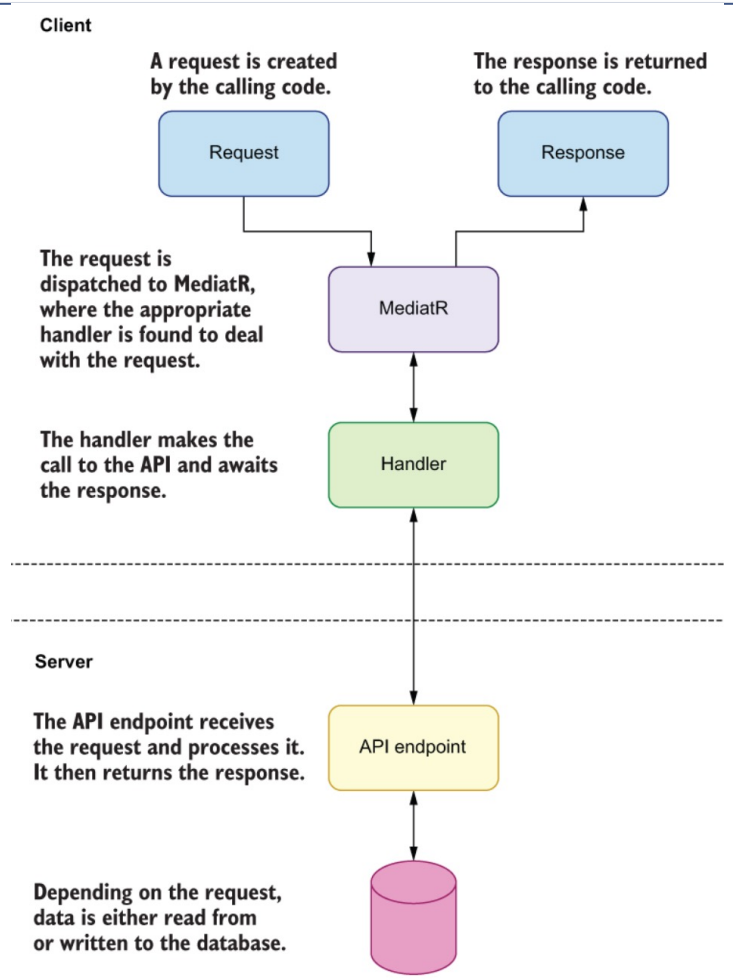
Submitting data to the server

- APIEndpoints
  - ApiEndpoints solves this by allowing us to define an endpoint as a class with a single method to handle the incoming request.
  - This allows us to avoid all the issues that surround controllers and build clear and easy-to-maintain endpoints in our APIs.

Submitting data to the server

- MediatR
  - MediatR is an in-process messaging library that implements the mediator pattern
  - Essentially, requests are constructed and passed to the mediator, which then passes them to a handler.
  - MediatR uses dependency injection to connect requests with handlers.
  - This makes things very flexible and easy to test.
  - The main advantage of using MediatR is the ability to have loose coupling between components and server interactions.

Submitting data to the server



Submitting data to the server

- Taking our form as the example, we will create a request to post the data to the Mongo API and pass this request to MediatR.
- MediatR will route our request to a handler, which will process the request and make the API call.
- On the server, an API endpoint (Mongo) will receive the request and process it.
- In this case, it will save the data into a database, and if there are no issues, it will return a success response—otherwise an error response will be returned.
- This response will travel back until it reaches the calling code.

Submitting data to the server

- To add MediatR to the Project:
  - Add the following to Program.cs:

```
builder.Services.AddMediatR(cfg =>
cfg.RegisterServicesFromAssembly(typeof(Program).Assembly));
```

  - This line adds MediatR to the service collection, so we can inject it into our components and services.
  - It also tells MediatR to scan the current assembly for request handlers.

Submitting data to the server

- The final piece of configuration we're going to do is to add a using statement for MediatR in the root _Imports.razor file: `@using MediatR`

Creating a request and handler to post the form data to the API

- We are going to start by creating a request that will contain the data collected by our form.
- Once this is done, we can create a handler for that request; this will be responsible for posting the data up to the API.

Creating a request and handler to post the form data to the API

- Now we need to create a new class in the Features > ManageCard folder called AddCardRequest.

```
public record AddCardRequest(Card card) : IRequest<AddCardRequest.Response> {
    public const string RouteTemplate = "https://(...)/insertOne";
    public const string ApiKey = "…";
    public const string Collection = "cards";
    public const string Database = "CardsDB";
    public const string DataSource = "Cluster0";
    public record Response{
      public string insertedId { get; set; } = "";
      public Response(string insertedId){
         this.insertedId = insertedId;
      }
    }
}
```

- AddCardRequest is defined as a C# record as opposed to a class. Records are considered preferable for data transfer objects due to their immutability and value-type qualities.
- The record implements the IRequest<T> interface that is used by MediatR when locating a handler.
- T defines the response type of the request.
- These constants define the address of the Mongo API endpoint for the request.
- This nested C# record defines the response data for the request.

https://gist.github.com/brunobmo/d4b03c4a08bf232210f78758641eb846

Creating a request and handler to post the form data to the API

- The first thing to note is that the AddCardRequest is not a C# class, but a C# record.
- Records are a new type introduced in C# 9 and are considered the preferable option for DTOs, which is essentially what our request is.
- The reason for this is that records can be immutable, meaning once the values of its properties have been set, they can't be changed.
- If they need to be changed, then a new copy is made with the updated values.

Creating a request and handler to post the form data to the API

- Create a new C# class called AddCardHandler.cs.

```
public class AddCardHandler : IRequestHandler<AddCardRequest, AddCardRequest.Response>
{
  private readonly HttpClient _httpClient;

  public AddCardHandler(HttpClient httpClient){
    _httpClient = httpClient;
  }
(...)
```

Request handlers implement the IRequestHandler<TRequest, TResponse> interface. TRequest is the type of request the handler handles. TResponse isthe type of the response the handler will return.

An HttpClient is injected and stored in a field to be used to make API calls

Creating a request and handler to post the form data to the API

- Create a new C# class called AddCardHandler.cs.

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

     var json = System.Text.Json.JsonSerializer.Serialize(request.card);
     var mongoJson ="{\"collection\":\""+ AddCardRequest.Collection(...) + "\",\"document\":"+ json + "}";
     var requestBody = new StringContent(mongoJson,System.Text.Encoding.UTF8,"application/json");
     requestMessage.Content = requestBody;
     var response = await _httpClient.SendAsync(requestMessage);

     if (response.IsSuccessStatusCode){
        var responseJson = await response.Content.ReadAsStringAsync();
        var jsonObject = JsonConvert.DeserializeAnonymousType(responseJson,new { insertedId = "" });
        return new AddCardRequest.Response(
           jsonObject.insertedId == null ? "" : jsonObject.insertedId
        );
     }else{
        return new AddCardRequest.Response("");
     }
}
```

The Handler method is specified by the IRequestHandler interface and is the method called to handle the request by MediatR

https://gist.github.com/brunobmo/0b3344d0a33ab30b19df0bab3029f537

Creating a request and handler to post the form data to the API

- Request handlers contain only a single method called Handle.
- This method is a requirement of the IRequestHandler<TRequest, TResponse> interface.
- As part of implementing this interface, we need to define the type of TRequest and TResponse.
- TRequest is the type of request that the handler should process.
- For us, that type is AddCardRequest.
- TResponse is the type that the handler will return to the caller; for us, that is AddCardRequest.Response.

Creating a request and handler to post the form data to the API

- When the response is returned, we check to see if the request was successful.
- If it was, then we extract the InsertedId from the response and return a new AddCardRequest.Response containing the ID.
- If it wasn't successful, we still return the response but with an empty string, which we can use to show error messages on the UI.

Creating a request and handler to post the form data to the API

- To round out our work on the client, we need to hook up our request to the form's submit event.
- In the AddCardPage.razor component, we are going to inject MediatR at the top using the inject directive

```
@inject IMediator Mediator
```

Creating a request and handler to post the form data to the API

- Update AddCard.razor

```
@code {
  private Card _card = new Card();
  private bool _submitSuccessful;
  private string? _errorMessage;

  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;
    }
   _card = new Card();
   _errorMessage = null;
   _submitSuccessful = true;
  }

}
```

This is a new field to track if the form was submitted successfully

This is a new field to store an error message if something went wrong submitting the form.

MediatR is used to dispatch the AddCardRequest and await the response.

A new Card instance is created, which resets the form ready for a new card to be input

https://gist.github.com/brunobmo/595b78580032fd303b20ea5c07b779ec

Creating a request and handler to post the form data to the API

- Inside the SubmitForm method, we're using the Mediator service supplied by MediatR to send the AddCardRequest—we then await the response.
- If it has a empty string, we assign an error message and set _submitSuccessful to false.
- Otherwise, we create a new Card instance, which clears the form ready for another card to be added.

Creating a request and handler to post the form data to the API

- The final update is to add a small piece of UI to show if the form submitted successfully or not. We're going to add the code shown in the following listing directly above the EditForm component.

```
@if (_submitSuccessful){
    <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="(...)" />
       </svg>
       Your card has been submitted successfully!
    </div>
}
else if (_errorMessage is not null){
    <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="(...)" />
      </svg>
      @_errorMessage
   </div>
}
```

Summary

- The primary advantage of using Blazor form components over traditional HTML forms is validation.
- The EditForm component is a drop-in replacement for the HTML form element.
- Blazor ships with component versions of all the standard HTML input controls.
- The EditForm requires a model that represents the data the form will collect, as well as a handler for one of the submit events it exposes (OnSubmit, On-ValidSubmit, OnInvalidSubmit).

Summary

- Internally, the EditForm will construct an EditContext, which is the brain of the form and keeps track of the validation state of the model, as well as coordinates validation events.
- To use Blazor's input components, they must be bound to a property on the model passed to the EditContext. This is done using the @bind directive.
- Blazor ships with a validation component called DataAnnotationsValidator, which enables the Data Annotations validation system to be used with models.
- The validation system in Blazor is flexible and extendable, and other validation systems can easily be added by swapping out the validator component

Blazor Server

Forms and Validation - Fundamentals

2023