ASP.NET Core Clean Architecture, DDD, and Modular Monoliths

Steve “ardalis” Smith | NimblePros.com | October 2024

Workshop Description

In this hands-on workshop you will work with your instructor to build a new ASP.NET Core app from scratch using Clean Architecture, Domain-Driven Design, and Modular Monolith principles and patterns. Along the way, you will learn - and experience firsthand - the benefits of these ideas - when used appropriately - for building testable, maintainable applications.

Lab 0: Set up

Goal

Have a working development environment and starter application.

Requirements

- **Visual Studio 2022** or **Visual Studio Code plus CLI tools**.
- **.NET Core 8.0 SDK (or later)**

It will also be helpful to install:

- **Docker**
- **Postman/Fiddler/.http files** (some way to test API endpoints)
- **Papercut** or another local email testing tool.

Instructions

We’re going to be building a simple book store application today using ASP.NET Core, Clean Architecture, and DDD. We’re going to build it in a modular fashion and deploy it as a single monolith. We’re going to make heavy use of some templates that will allow us to move quickly, which we’ll install now.

```
dotnet new install Ardalis.CleanArchitecture.Template
```

```
dotnet new install Ardalis.Modulith
```

Latest versions as of this edition of this document: 9.3.0 and 1.1.0.

If you need to confirm that emails are being sent via localhost, run this docker command and navigate to localhost:37408 to see any emails your app has sent.

```
docker run -d --name=papercut -p 25:25 -p 37408:37408 jijiechen/papercut:latest
```

You can access these at localhost:37408 (papercut) and localhost:8080 (rabbitmq).

Lab 1: The Guestbook application domain model

Expected time to complete: About 30 minutes. (Less if copy/pasting)

Goal
Create a domain model consisting of a Guestbook and GuestbookEntry Entities. Create an API endpoint that returns the most recent guestbook entries.

Topics Used
Entities, Mapping, Request-Endpoint-Response (REPR) pattern

Requirements
We are building an online guestbook. Anonymous users will be able to view entries on the guestbook, as well as (later) leave messages (GuestbookEntries) on the site. Each **GuestbookEntry** must include a value for the user's email address and a message, as well as a DateTimeCreated. For now these won't have much behavior; we'll add more later.

The home page (of whatever app we end up building - we're just making API endpoints) should display the most recent ten (10) entries recorded on the guestbook, ordered by when the entry was recorded (most recent first). This will be the default behavior of the **List endpoint** - return the most recent 10 entries.

Details
Create a New Guestbook App

To start you'll need to **create a new app in a new folder** using the Clean Architecture template. Run this command:

```
dotnet new clean-arch -o Nimble.GuestbookApp
```

Navigate to the new Nimble.GuestbookApp folder and open the solution file.

Guestbook Entities

1. **Note:** Your templated solution's **Core** project includes a ContributorAggregate folder which you can use for reference. We won't be using any of the files in it directly for this lab.
2. In the **Core** project, create a new root level folder, **GuestbookAggregate.**
3. In this folder, add a new class, `GuestbookEntry`, with these properties:
    a. string `EmailAddress`
    b. string `Message`
    c. DateTimeOffset `DateTimeCreated` (defaults to UtcNow)
    d. Be sure to initialize strings to string.Empty to avoid nullable errors.
4. In the same folder, add another new class, `Guestbook`, with these properties:
    a. string `Name`
    b. List<GuestbookEntry> `Entries`
5. **Both Guestbook** and **GuestbookEntry** should inherit from EntityBase, which defines an integer Id property.
6. Make sure both classes are marked as public.
7. `Guestbook` (only) should implement IAggregateRoot.

(Example code at the end of the lab)

Don't forget to initialize your collection properties to [] (empty collection) to avoid null reference exceptions later.

List Endpoint

Next we're going to add a List endpoint in the Web project. We're using the FastEndpoints approach to creating minimal API endpoints, and we're going to have a single root level folder for all of our GuestbookEntry endpoints.

1. Create a folder in **Nimble.Guestbook.Web** called **Entries**.
2. Create a new class, **List.cs**, in the Entries folder.
3. Implement List.cs as follows:
    a. Inherit from `EndpointWithoutRequest<List<GuestbookEntry>>`
    b. Override `Configure()` method and add `Get("/entries")` and `AllowAnonymous()`
    c. Override `HandleAsync()`
        i. Create a new Guestbook instance.
        ii. Add 3 Guestbook entries to the guestbook's Entries property (just make up data)
        iii. Assign Guestbook.Entries to Response.
4. Test the application. You should be able to test the Entries endpoint in Swagger.
5. Once that works, change the result to be ordered by DateTimeCreated, Descending.

There are some samples on the next page to help you if you get stuck.

Lab 1 Samples

Core/GuestbookAggregate/Guestbook.cs
```
using Ardalis.SharedKernel;

namespace Nimble.Guestbook.Core.GuestbookAggregate;
public class Guestbook : EntityBase, IAggregateRoot
{
  public string Name { get; set; } = "";

  public List<GuestbookEntry> Entries { get; } = [];
}
```

Core/GuestbookAggregate/GuestbookEntry.cs
```
using Ardalis.SharedKernel;

namespace Nimble.Guestbook.Core.GuestbookAggregate;

public class GuestbookEntry : EntityBase
{
  public string EmailAddress { get; set; } = "";
  public string Message { get; set; } = "";
  public DateTimeOffset DateTimeCreated { get; set; } = DateTime.UtcNow;
}
```

Web/Entries/List.cs
```
using FastEndpoints;
using Nimble.GuestbookApp.Core.GuestbookAggregate;

namespace Nimble.GuestbookApp.Web.Entries;

public class List : EndpointWithoutRequest<List<GuestbookEntry>>
{
  public override void Configure()
  {
    Get("/Entries");
    AllowAnonymous();
  }

  public override async Task HandleAsync(CancellationToken cancellationToken)
  {
    var guestbook = new Guestbook();

    guestbook.Entries.Add(new GuestbookEntry()
    {
      DateTimeCreated = new DateTime(2024, 1, 1),
      EmailAddress = "alice@test.com",
      Message = "Hello world!",
    });
    guestbook.Entries.Add(new GuestbookEntry()
    {
      DateTimeCreated = new DateTime(2024, 2, 14),
      EmailAddress = "bob@test.com",
      Message = "Happy Valentine's Day!",
    });
    guestbook.Entries.Add(new GuestbookEntry()
    {
      DateTimeCreated = new DateTime(2024, 6, 20),
      EmailAddress = "carol@test.com",
      Message = "Happy summer!",
    });

    await Task.Delay(1);
    Response = guestbook.Entries
      .OrderByDescending(e => e.DateTimeCreated)
      .ToList();
  }

}
```

Try it out! Run the app using the https profile, then open https://localhost:57679/swagger OR open the **api.http** file and add a GET request to `/entries`:

```
// List all entries
GET {{host}}:{{port}}/entries

###
```

Sample List Endpoint Response body:

```
[ { "emailAddress": "carol@test.com", "message": "Happy summer!", "dateTimeCreated": "2024-06-20T00:00:00-04:00", "id": 0, "domainEvents": [] }, { "emailAddress": "bob@test.com", "message": "Happy Valentine's Day!", "dateTimeCreated": "2024-02-14T00:00:00-05:00", "id": 0, "domainEvents": [] }, { "emailAddress": "alice@test.com", "message": "Hello world!", "dateTimeCreated": "2024-01-01T00:00:00-05:00", "id": 0, "domainEvents": [] } ]
```

Notes:

- Follow **"Make it work; make it right; make it fast."** Just hardcode things to get them working the first time
- In a real app: **Avoid exposing collections directly from your domain model types**
- In a real app: **Avoid using your domain or data model as your API model**
- **Look at your log messages when you hit the endpoint - what do they tell you?**

We'll correct these issues as we proceed through later labs. Don't assume anything in early labs is recommended or "correct"!

Lab 2: Adding and Persisting Entries

Expected time to complete: About 45 minutes. (Less if copy/pasting)

Goal

Add an endpoint that allows users to add new entries to the Guestbook.

Topics Used

Repository, Mediator. CQRS, Use Cases

Requirements

Update the List endpoint so it uses **Mediatr**, and persistence.

Add a new **Create** endpoint that adds new entries to persistence (which can immediately be seen via the List endpoint).

The Create endpoint expects an object with an **EmailAddress** and a **Message.**

Details

The CleanArchitecture template includes everything we need to pass messages using MediatR and to persist entities using a repository abstraction. You can see a working set of endpoints and handlers set up for Contributor in the Web and UseCases projects.

In this lab we are going to update the code for the List endpoint so that it sends a Query message via MediatR to one of our Use Cases in the UseCases project. When we're done, the List endpoint will closely resemble the one in the Contributors folder, and there will likewise be an Entries folder in the UseCases project that begins to mirror the Contributors folder's contents.

Note: Use Cases as a separate project is optional. In many apps it makes perfect sense to just work with the domain model (or make queries) directly from the UI or endpoints. Having them separate helps enforce CQRS and to reuse things like MediatR behaviors (for cross-cutting concerns like logging, timing, caching).

Step 1: Introducing MediatR

Our goal is to pull business and data access logic out of the Web project and keep it in our Core and UseCases projects. In addition, we intend to follow the Command Query Responsibility Segregation (CQRS) principle, so we will create specific messages for Commands and Queries and we will handle these messages in handlers defined in the UseCases project. The first use case we'll be tackling is for **Entries**, and we need to implement a **Query** that returns the most recent entries.

We already have working code from Lab 1 to return several entries - we'll start by using that code in the query handler, and then introduce real persistence. **Add all code in files to a namespace matching their folder path.**

1. In the UseCases project, add a new folder, **Entries**.
2. In the Entries folder, add a new *EntryDto.cs* file.

```
public record EntryDto(int Id, string EmailAddress, string Message,
DateTimeOffset DateTimeCreated);
```

3. In the Entries folder, add a new folder, **List**.
4. In the List folder, add a new ListEntriesQuery.cs file.

```
public record ListEntriesQuery(int? Skip, int? Take) :
      IQuery<Result<List<EntryDto>>>;
```

5. In the List folder, add a new interface, IListEntriesQueryService.cs with one method:

   ```
   Task<List<EntryDto>> ListAsync();
   ```

6. In the List folder, add a new *ListEntriesHandler.cs* file as a class.

   Inject in the IListEntriesQueryService and assign it to _queryService.

   Inherit from:
   ```
   IQueryHandler<ListEntriesQuery, Result<List<EntryDTO>>>
   ```

7. Implement the interface's Handle method. In it, call the List method on the query service and return the result.

Example (ListEntriesHandler.cs):

```
public class ListEntriesHandler(IListEntriesQueryService queryService)
  : IQueryHandler<ListEntriesQuery, Result<List<EntryDto>>>
{
  private readonly IListEntriesQueryService _queryService = queryService;

  public async Task<Result<List<EntryDto>>> Handle(ListEntriesQuery request,
    CancellationToken cancellationToken)
  {
    var entries = await _queryService.ListAsync();
    return Result<List<EntryDto>>.Success(entries);
  }
}
```

Now we have the use case defined - basically the handler just serves to delegate the request to a query service, which is responsible for the actual work. Let's implement the query service.

1. Open the Infrastructure project and go to Data/Queries.
2. Create a new file, FakeListEntriesQueryService.cs.
3. Copy the code that creates entries from Web/Entries/List.cs into the service (see next page)
4. Note that we need to make sure we return a `List<EntryDto>` and wrap it in a `Result`.
5. Since we only care about the successful case for now, use `Result.Success(entryList)`.

```
public class FakeListEntriesQueryService : IListEntriesQueryService
{
  public Task<Result<List<EntryDto>>> ListAsync()
  {
    var guestbook = new.Guestbook();

    guestbook.Entries.Add(new GuestbookEntry()
    {
      DateTimeCreated = new DateTime(2024, 1, 1),
      EmailAddress = "alice@test.com",
      Message = "Hello world!",
    });
    guestbook.Entries.Add(new GuestbookEntry()
    {
      DateTimeCreated = new DateTime(2024, 2, 14),
      EmailAddress = "bob@test.com",
      Message = "Happy Valentine's Day!",
    });
    guestbook.Entries.Add(new GuestbookEntry()
    {
      DateTimeCreated = new DateTime(2024, 6, 20),
      EmailAddress = "carol@test.com",
      Message = "Happy summer!",
    });
   var entryList = guestbook.Entries
      .OrderByDescending(e => e.DateTimeCreated)
      .Select(e => new EntryDto(e.Id, e.EmailAddress, e.Message, e.DateTimeCreated))
      .ToList();
    return Task.FromResult(Result.Success(entryList));
  }
```

Now to finish wiring this up, we need to inject IMediator into the List endpoint and return the result of the query. Essentially we've copied */Contributors/List.cs* endpoint code into */Entries/List.cs* endpoint code. We will also need to create an *EntryRecord.cs* and an *EntryListResponse.cs* file, all in the */Entries* folder in the **Web project**. EntryRecord will just happen to look very similar to EntryDto.

EntryRecord.cs

public record EntryRecord(int Id, string EmailAddress, string Message, DateTimeOffset DateTimeCreated);

List.EntryListResponse.cs (name it with the "List." prefix so it groups with the List.cs endpoint)

```
public class EntryListResponse
{
  public List<EntryRecord> Entries { get; set; } = new();
}
```

List.cs

```
public class List(IMediator mediator) : EndpointWithoutRequest<EntryListResponse>
{
  private readonly IMediator _mediator = mediator;
  public override void Configure()
  {
    Get("/entries");
    AllowAnonymous();
  }

  public override async Task HandleAsync(CancellationToken ct)
  {
    Result<List<EntryDto>> result = await _mediator.Send(new ListEntriesQuery(null, 
null));

    if (result.IsSuccess)
    {
      Response = new EntryListResponse
      {
        Entries = result.Value
                  .Select(e => new EntryRecord(e.Id, e.EmailAddress, e.Message, 
e.DateTimeCreated))
                  .ToList()
      };
    }
  }
}
```

Now we can test the application again. **We should get an error message** because we haven't wired up our query service interface to the `FakeListEntriesQueryService` implementation. To fix this, add the following to the **Infrastructure** project's *InfrastructureServiceExtensions.cs* (in the root of the project) just above the logger line.:

```
services.AddScoped<IListEntriesQueryService, FakeListEntriesQueryService>();
```

Finally we can test and it should work! You should see the same data you had previously - so why did we bother with UseCases and Mediator? Separation of concerns, for one, but we also got some real benefits.

Look at your log output again. Hit the endpoint. What kind of details do you see? Where is that coming from?

```
[14:18:00 INF] Request starting HTTP/1.1 GET https://localhost:57679/entries - null null
[14:18:00 INF] Executing endpoint 'HTTP: GET /entries'
[14:18:00 INF] Handling ListEntriesQuery
[14:18:00 INF] Property Skip : null
[14:18:00 INF] Property Take : null
[14:18:00 INF] Handled ListEntriesQuery with
Ardalis.Result.Result`1[System.Collections.Generic.List`1[Nimble.GuestbookApp.UseCases.Entries.EntryDto]] in 1 ms
[14:18:00 INF] Executed endpoint 'HTTP: GET /entries'
[14:18:00 INF] Request finished HTTP/1.1 GET https://localhost:57679/entries - 200 null application/json; charset=utf-8 39.7225ms
```

See:

https://github.com/ardalis/Ardalis.SharedKernel/blob/main/src/Ardalis.SharedKernel/LoggingBehavior.cs

```
public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TResponse> next,
CancellationToken cancellationToken)
{
    Guard.Against.Null(request);
    if (_logger.IsEnabled(LogLevel.Information))
    {
      _logger.LogInformation("Handling {RequestName}", typeof(TRequest).Name);


      // Reflection! Could be a performance concern
      Type myType = request.GetType();
      IList<PropertyInfo> props = new List<PropertyInfo>(myType.GetProperties());
      foreach (PropertyInfo prop in props)
      {
        object? propValue = prop?.GetValue(request, null);
        _logger.LogInformation("Property {Property} : {@Value}", prop?.Name, propValue);
      }
    }

    var sw = Stopwatch.StartNew();

    var response = await next();

    _logger.LogInformation("Handled {RequestName} with {Response} in {ms} ms", typeof(TRequest).Name,
response, sw.ElapsedMilliseconds);
    sw.Stop();
    return response;
  }
```

Step 2: Adding Persistence

Now we need to add EF Core support to our guestbook and its entries. First we need to add a Guestbook DbSet to our *AppDbContext* in the **Infrastructure** project (in the Data folder). Add this line:

public DbSet<Guestbook> Guestbooks => Set<Guestbook>();

Seed Data

We will need seed data to work with. Add this to *Infrastructure/SeedData.cs* at the end of the `PopulateTestData()` method:

```
var guestbook = new Guestbook();
guestbook.Name = "Default Guestbook";
dbContext.Guestbooks.Add(guestbook);
dbContext.SaveChanges();
```

```
var entry1 = new GuestbookEntry()
{
  DateTimeCreated = new DateTimeOffset(DateTime.Today),
  EmailAddress = "alice@test.com",
  Message = "Hello world!"
};
guestbook.Entries.Add(entry1);
dbContext.SaveChanges();
```

Next we need to implement a real query service in the *Data/Queries* folder in the Infrastructure project. Add a new class, *ListEntriesQueryService.cs*.

Here you have a choice:

1. Implement the query using Dapper and raw SQL. This requires a fair bit of code but demonstrates the separation of queries from commands, and the fact that queries don't need to use the domain model *at all*.
2. Use EF Core's DbSet and our Entity directly. This is simplest but does couple the query to the domain model. Many queries can (and should) be independent of the domain model, which is mainly used for mutation, not querying.
3. Use EF Core. This is simpler. Starting in .NET 8 it supports ad hoc queries like we prefer for this step. I'm not showing it in this lab but it's very similar to the Dapper approach and uses raw SQL.

Step 2a: Using Dapper

We'll add Dapper to support this, as well as SQLite. Here are the two packages we need to add to Infrastructure:

- `dotnet add package Microsoft.Data.Sqlite --version 8.0.8`
- `dotnet add package Dapper --version 2.1.35`

Adding a *DapperContext.cs* in the *Infrastructure/Data* folder is also helpful (see reference):
(https://code-maze.com/using-dapper-with-asp-net-core-web-api/)

```
public class DapperContext
{
    private readonly IConfiguration _configuration;
    private readonly string _connectionString;
    public DapperContext(IConfiguration configuration)
    {
        _configuration = configuration;
        _connectionString = _configuration.GetConnectionString("SqliteConnection")!;
    }
    public IDbConnection CreateConnection()
        => new SqliteConnection(_connectionString);
}
```

And add the following to *InfrastructureServiceExtensions.cs* (toward the bottom, before the log statement):

```
services.AddSingleton<DapperContext>();
```

In that same file, swap out the registration of FakeListEntriesQueryService with the new real one:

```
services.AddScoped<IListEntriesQueryService, ListEntriesQueryService>();
```

Now we can implement our query service. Inject in a `DapperContext` and use it in the `ListAsync()` method.

Define a sql query variable as "**SELECT Id, EmailAddress, Message, DateTimeCreated FROM GuestbookEntry"**

Use dapper's `CreateConnection()` method to create a connection, and then call `QueryAsync<EntryDto>` to get the query result, passing in the sql query. Return the result.

Queries/ListEntriesQueryService.cs

```
public class ListEntriesQueryService(DapperContext dapper) : IListEntriesQueryService
{
  private readonly DapperContext _dapper = dapper;

  public async Task<Result<List<EntryDto>>> ListAsync()
  {
    // configure Dapper types
    SqlMapper.ResetTypeHandlers();
    SqlMapper.AddTypeHandler(new DateTimeOffsetHandler());
    // check table name; might be plural
    var query = "SELECT Id, EmailAddress, Message, DateTimeCreated FROM
GuestbookEntry";
    using (var connection = _dapper.CreateConnection())
    {
      var entries = (await connection.QueryAsync<EntryDto>(query)).ToList();
      return entries;
    }
  }
}
```

We need the custom DateTimeOffset handler, found here:

https://learn.microsoft.com/en-us/dotnet/standard/data/sqlite/dapper-limitations

Paste the classes into the *DapperContext.cs* file. Note for SetValue you may need to change **T** to **T?** (I have a pull request pending to fix their sample).

You'll also need to convert *EntryDto.cs* to use a class rather than a record. Dapper can't materialize it as a record.

```
public class EntryDto
{
  public EntryDto()
  {
  }
  public EntryDto(int id, string emailAddress, string message, DateTimeOffset dateTimeCreated)
  {
    Id = id;
    EmailAddress = emailAddress;
    Message = message;
    DateTimeCreated = dateTimeCreated;
  }

  public int Id { get; }
  public string EmailAddress { get; } = string.Empty;
  public string Message { get; } = string.Empty;
  public DateTimeOffset DateTimeCreated { get; } = DateTimeOffset.Now;
}
```

You'll also need to run migrations - see below - in order to ensure the database has the correct schema.

If you've gotten this far you may want to rename your service *DapperListEntriesQueryService.cs.*

Step 2b: Using EF Core (instead of Dapper)

Using EF Core is much simpler. We'll still need to run the migrations listed below. But the query service is just:

```
public class ListEntriesQueryService(AppDbContext dbContext) : IListEntriesQueryService
{
  private readonly AppDbContext _dbContext = dbContext;

  public async Task<Result<List<EntryDto>>> ListAsync()
  {
    var firstGuestbook = await _dbContext.Guestbooks
                    .Include(g => g.Entries)
                    .FirstOrDefaultAsync();

    if(firstGuestbook is null) return new List<EntrytoO>();


    return firstGuestbook.Entries.Select(entry =>
      new EntryDto(entry.Id, entry.EmailAddress,
        entry.Message, entry.DateTimeCreated))
      .ToList();
  }
}
```

Add this to configure the service (in *InfrastructureServiceExtensions.cs*):

`services.AddScoped<IListEntriesQueryService, ListEntriesQueryService>();`

Consider renaming the type to EfListEntriesQueryService or similar.

If we run the app now, and hit the List endpoint, we should get a notification that there is no table GuestbookEntry. It's time to add and run a migration.

Migrations

We need to use migrations to configure the SqlLite database the application is using. You need to follow these steps in order to set up migrations:

1. Make sure you can build your solution. Migrations don't work if there are build errors.
2. Delete the **database.sqlite** file from the **Web** project folder (you probably need to **stop your app**).
3. Install/Update ef tools if needed:
    a. `dotnet tool install --global dotnet-ef`
    b. `dotnet tool update --global dotnet-ef`
4. Open a terminal window **in the Web project folder.**
5. Run this command to create an initial migration from a command prompt **in the Web project folder**:
```
dotnet ef migrations add InitialModel --context AppDbContext -p
../Nimble.GuestbookApp.Infrastructure/Nimble.GuestbookApp.Infrastructure.cspro
j -s Nimble.GuestbookApp.Web.csproj -o Data/Migrations
```
6. Run this command to apply the migration:
```
dotnet ef database update -c AppDbContext -p
../Nimble.GuestbookApp.Infrastructure/Nimble.GuestbookApp.Infrastructure.cspro
j -s Nimble.GuestbookApp.Web.csproj
```
7. **Troubleshooting:** Make sure Web references Microsoft.EntityFrameworkCore.Design (latest version). If you get net8 compatibility errors try rebuilding. Also make sure you are in the **Nimble.GestbookApp.Web** folder.

Note: If you don't have **dotnet ef** installed, get it here: https://learn.microsoft.com/en-us/ef/core/cli/dotnet

I recommend adding these two commands (for adding migrations and updating the database) to your project's README file - you'll probably need them again in a real app.

At this point we should have schema and seed data. You might need to delete your database files from the root of the web project, then run the app and hit the List Entries endpoint. You should see a single message from "alice" corresponding to your seed data.

```
{
  "entries": [
    {
      "id": 1,
      "emailAddress": "alice@test.com",
      "message": "Hello world!",
      "dateTimeCreated": "2024-10-03T00:00:00-04:00"
    }
  ]
}
```

Implement Adding Entries

To support adding new entries to the guestbook, you'll need a Create endpoint and Use Case. Here are the high level steps - code at the end. Model after the Create Contributor handlers and endpoint.

1. In the UseCases/Entries folder, add a new folder Create
2. In this folder, add a new CreateEntryCommand.cs file
3. Next add a CreateEntryHandler.cs file.
4. In the Web/Entries folder add a Create.cs file.
5. In the Web/Entries folder add 3 more files (named with a Create. prefix to group them):
    a. Create.CreateEntryRequest.cs
        i. Has route data and properties for EmailAddress and Message.
    b. Create.CreateEntryResponse.cs
        i. Returns the new entry Id, EmailAddress, and Message, as well as GuestbookId
    c. Create.CreateEntryValidator.cs
        i. Verifies EmailAddress and Message are not null or whitespace

Sample Code:

CreateEntryCommand.cs

```
public record CreateEntryCommand(string emailAddress, string message):
Ardalis.SharedKernel.ICommand<Result<int>>;
```

CreateEntryHandler.cs

```
public class CreateEntryHandler : ICommandHandler<CreateEntryCommand, Result<int>>
{
  private readonly IRepository<Guestbook> _guestbookRepo;

  public CreateEntryHandler(IRepository<Guestbook> guestbookRepo)
    {
    _guestbookRepo = guestbookRepo;
  }
  public async Task<Result<int>> Handle(CreateEntryCommand request,
    CancellationToken cancellationToken)
  {
    var guestbook = (await _guestbookRepo.ListAsync()).First();
    var newEntry = new GuestbookEntry()
    {
        EmailAddress = request.emailAddress,
        Message = request.message
    };
    guestbook.Entries.Add(newEntry);
    await _guestbookRepo.SaveChangesAsync();

    return newEntry.Id;
  }
}
```

Web/Create.cs

```
public class Create : Endpoint<CreateEntryRequest, CreateEntryResponse>
{
  private readonly IMediator _mediator;

  public Create(IMediator mediator)
  {
    _mediator = mediator;
  }


  public override void Configure()

  {

    Post(CreateEntryRequest.Route);

    AllowAnonymous();

  }


  public override async Task HandleAsync(

    CreateEntryRequest request,

    CancellationToken cancellationToken)

  {

    var result = await _mediator.Send(new CreateEntryCommand(request.EmailAddress!,
request.Message!));


    if(result.IsSuccess)

    {

      Response = new CreateEntryResponse()

      {

        Id = result.Value,

        EmailAddress = request.EmailAddress!,

        Message = request.Message!

      };

    }

  }

}
```

Create.CreateEntryRequest.cs

```
public class CreateEntryRequest
{
  public const string Route = "/entries";

  [Required]
  public string? EmailAddress { get; set; }
  [Required]
  public string? Message { get; set; }
}
```

Create.CreateEntryResponse.cs

```
public class CreateEntryResponse
{
  public int Id { get; set; }
  public string EmailAddress { get; set; } = "";
  public string Message { get; set; } = "";
}
```

Create.CreateEntryRequestValidator.cs

```
public class CreateEntryValidator : Validator<CreateEntryRequest>
{
  public CreateEntryValidator()
  {
    RuleFor(x => x.EmailAddress)
      .NotEmpty()
      .WithMessage("Email Address is required.")
      .MinimumLength(2)
      .MaximumLength(DataSchemaConstants.DEFAULT_NAME_LENGTH);
    RuleFor(x => x.Message)
      .NotEmpty()
      .WithMessage("Message is required.")
      .MinimumLength(2)
      .MaximumLength(DataSchemaConstants.DEFAULT_NAME_LENGTH);
  }
}
```

Run the application. Add a new entry to your *api.http* file to allow you to POST a new entry.

You should be able to add additional messages to the guestbook, and they should appear in the list of entries in the GET endpoint. If they're not being ordered most recent first, see if you can fix that now (the code above didn't include that logic). This should go into your query service.

Note: Since we're using a real data store, test data will accumulate as you work on these labs. Feel free to delete the database.sqlite file any time you want to reset the data. If you do, you will need to re-run the **dotnet ef database update** script noted above.

Updated api.http

```
@host=https://localhost
@port=57679

// List all guestbook entries
GET http://{{hostname}}:{{port}}/entries

###

// Create a new guestbook entry
POST http://{{hostname}}:{{port}}/entries
Content-Type: application/json
{
  "emailAddress": "abc@def.com",
  "message": "hello students!!!"
}

###
```

Discussion/Review Questions:

1. What do you think about this design so far?
2. Is it following separation of concerns?
3. Is it testable?
4. Is there anything you would refactor to improve it?
5. How is model validation working at this point?
6. Why do you think we have a Guestbook in the model, rather than just Entries?

Reminder: These labs demonstrate how to add functionality in an expedient, but not necessarily well-designed, way, at first. As they continue, you will refactor code from earlier labs to improve upon its design. If you think things aren't necessarily cleanly designed at this point, *I would agree with you* (though so far things are very simple so there's not a huge need for better design).

GitHub Notes – How to Catch Up If You're Behind

The labs build on one another, but if you fall behind you can jump to a starting point for each lab by using its branch. The easiest way to do this is to clone a new version of the lab from its source, and then jump to the appropriate lab using a branch. Save (and commit, if you're using git for your own work) your work. Then go to a new folder to clone the sample labs using these commands:

```
git clone https://github.com/NimblePros/GuestbookApp
```

```
git checkout net8-lab2
```

Using the above command should result in a solution that has just finished Lab 2 and is ready to begin Lab 3.

Lab 3: Notifying Users of New Entries

Expected time to complete: About 20-30 minutes. (Less if copy/pasting)

Goal

When a new entry is added to the guestbook, notify others about the new entry.

Topics Used

Domain Events

Requirements

Consider where to implement notification-sending functionality - there are many options:.

- **In Create.cs endpoint in Web**
- **In CreateHandler Use Case**
- **In a new service (presumably called by the Use Case handler)**
- **In Guestbook itself (in a new Add Entry method)**
- **Using Domain Events and a new handler**

There are pros and cons - tradeoffs - to each of these approaches but in the interest of time we are going to implement only the last option, which I think is one of the most flexible ones.. The template already has an `ISendEmail` interface with an `SmtpEmailSender` class we can use. Our high level steps are to:

- Refactor `Guestbook` to have an explicit `AddEntry` method
  - We will no longer expose the `List<GuestbookEntry>` directly. You should never do this from types you want to support encapsulation. See: **https://ardalis.com/avoid-collections-as-properties/**
- Create a new `EntryAddedEvent` type (in the GuestbookAggregate folder)
- Create an `EntryAddedEvent` instance when an entry is added and add it to our collection of events (inherited from `EntityBase`)
- Handle the event and send an email (where should this live? Ideally in GuestbookAggregate as well)

Guestbook.cs

```
public class Guestbook : EntityBase, IAggregateRoot
{
    public string Name { get; set;} = "";
    private List<GuestbookEntry> _entries  = new();
    public IEnumerable<GuestbookEntry> Entries => _entries.AsReadOnly();

    public void AddEntry(GuestbookEntry entry)
    {
        _entries.Add(entry);

        RegisterDomainEvent(new EntryCreatedEvent(entry));

    }

    public void SeedEntry(GuestbookEntry entry)
    {
        _entries.Add(entry);
    }
}
```

Change anything that previously invoked "guestbook.Entries.Add" to use `AddEntry` or `SeedEntry` instead. **Use SeedEntry in SeedData.cs and FakeEntryQueryService and AddEntry in the actual Create handler**. You can use the compiler to find all such instances. In a real app you wouldn't need `SeedEntry`.

Core/GuestbookAggregate/EntryCreatedEvent.cs

```
internal class EntryCreatedEvent : DomainEventBase
{
    public EntryCreatedEvent(GuestbookEntry entry)
    {
        Entry = entry;
    }
    public GuestbookEntry Entry { get; set;}
}
```

Core/GuestbookAggregate/NotifyCreatorEntryCreatedHandler.cs

```
internal class NotifyCreatorEntryCreatedHandler(
    ILogger<NotifyCreatorEntryCreatedHandler> logger,
    IEmailSender emailSender) : INotificationHandler<EntryCreatedEvent>
{
  private readonly ILogger<NotifyCreatorEntryCreatedHandler> _logger = logger;
  private readonly IEmailSender _emailSender = emailSender;

  public async Task Handle(EntryCreatedEvent domainEvent, CancellationToken
 cancellationToken)
  {
    _logger.LogInformation("Handling EntryCreatedEvent for {entryId}",
domainEvent.Entry.Id);

    await _emailSender.SendEmailAsync(
        domainEvent.Entry.EmailAddress,
        "donotreply@test.com",
        "Guestbook Entry Accepted",
        $"Your entry with message {domainEvent.Entry.Message} was received"
    );
 }
}
```

Now, if you want to see it work with a fake email sender, check *Web/Configurations/ServiceConfigs* and make sure this line is uncommented (and others are commented):

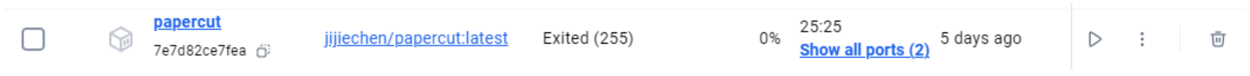
```
services.AddScoped<IEmailSender, FakeEmailSender>();
```

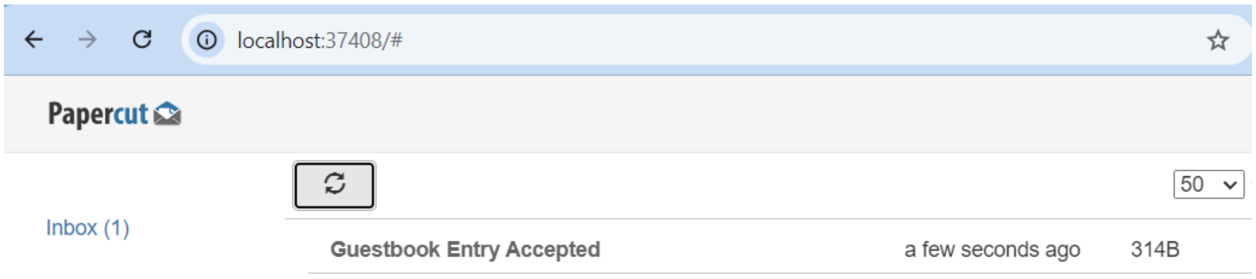
When you create a new entry, check the logs and you should see:
"Not actually sending an email to…"

If you want it to actually try to send mail, use the `MimeKitEmailSender` instead:

```
services.AddScoped<IEmailSender, MimeKitEmailSender>();
```

With a real email provider wired up, as long as you have Papercut running, emails should successfully send and you should be able to see them by going to http://localhost:37408/. If it's not working, make sure Papercut is started in Docker:


papercut 7e7d82ce7fea   jijiechen/papercut:latest   Exited (255)   0%   25:25 Show all ports (2)   5 days ago


localhost:37408/#

Papercut

Inbox (1)

Guestbook Entry Accepted	a few seconds ago	314B

Now's a good time to check your logic. Are you sending emails to every email address who has left an entry, except for the one who just added one? Or are you only sending an email to the person who just left an entry? They don't need an email - they know they just signed the guestbook!

How would we do this? In our handler we have the entry, but not the Guestbook. We don't have an easy way to access the *other* entries in the Guestbook.

We can update GuestbookEntry to add a GuestbookId.
Then we can inject a Repository<Guestbook> and call GetById on it to fetch the Guestbook.

But by default that won't include the sub-collection of Entries (it will be null). We have to call .Include(g => g.Entries) but a typical Repository interface doesn't expose that. Do we need to inject in a DbContext (ick)?

We'll see in the next lab! 😎

Lab 4: Using Specifications for Data Access

Expected time to complete: About 15 minutes.

Goal

Create a specification for fetching a Guestbook with all of its Entries. Use it to properly implement our email notification logic.

Topics Used

Specifications. Aggregates. Domain Events.

Requirements

When someone signs the guestbook, we want to send notifications emails to everyone who previously signed it, but not to the one who just signed it (even if they'd done so previously).

Details

We're going to update the NotifyCreatorEntryCreatedHandler.cs to do most of the work. We'll also be creating a new Specification in the GuestbookAggregate.

1. `NotifyCreatorEntryCreatedHandler` is now poorly named. Let's rename it `NotifyOtherSignersEntryCreatedHandler`.
2. Add a `public int GuestbookId` property to `GuestbookEntry`.
3. Create a new `GuestbookWithEntriesByIdSpec` class in the `GuestbookAggregate` folder.
    a. Inherit from `Specification<Guestbook>`
    b. Pass its constructor an `int id`
    c. In the constructor, add `Query.Where(g => g.Id == id);`
    d. And `Query.Include(g => g.Entries);`
4. Back in the handler, modify the handle method so it:
    a. Creates a GuestbookWithEntriesByIdSpec and the GuestbookId from the entry
    b. Calls the repository.FirstOrDefault(spec) method to get a Guestbook
    c. Fetch unique emails that aren't the most recent email from Guestbook.Entries into a list
    d. Loop over the list and send to each email address
5. Test your implementation

Sample files on next page.

GuestbookWithEntriesByIdSpec.cs

```
public class GuestbookWithEntriesByIdSpec : Specification<Guestbook>
{
  public GuestbookWithEntriesByIdSpec(int id)
  {
    Query.Where(g => g.Id == id);
    Query.Include(g => g.Entries);
  }
}
```

GuestbookEntry.cs

```
public class GuestbookEntry : EntityBase
{
  public int GuestbookId { get; set; }
  public string EmailAddress { get; set; } = string.Empty;
  public string Message { get; set; } = string.Empty;
  public DateTimeOffset DateTimeCreated { get; set; } = DateTimeOffset.Now;
}
```

NotifyOtherSignersEntryCreatedHandler.cs:
```
internal class NotifyOtherSignersEntryCreatedHandler(
    IReadRepository<Guestbook> repository,
    ILogger<NotifyOtherSignersEntryCreatedHandler> logger,
    IEmailSender emailSender) : INotificationHandler<EntryCreatedEvent>
{
  private readonly ILogger<NotifyOtherSignersEntryCreatedHandler> _logger = logger;
  private readonly IEmailSender _emailSender = emailSender;

  public async Task Handle(EntryCreatedEvent domainEvent, CancellationToken
cancellationToken)
  {
    _logger.LogInformation("Handling EntryCreatedEvent for {entryId}",
domainEvent.Entry.Id);

    var spec = new GuestbookWithEntriesByIdSpec(domainEvent.Entry.GuestbookId);
    var guestbook = await repository.FirstOrDefaultAsync(spec);
    if (guestbook is null)
    {
      _logger.LogWarning("Guestbook not found for EntryCreatedEvent with id
{entryId}",
        domainEvent.Entry.Id);
      return;
    }

    var otherSignersEmailAddresses = guestbook.Entries
      .Where(e => e.EmailAddress != domainEvent.Entry.EmailAddress)
      .Select(e => e.EmailAddress)
      .Distinct()
      .ToList();

    foreach (var emailAddress in otherSignersEmailAddresses)
    {
        await _emailSender.SendEmailAsync(
          emailAddress,
          "donotreply@test.com",
          "Guestbook Entry Accepted",
          $"Your entry with message {domainEvent.Entry.Message} was received");
    }
  }
}
```

Lab 5: River Books - Books Module

Expected time to complete: About 30 minutes. (Less if copy/pasting)

Goal

Create a simple web application for the single domain model of Books. Create API endpoints to interact with the book entries. Use a modular design so the Books module is independent of the rest of the application.

Topics Used

Modular Monoliths, Entities, Mapping, Request-Endpoint-Response (REPR) pattern

Requirements

Create a simple web API that provides for management of books to be sold by a store. Ensure the design follows separation of concerns internally (book business logic, data access, and UI logic should all be separate at minimum) as well as modularity (Book functionality should be separate from other future functionality).

Details

Solution Setup

1. Create a folder *src* and navigate to it in a terminal window.
2. Create the project solution:
   ```
   dotnet new sln -n RiverBooks
   ```
3. Create the Web Api project and add it to the solution:
   ```
   dotnet new webapi -n RiverBooks.Web
   dotnet sln RiverBooks.sln add .\RiverBooks.Web\RiverBooks.Web.csproj
   ```
4. Create the *Books* module and add it to the solution:
   ```
   dotnet new classlib -n RiverBooks.Books -o Books\RiverBooks.Books
   dotnet sln RiverBooks.sln add .\Books\RiverBooks.Books\RiverBooks.Books.csproj
   ```
5. Open the solution in your IDE.
6. Find the *Web* project and add a reference to the *Books* project.
   a. ```
      dotnet add .\RiverBooks.Web\RiverBooks.Web.csproj reference
      .\Books\RiverBooks.Books\RiverBooks.Books.csproj
      ```

**Note:** Unless otherwise specified, all classes added to *Module* projects should have i*nternal* access instead of *public.* Classes in *Contract* projects should have *public* access.
      ```

Web Project Setup

1. Open *Properties/launchSettings.json* and find the "https" section
2. Change the "*applicationUrl*" to "*https://localhost:7179*"
3. Delete launchBrowser and launchUrl. We'll be using .http files for testing.
4. Update the *RiverBooks.Web.http* file so use this URI (note: http**s**)

```
@HostAddress = https://localhost:7179

GET {{HostAddress}}/books
Accept: application/json

###
```

Books Module - Initial Setup

The changes will be applied to the *RiverBooks.Books* project.

1. Add a *FrameworkReference* to Asp.NetCore.App framework to a new *ItemGroup* in the csproj file. (reference)

```
<ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App"/>
</ItemGroup>
```

2. Create files in the root of the RiverBooksBooks project :
    a. *BookDto* should include
        i. Guid Id
        ii. string Title
        iii. string Author
        iv. decimal Price
    b. All properties should be settable and a constructor should take all 4 in and assign the values to the properties
    c. *IBookService*

```
internal interface IBookService
{
  List<BookDto> ListBooks();
}
```

    d. *BookService* (method to return hard coded list)

```
internal class BookService : IBookService
{
  public List<BookDto> ListBooks()
  {
    return [
      new BookDto(Guid.NewGuid(), "The Fellowship of the Ring", "J.R.R. Tolkien",
1m),
      new BookDto(Guid.NewGuid(), "The Two Towers", "J.R.R. Tolkien", 2m),
      new BookDto(Guid.NewGuid(), "The Return of the King", "J.R.R. Tolkien", 3m)
    ];
  }
}
```

Books Module - List Books Endpoint

1. Add the *FastEndpoints* nuget package to the **RiverBooks.Books** project.
2. Create extension method to wire up Books module (in the root of Books project)

```
public static class BooksServiceExtensions
{
  public static IServiceCollection AddBooksServices(this IServiceCollection services)
  {
    services.AddScoped<IBookService, BookService>();
    return services;
  }
}
```

3. Add the *ListBooksResponse* class (in RiverBooks.Books).
    a. It should contain a property *Books* which is a List of *BookDto.*
4. Create the *ListBooksEndpoint* to return a list of books.
    a. It should inherit from *EndpointWithoutRequest<ListBooksResponse>.*
    b. The route should be "/books" and it should allow anonymous access.
    c. Override *HandleAsync* to do the following:
        i. Get a list of books from *IBookService*
        ii. Return *ListBooksReponse* and set the *Books* property
5. See sample file later in this lab

6. Test with .http file

In the Web Project:

1. In *Program.cs* (of the **Web** project)
    a. Remove everything related to the Weather Api (summaries, endpoint, record)
2. In *Program.cs*, wire up FastEndpoints and the Books Module. You may need to build first and you'll need to add `using RiverBooks.Books`.

```
builder.Services.AddFastEndpoints();
......
builder.Services.AddBooksServices();
.........
app.UseFastEndpoints();
```

Sample Files

ListBooksEndpoint.cs

```
internal class ListBooksEndpoint : EndpointWithoutRequest<ListBooksResponse>
{
    private readonly IBookService _bookService;
    public ListBooksEndpoint(IBookService bookService)
    {
        _bookService = bookService;
    }
    public override void Configure()
    {
        Get("/books");
        AllowAnonymous();
    }

    public override async Task HandleAsync(CancellationToken ct)
    {
       var books = _bookService.ListBooks();

        await SendAsync(new ListBooksResponse()
        {
            Books = books
        });
    }
 }
```

Books Module - Adding Persistence

Add entity and extend the DTO:

1. Add Guard clause package: *Ardalis.GuardClauses*
2. Add *Book* Entity with the following fields:
    a. public Guid Id
    b. public string Title
    c. public string Author
    d. public decimal Price
3. Add a method to *UpdatePrice* to the entity

Adding Persistence Abstraction

4. Add interfaces for the repository class:

```
internal interface IBookRepository : IReadOnlyBookRepository
{
  Task AddAsync(Book book);
  Task DeleteAsync(Book book);
  Task SaveChangesAsync();
}
```

And

```
internal interface IReadOnlyBookRepository
{
  Task<Book?> GetByIdAsync(Guid id);
  Task<List<Book>> ListAsync();
}
```

Updating the Book Service

1. Change *BookService* and its interface to delegate to the *IBookRepository* instead of using a hard coded list.
2. The method should be:
    a. GetBookByIdAsync - fetch a book from the repository and return a new matching instance of *BookDto.*
    b. CreateBookAsync - accept a BookDto, create a new *Book* entity, add it to the repository and save.
    c. DeleteBookAsync - accept an Id of a book, get a book from the repository; if it’s not null delete it from the repository and save.
    d. ListBooksAsync - pull all books from the repository and return a list of *BookDto.*
    e. UpdateBookPriceAsync - accepts an Id for a book and a new price, fetches the book from the repository, updates the price and saves.

Using EntityFramework Core.
Here we will add a concrete implementation of the repository using EF Core

3. Add the following nuget packages:
    a. To *RiverBooks.Books: Microsoft.EntityFrameworkCore.SqlServer*
    b. To *RiverBooks.Web*: Microsoft.EntityFrameworkCore.Design
4. Create the *BookDbContext* class:

```
internal class BookDbContext(DbContextOptions<BookDbContext> options) :
DbContext(options)
{
  internal DbSet<Book> Books { get; set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.HasDefaultSchema("Books");
    modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
  }

  protected override void ConfigureConventions(
    ModelConfigurationBuilder configurationBuilder)
  {
    configurationBuilder.Properties<decimal>()
      .HavePrecision(18, 6);
  }
}
```

5. Create the *BookConfiguration* file used to configure properties for the *Book* entity.
   It should:
    a. Set appropriate widths on the Title and Author fields; and also make them required.
    b. Use *HasData* to seed at least 3 books into the database.

```
internal class BookConfiguration : IEntityTypeConfiguration<Book>
{
  internal static readonly Guid Book1Guid = new
Guid("A89F6CD7-4693-457B-9009-02205DBBFE45");
  internal static readonly Guid Book2Guid = new
Guid("E4FA19BF-6981-4E50-A542-7C9B26E9EC31");
  internal static readonly Guid Book3Guid = new
Guid("17C61E41-3953-42CD-8F88-D3F698869B35");

  public void Configure(EntityTypeBuilder<Book> builder)
  {
    builder.Property(p => p.Title)
      .HasMaxLength(DataSchemaConstants.DEFAULT_NAME_LENGTH)
      .IsRequired();

    builder.Property(p => p.Author)
      .HasMaxLength(DataSchemaConstants.DEFAULT_NAME_LENGTH)
      .IsRequired();
    builder.HasData(GetSampleBookData());
  }

  private IEnumerable<Book> GetSampleBookData()
  {
    var tolkien = "J.R.R. Tolkien";
    yield return new Book(Book1Guid, "The Fellowship of the Ring", tolkien, 10.99m);
    yield return new Book(Book2Guid, "The Two Towers", tolkien, 11.99m);
    yield return new Book(Book3Guid, "The Return of the King", tolkien, 12.99m);
  }
}
```

6. Create the *EfBookRepository* class as a concrete implementation of *IBookRepository*.
    a. It should delegate its tasks to *BookDbContext.*
7. Modify *BooksServiceExtensions.AddBooksServices* to configure EF Core
    a. Modify signature to pass an instance of *IConfiguration* from *builder.Configuration* in Program.cs
    b. Get *BooksConnectionString* from configuration.
    c. Wire up EF Core with *AddDbContext* and *UserSqlServer*.
    d. Register *EfBookRepository.cs* to the interface appropriately.

Don't forget to add the connection string to appsettings.json:

```
 "ConnectionStrings": {
    "BooksConnectionString": "Server=(localdb)\\mssqllocaldb;Integrated
Security=true;Initial Catalog=RiverBooks;"
  },
```

Entity Framework Migrations
Note: Run all commands from the *RiverBooks.Books* folder.

1. Make sure ef tools are installed and updated:
   ```
   dotnet tool install --global dotnet-ef
   ```

2. Make sure the Web project has a nuget reference to *Microsoft.EntityFrameworkCore.Design*
3. Add an *Initial* migration.

   ```
   dotnet ef migrations add Initial -c BookDbContext -s
   ..\..\RiverBooks.Web\RiverBooks.Web.csproj -o Data/Migrations
   ```

4. Apply that migration to the database.

   ```
   dotnet ef database update -c BookDbContext -s
   ..\..\RiverBooks.Web\RiverBooks.Web.csproj
   ```

5. Test *List* endpoint again using http request file.

Books Module - Adding other Endpoints

1. Refactor the *List* endpoint as follows:
    a. Move to folder *BookEndpoints*.
    b. Optional:
        i. Rename the endpoint to *List.cs*
        ii. Rename the response to *List.ListBooksResponse.cs*
        iii. Turn on *Web* view in Visual Studio solution explorer settings.

Create the following endpoints with their matching request objects, endpoint files, and validators (where applicable).

GetById

- Request: *GetBookByIdRequest*
    - With a Guid Id property
- Endpoint
    - inherit from *Endpoint<GetBookByIdRequest, BookDto>*
    - Route: "/books/{Id}" with anonymous access.
    - It should fetch a book from *BookService* based on the Id and return the result.
    - Should return 404 not found if the book isn't found.

Create

- Request: *CreateBookRequest* will all the properties needed to create a book
- Endpoint:
    - Inherit from *Endpoint<CreateBookRequest, BookDto>*
    - Route: "books" with anonymous access.
    - Should create a *BookDto* from the request passed in, delegate to *BookService.CreateBookAsync()*. And it should return a 201 *created at* with the appropriate location.
- Validator:
    - Ensures the Title and Author fields are required and that there is a positive price:

```
internal class CreateBookRequestValidator : Validator<CreateBookRequest>
{
  public CreateBookRequestValidator()
  {
    RuleFor(x => x.Title)
      .NotNull()
      .NotEmpty()
      .WithMessage("A book title is required.");

    RuleFor(x => x.Author)
      .NotNull()
      .NotEmpty()
      .WithMessage("A book author is required.");

    RuleFor(x => x.Price)
      .GreaterThanOrEqualTo(0m)
      .WithMessage("Book prices must be positive values.");
  }
}
```

Delete

- Request: *DeleteBookRequest* with a Guid Id
- Endpoint:
  - Inherit from *Endpoint<DeleteBookRequest>*
  - Route: “/books/{Id}” with anonymous access.
  - Should deleted delete to the *BookService*

UpdatePrice

- Request: *UpdateBookPriceRequest* with *Guid Id* and *decimal NewPrice*
- Endpoint:
  - Inherit from: *Endpoint<UpdateBookPriceRequest, BookDto>*
  - Route: *"/books/{Id}/pricehistory"* with anonymous access.
  - Should delegate to *BookService.UpdateBookPriceAsync*, then get the updated book and return the result.
- Validator
  - *UpdateBookPriceRequestValidator*
  - Ensure that Id is required and the new price is positive.

Test the endpoints using the .http file:

```
@BookID=a89f6cd7-4693-457b-9009-02205dbbfe45
GET {{HostAddress}}/books/{{BookID}}
Accept: application/json

###

POST {{HostAddress}}/books
Accept: application/json
Content-Type: application/json

{
  "id": "b89f6cd7-4693-457b-9009-02205dbbfe45",
  "title": "Modular Monoliths - Getting Started",
  "author": "Steve Smith",
  "price": 29.99
}

###

# Test the location header
GET {{HostAddress}}/books/b89f6cd7-4693-457b-9009-02205dbbfe45
Accept: application/json
```

```
###

DELETE {{HostAddress}}/books/b89f6cd7-4693-457b-9009-02205dbbfe45
Accept: application/json

###

POST {{HostAddress}}/books/b89f6cd7-4693-457b-9009-02205dbbfe45/pricehistory
Accept: application/json
Content-Type: application/json

{
  "newPrice": -9.99
}
```

2. File and Folder Refactoring
   a. Refactor these new endpoints like we did with *List.*
   b. Create *Data* folder and move the data related files there:
      i. BookConfiguration.cs
      ii. BookDbContext.cs
      iii. DataSchemaConstants.cs
      iv. EfBookRepository.cs

Note: Some IDEs will automatically adjust the namespaces on the classes as you move them around. Others might not and an additional step might be needed.

Books Module - Adding Unit Tests

1. Add a new xUnit test project to the books folder: RiverBooks.Books.Tests
2. Add the following nuget packages:
   a. FluentAssertions
   b. FastEndpoints.Testing
3. Add a reference to the Books.Web project
4. In Program.cs add a partial class for *Program*
5. Create a folder for *Endpoints*
6. Add the Fixture class for testing FastEndpoints
7. Add the *BookList* test class to hit the *List* endpoint.
   a. Note that this is hitting the primary database.
8. Add specific appsettings.Testing.json and add test Db
   a. `"BooksConnectionString": "Server=(localdb)\\mssqllocaldb;Integrated Security=true;Initial Catalog=RiverBooks.Testing;"`
9. `dotnet ef database update -p .\Books\RiverBooks.Books\RiverBooks.Books.csproj -s .\RiverBooks.Web\RiverBooks.Web.csproj -- --environment Testing`
   a. learn.microsoft.com/en-us/ef/core/cli/dotnet#aspnet-core-environment

Lab 5 Samples

Books\RiverBooks.Books\BookConfiguration.cs

```
internal class BookConfiguration : IEntityTypeConfiguration<Book>
{
  internal static readonly Guid Book1Guid = new
Guid("A89F6CD7-4693-457B-9009-02205DBBFE45");
  internal static readonly Guid Book2Guid = new
Guid("E4FA19BF-6981-4E50-A542-7C9B26E9EC31");
  internal static readonly Guid Book3Guid = new
Guid("17C61E41-3953-42CD-8F88-D3F698869B35");

  public void Configure(EntityTypeBuilder<Book> builder)
  {
    builder.Property(p => p.Title)
      .HasMaxLength(DataSchemaConstants.DEFAULT_NAME_LENGTH)
      .IsRequired();

    builder.Property(p => p.Author)
      .HasMaxLength(DataSchemaConstants.DEFAULT_NAME_LENGTH)
      .IsRequired();

    builder.HasData(GetSampleBookData());
  }

  private IEnumerable<Book> GetSampleBookData()
  {
    var tolkien = "J.R.R. Tolkien";
    yield return new Book(Book1Guid, "The Fellowship of the Ring", tolkien, 10.99m);
    yield return new Book(Book2Guid, "The Two Towers", tolkien, 11.99m);
    yield return new Book(Book3Guid, "The Return of the King", tolkien, 12.99m);
  }
}
```

Lab 6: Books Module using *Modulith*

Goal

Reimplement the Books module using the *Ardalis.Modulith* package.

Topics Used

Solution templates, Modular Monoliths

Requirements

Use a solution template to set up the modular app we added in lab 5. We'll reuse most of the implementation code.

Details

1. Make sure the Ardalis.*Modulith* templates are installed (v1.1.2):
   ```
   dotnet new install Ardalis.Modulith
   ```
2. Move the *src* folder somewhere safe. We will be copying and pasting the code files, so don't delete them.
3. Recreate the application using the new template

   ```
   dotnet new modulith -n RiverBooks --with-module Books -o src
   ```

   NOTE: You can choose different names but it may make your life hard when it comes to the namespaces…
4. Copy the following files in the *src* folder from your safe location. Override the files wherever you are prompted.

   Note: If you drag these files from explorer into your IDE, it *should* include the files in the project appropriately.
   a. RiverBooks.Books project
      i. Data Folder
      ii. Endpoints Folder
      iii. Books.cs
      iv. BookDto.cs
      v. BookService.cs
      vi. IBookRepository.cs
      vii. IBookServer.cs
   b. RiverBooks.Books.Tests project
      i. Endpoints Folder
   c. RiverBooks.Web.http (to RiverBooks.Web)
   d. launchSettings.json (to RiviberBooks.Web/Properties)
   e. appSettings.json, appSettings.Development.json and appSettings.Testing.json
5. Ensure the appropriate nuget packages are re-added to the projects. You can do this by copying the list straight from one *csproj* file to another
   a. RiverBooks.Web
      i. Microsoft.EntityFrameworkCore.Design
   b. RiverBooks.Books
      i. Ardalis.Guard
      ii. Microsoft.EntityFrameworkCore.SqlServer
   c. RiverBooks.Books.Tests

            i. FastEndpoints.Testing
            ii. FluentAssertions

6. Delete files that we’re not going to use from the Books module:
    a. RiverBooks.Books
        i. IWeatherForecastService
        ii. ServerWeatherForecastService
        iii. WeatherForeCastEndpoint
7. Recreate the following modifications:
    a. In RiverBooks.Books.Tests add a project reference to RiverBooks.Web
    b. In RiverBooks.Web add the *InternalsVisibleTo* item group.
8. Re-register the Books module

    Note that *BooksServiceExtensions* has been replaced by *BooksModuleServiceRegistrar* and some small modifications will need to be made
    a. Go to the *BooksServiceExtention.cs* copy the code in the method and add it to *BooksModuleServiceRegistrar.ConfigureServices*.

You can now build and run the web application, like before.

Lab 7: Adding Users Module

Goal

Add a Users module that interacts with the Books module.

Topics Used

Entities, Mapping, Request-Endpoint-Response (REPR) pattern, Modular Monoliths

Requirements

Details

Adding User Module

1. Go to the *src* folder and run the following command:
   ```
   dotnet new modulith --add basic-module --with-name Users --to RiverBooks
   ```
2. Delete the following unused files:
   a. *RiverBooks.Users* project
      i. IWeatherForecastService.cs
      ii. ServerWeatherForecastService.cs
      iii. Users.cs
      iv. WeatherForeCastEndpoint.cs
3. Add the following nuget packages to the *RiverBooks.Users* project:
   a. Microsoft.EntityFrameworkCore.SqlServer
   b. Microsoft.AspNetCore.Identity.EntityFrameworkCore
   c. Ardalis.GuardClauses
   d. FastEndpoints.Security
4. Add a reference to the *RiverBooks.Users* project on the *RiverBooks.Users.Web* project.
5. Create the following files in the Users Module:
   a. IApplicationUserRepository.cs with the following methods:

      ```
      Task<ApplicationUser> GetUserWithCartByEmailAsync(string email);
      Task SaveChangesAsync();
      ```

   b. CartItem.cs with the following properties:

```
public Guid Id { get; private set; } = Guid.NewGuid();
public Guid BookId { get; private set; }
public string Description { get; private set; } = string.Empty;
public int Quantity { get; private set; }
public decimal UnitPrice { get; private set; }
```

   c. ApplicationUser.cs
   d. Data Folder:
      i. UsersDbContext.cs
      ii. CartItemConfiguration.cs
      iii. EfApplicationUserRepository.cs

6. Go to *UsersModuleServiceRegistrar.cs* and register the Users module for EF Core, AspIdentity and the *EfApplicationUserRepository* class.
7. Add the connection string (*UsersConnectionString*) for the *Users* module the *appsetings.json* and *appsettings.Testing.json* files. These will point to the same database.
8. Migrate database (run from the *RiverBooks.Users* folder):
    a. **`dotnet ef migrations add Initial -c UsersContext -s ..\..\RiverBooks.Web\RiverBooks.Web.csproj -o Data/Migrations`**

    b. **`dotnet ef database update -c UsersContext  -s ..\..\RiverBooks.Web\RiverBooks.Web.csproj`**

Add User Registration and Login Endpoints

1. In the *UserEndpoints* folder create the following files:
    a. Create.CreateUserRequest.cs
    b. Create.cs (for the endpoint file).
2. Test the endpoint by posting:
    a. email: steve@test.com
    b. password: Pass@word1
3. Turn off HTTPS redirection in Development
4. In the User*Endpoints* folder create the following files:
    a. Login.UserLoginRequest.cs
    b. Login.cs (for the endpoint file).
5. Add The secret for the Jwt to the appsettings.config file
6. Test the endpoint by posting:
    a. email: steve@test.com
    b. password: Pass@word1

Linking Users and Books via Cart

1. Add the *Ardalis.Result* nuget package to the Users Module.
2. Add the *AddItem* use case to the UseCases folder.
    a. AddItemToCartCommand.cs
    b. AddItemToCartHandler.cs
3. In *CartEndpoints* folder create the request and endpoint file for AddItem:
    a. AddItem.AddCartItemRequest.cs
    b. AddItem.cs
4. Add Endpoint - AddCartItemRequest
    a. AddItem.AddCartItemRequest.cs
    b. Requires authentication (claim - EmailAddress)
    c. Use MediatR to invoke AddItemToCartCommand
    d. Use Result pattern (add Ardalis.Result)
5. Add Endpoint - ListCartItems
    a. CartItemDto
    b. CartResponse
    c. MediatR query: ListCartItemsQuery
    d. Add configuration for CartItem
        i. Never generate Id
6. Test endpoints (post and get)

Module Communication

1. In the *RiverBooks.Books.Contracts* project, add the following nuget packages:

        a. Ardalis.Result
        b. MediatR.Contracts
2. In the *RiverBooks.Books* project:
    a. add a reference to *RiberBooks.Books.Contracts.*
    b. Add Handler in Books Module
        i. *BookDetailsQueryHandler* in a folder called *Integrations*
3. In the *RiverBooks.Users* project:
    a. add a reference to *RiberBooks.Books.Contracts.*
    b. Go to AddItemToCartHandler
    c. Use MediatR to run the BookDetailsQuery to get the book details from the Books module.

Lab 7 Samples

Users/RiverBooks.Users/IApplicationUserRepository.cs

```
public interface IApplicationUserRepository
{
  Task<ApplicationUser> GetUserWithCartByEmailAsync(string email);
  Task SaveChangesAsync();
}
```

Users/RiverBooks.Users/CartItem.cs

```
public class CartItem
{
  public CartItem(Guid bookId, int quantity, decimal unitPrice, string description)
  {
    BookId = Guard.Against.Default(bookId);
    Quantity = Guard.Against.Negative(quantity);
    UnitPrice = Guard.Against.Negative(unitPrice);
    Description = Guard.Against.NullOrEmpty(description);
  }

  private CartItem()
  {
    // EF
  }

  public Guid Id { get; private set; } = Guid.NewGuid();
  public Guid BookId { get; private set; }
  public int Quantity { get; private set; }
  public string Description { get; private set; } = string.Empty;
  public decimal UnitPrice { get; private set; }

  public void AdjustQuantity(int quantity)
  {
    Quantity = Guard.Against.Negative(quantity);
  }

  public void AdjustUnitPrice(decimal unitPrice)
  {
    UnitPrice = Guard.Against.Negative(unitPrice);
  }

  public void UpdateDescription(string newDescription)
  {
    Description = Guard.Against.NullOrEmpty(newDescription);
  }
}
```

Users/RiverBooks.Users/ApplicationUser.cs

```
public class ApplicationUser : IdentityUser, IHaveDomainEvents
{
  public string FullName { get; set; } = string.Empty;

  private readonly List<CartItem> _cartItems = new();
  public IReadOnlyCollection<CartItem> CartItems => _cartItems.AsReadOnly();

  private readonly List<UserStreetAddress> _addresses = new();
  public IReadOnlyCollection<UserStreetAddress> Addresses => _addresses.AsReadOnly();

  private List<DomainEventBase> _domainEvents = new();
  [NotMapped]
  public IEnumerable<DomainEventBase> DomainEvents => _domainEvents.AsReadOnly();

  protected void RegisterDomainEvent(DomainEventBase domainEvent) => _domainEvents.Add(domainEvent);
  void IHaveDomainEvents.ClearDomainEvents() => _domainEvents.Clear();

  public void AddItemToCart(CartItem item)
  {
    Guard.Against.Null(item);

    var existingBook = _cartItems.SingleOrDefault(c => c.BookId == item.BookId);
    if (existingBook != null)
    {
      existingBook.AdjustQuantity(existingBook.Quantity + item.Quantity);

      existingBook.AdjustUnitPrice(item.UnitPrice);
      existingBook.UpdateDescription(item.Description);
      return;
    }
    _cartItems.Add(item);
  }

  internal UserStreetAddress AddAddress(Address address)
  {
    Guard.Against.Null(address);

    // find existing address and just return it
    var existingAddress = _addresses.SingleOrDefault(a => a.StreetAddress == address);
    if (existingAddress != null)
    {
      return existingAddress;
    }

    var newAddress = new UserStreetAddress(Id, address);
    _addresses.Add(newAddress);

    RegisterDomainEvent(new AddressAddedEvent(newAddress));
    return newAddress;
  }

  internal void ClearCart()
  {
    _cartItems.Clear();
  }
}
```

Users/RiverBooks.Users/Data/UsersDbContext.cs

```
internal class UsersDbContext : IdentityDbContext
{
  private readonly IDomainEventDispatcher? _dispatcher;

  public UsersDbContext(DbContextOptions<UsersDbContext> options,
    IDomainEventDispatcher? dispatcher) : base(options)
  {
    _dispatcher = dispatcher;
  }

  public DbSet<ApplicationUser> ApplicationUsers { get; set; }
  public DbSet<UserStreetAddress> UserStreetAddresses { get; set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.HasDefaultSchema("Users");

    modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());

    base.OnModelCreating(modelBuilder);
  }

  protected override void ConfigureConventions(
  ModelConfigurationBuilder configurationBuilder)
  {
    configurationBuilder.Properties<decimal>()
        .HavePrecision(18, 6);
  }

  /// <summary>
  /// This is needed for domain events to work
  /// </summary>
  /// <param name="cancellationToken"></param>
  /// <returns></returns>
  public override async Task<int> SaveChangesAsync(CancellationToken
cancellationToken = new CancellationToken())
  {
    int result = await
base.SaveChangesAsync(cancellationToken).ConfigureAwait(false);
```

```
    // ignore events if no dispatcher provided
    if (_dispatcher == null) return result;

    // dispatch events only if save was successful
    var entitiesWithEvents = ChangeTracker.Entries<IHaveDomainEvents>()
        .Select(e => e.Entity)
        .Where(e => e.DomainEvents.Any())
    .ToArray();

    await _dispatcher.DispatchAndClearEvents(entitiesWithEvents);

    return result;
  }
}
```

Users/RiverBooks.Users/Data/EfApplicationUserRepository.cs

```
internal class EfApplicationUserRepository : IApplicationUserRepository
{
  private readonly UsersDbContext _dbContext;

  public EfApplicationUserRepository(UsersDbContext dbContext)
  {
    _dbContext = dbContext;
  }

  public Task<ApplicationUser> GetUserByEmailAsync(string email)
  {
    return _dbContext.ApplicationUsers
      .SingleAsync(u => u.Email == email);
  }

  public Task<ApplicationUser> GetUserByIdAsync(Guid userId)
  {
    return _dbContext.ApplicationUsers
      .SingleAsync(u => u.Id == userId.ToString());
  }

  public Task<ApplicationUser> GetUserWithAddressesByEmailAsync(string email)
  {
    return _dbContext.ApplicationUsers
      .Include(user => user.Addresses)
      .SingleAsync(u => u.Email == email);
  }

  public Task<ApplicationUser> GetUserWithCartByEmailAsync(string email)
  {
    return _dbContext.ApplicationUsers
      .Include(user => user.CartItems)
      .SingleAsync(u => u.Email == email);
  }

  public async Task SaveChangesAsync()
  {
    await _dbContext.SaveChangesAsync();
  }
}
```

Users/RiverBooks.Users/Data/CartItemConfiguration.cs

```
public class CartItemConfiguration : IEntityTypeConfiguration<CartItem>
{
  public void Configure(EntityTypeBuilder<CartItem> builder)
  {
    builder.Property(x => x.Id)
      .ValueGeneratedNever();

    builder.Property(x => x.Description)
      .HasMaxLength(100)
      .IsRequired();
  }
}
```

Users/RiverBooks.Users/UsersModuleServiceRegistrar.cs

```
public static class UsersModuleServicesExtensions
{
  public static IServiceCollection AddUsersModuleServices(this IServiceCollection
services,
    ConfigurationManager config,
    ILogger logger,
    List<System.Reflection.Assembly> mediatRAssemblies)
  {
    string? connectionString = config.GetConnectionString("UsersConnectionString");
    services.AddDbContext<UsersDbContext>(config =>
      config.UseSqlServer(connectionString));

    services.AddIdentityCore<ApplicationUser>()
            .AddEntityFrameworkStores<UsersDbContext>();

    services.AddScoped<IApplicationUserRepository, EfApplicationUserRepository>();
    services.AddScoped<IReadOnlyUserStreetAddressRepository,
EfUserStreetAddressRepository>();

    // if using MediatR in this module, add any assemblies that contain handlers to
the list
    mediatRAssemblies.Add(typeof(UsersModuleServicesExtensions).Assembly);

    logger.Information("{Module} module services registered", "Users");
    return services;
  }
}
```

Users/RiverBooks.Users/UserEndpoints/Create.CreateUserRequest.cs

```
public record CreateUserRequest(string Email, string Password);
```

Users/RiverBooks.Users/UserEndpoints/Create.cs

```
internal sealed class Create : Endpoint<CreateUserRequest>
{
  private readonly UserManager<ApplicationUser> _userManager;
  private readonly IMediator _mediator;

  public Create(UserManager<ApplicationUser> userManager,
    IMediator mediator)
  {
    _userManager = userManager;
    _mediator = mediator;
  }

  public override void Configure()
  {
    Post("/users");
    AllowAnonymous();
  }

  public override async Task HandleAsync(CreateUserRequest req,
    CancellationToken ct)
  {
    var command = new CreateUserCommand(req.Email, req.Password);

    var result = await _mediator.Send(command);

    if (!result.IsSuccess)
    {
      await SendResultAsync(result.ToMinimalApiResult());
      return;
    }
    await SendOkAsync();
  }
}
```

Users/RiverBooks.Users/UserEndpoints/Login.UserLoginRequest.cs

```
public record UserLoginRequest(string Email, string Password);
```

Users/RiverBooks.Users/UserEndpoints/Login.cs

```
public class Login : Endpoint<UserLoginRequest>
{
  private readonly UserManager<ApplicationUser> _userManager;

  public Login(UserManager<ApplicationUser> userManager)
  {
    _userManager = userManager;
  }
  public override void Configure()
  {
    Post("/users/login");
    AllowAnonymous();
  }

  public override async Task HandleAsync(UserLoginRequest request, CancellationToken
ct)
  {
    var user = await _userManager.FindByEmailAsync(request.Email!);
    if (user is null)
    {
      await SendUnauthorizedAsync();
      return;
    }
    var loginSuccessful = await _userManager.CheckPasswordAsync(user,
request.Password);

    if (!loginSuccessful)
    {
      await SendUnauthorizedAsync();
      return;
    }

    var jwtSecret = Config["Auth:JwtSecret"]!;
    var token = JWTBearer.CreateToken(jwtSecret, p => p["EmailAddress"] =
user.Email!);
    await SendAsync(token);
  }
}
```

RiverBooks.Web/appsettings.json

```json
{
  "ConnectionStrings": {
    "BooksConnectionString": "Server=(localdb)\\mssqllocaldb;Integrated
Security=true;Initial Catalog=RiverBooks;",
    "UsersConnectionString": "Server=(localdb)\\mssqllocaldb;Integrated
Security=true;Initial Catalog=RiverBooks;"
  },
  "AllowedHosts": "*",
  "Auth": {
    "JwtSecret": "really really REALLY long secret key goes here"
  }
}
```

Users/RiverBooks.Users/UseCases/AddItemToCartCommand.cs

```csharp
public record AddItemToCartCommand(Guid BookId, int Quantity, string EmailAddress) :
IRequest<Result>;
```

Users/RiverBooks.Users/UseCases/AddItemToCartHandler.cs

```csharp
internal class AddItemToCartHandler(IApplicationUserRepository userRepository,
    IMediator mediator)
 : IRequestHandler<AddItemToCartCommand, Result>
{
  private readonly IApplicationUserRepository _userRepository = userRepository;
  private readonly IMediator _mediator = mediator;

  public async Task<Result> Handle(AddItemToCartCommand request,
CancellationToken cancellationToken)
  {
    var user = await
_userRepository.GetUserWithCartByEmailAsync(request.EmailAddress);

    if (user is null) return Result.Unauthorized(); }

    var bookDetailsQuery = new BookDetailsQuery(request.BookId);
    var result = await _mediator.Send(bookDetailsQuery);

    var bookDetails = result.Value;

    var newCartItem = new CartItem(request.BookId, request.Quantity,
bookDetails.Price, $"{bookDetails.Title} by {bookDetails.Author}");

    user!.AddItemToCart(newCartItem);
    await _userRepository.SaveChangesAsync();

    return Result.Success();
  }
}
```