DDD with ASP.NET Core – Labs

Steve Smith (@ardalis) **October 2017**

Lab 0: Set up

Goal

Have a working development environment and starter application.

Requirements

- **Visual Studio 2017 with .NET Core Workload** or **Visual Studio Code plus CLI tools**.
- **.NET Core 2.0 SDK**
- A git client (command line is fine if you know it)

It will also be helpful to install:

- **Postman** (or **Fiddler** or another tool that allows you to GET/POST/PUT/DELETE to HTTP endpoints)
- **Smtp4Dev** or **Papercut** or another local email testing tool

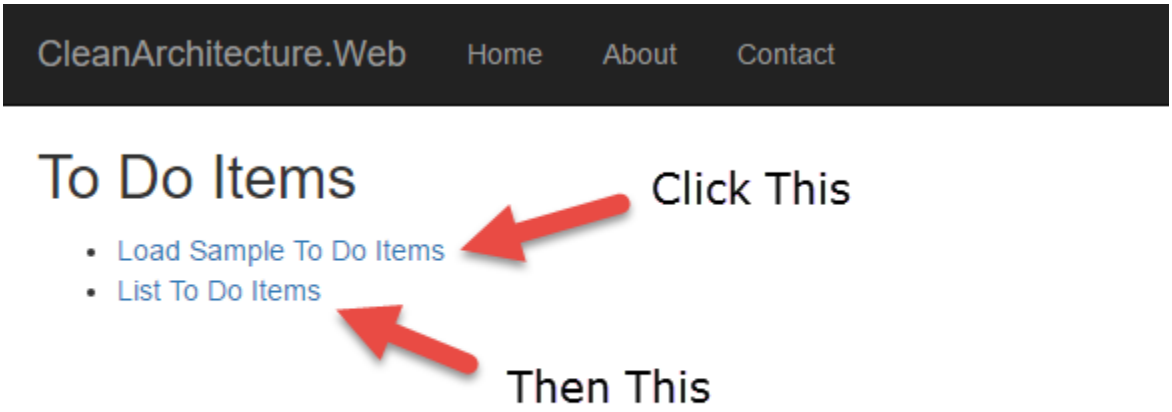
Instructions

We're going to be building a Guestbook application today using ASP.NET Core, Domain-Driven Design principles and patterns, and a domain-centric architecture. It's important that the solution be structured properly to support this approach. We will be starting from a Clean Architecture template which you'll find here:

https://github.com/ardalis/CleanArchitecture

Clone this sample to a working folder on your machine. If you download a ZIP of the repository, be sure to **Unblock** the zip file before you extract its contents (right click, Properties, Unblock).

You should be able to run the application and view the web app in your browser (F5 in Visual Studio; **dotnet restore / dotnet run** from terminal in web project folder). You should be able to populate and view several ToDo items (see image below):



You should be able to run the tests in the tests folder, either from the command line or a test runner within your IDE. From command line, go to test project folder, **dotnet restore / dotnet test**. They should all pass.

For now the labs will only be using InMemory data, so no database configuration is required at this time. The application's data will reset each time it is started.

Rename the solution to **DDDGuestbook.sln**.

Lab 1: The Guest Book application

Goal

Create a domain model consisting of a Guestbook and GuestbookEntry Entities. Display (dummy) entries on the web application's home page.

Topics Used

Entities, ViewModels

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 should display the most recent ten (10) entries recorded on the guestbook, ordered by when the entry was recorded (most recent first). For now, have the home page just display several hard-coded entries.

Details

Entities

1. Guestbook and GuestbookEntry should inherit from BaseEntity, which defines an integer Id property.
2. Entities should be defined in the Core project, in the Entities folder.
3. Guestbook should include a string Name property.
4. Guestbook should include a List<GuestbookEntry> Entries property.
5. GuestbookEntry should include
    a. string EmailAddress
    b. string Message
    c. DateTimeOffset DateTimeCreated (defaults to Now)

Example:

```
0 references | 0 changes | 0 authors, 0 changes
public class Guestbook : BaseEntity
{
    0 references | 0 changes | 0 authors, 0 changes
    public string Name { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public List<GuestbookEntry> Entries { get; } = new List<GuestbookEntry>();
}

5 references | 0 changes | 0 authors, 0 changes
public class GuestbookEntry : BaseEntity
{
    0 references | 0 changes | 0 authors, 0 changes
    public string EmailAddress { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Message { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public DateTimeOffset DateTimeCreated { get; set; } = DateTime.UtcNow;
}
```

Don't forget to initialize your collection properties to new Lists to avoid null reference exceptions later.

UI and ViewModel

6. The Home Page will display the guestbook name and up to 10 entries. Create a HomePageViewModel class in /ViewModels (create this folder) in the Web project. Include these properties:
    a. string GuestbookName
    b. List<GuestbookEntry> PreviousEntries
7. HomeController Index() should create a new Guestbook (Name="My Guestbook")
    a. Add several entries to its Entries collection (just hard-code them)
    b. Create a HomePageViewModel from a Guestbook and its entries
    c. Return a View, passing the viewmodel to the View
8. Modify the View/Index.cshtml view file in the Web project
    a. Specify @model CleanArchitecture.Web.ViewModels.HomePageViewModel at the top of the view file
    b. Display the Guestbook Name at the top of the page
    c. Loop through the Entries and display each entry, most recent first.

Example:

```
4 references | 0 changes | 0 authors, 0 changes
public class HomePageViewModel
{
    2 references | 0 changes | 0 authors, 0 changes
    public string GuestbookName { get; set; }
    3 references | 0 changes | 0 authors, 0 changes
    public List<GuestbookEntry> PreviousEntries { get; } = new List<GuestbookEntry>();
}
```

```
0 references | Steve Smith, 3 hours ago | 1 author, 1 change
public class HomeController : Controller
{
    0 references | Steve Smith, 3 hours ago | 1 author, 1 change
    public IActionResult Index()
    {
        var guestbook = new Guestbook() {Name = "My Guestbook"};
        guestbook.Entries.Add(new GuestbookEntry() { EmailAddress = "steve@devig.com", Message = "Hi!", DateTimeCreated = DateTime.UtcNow.AddHours(-2) });
        guestbook.Entries.Add(new GuestbookEntry() { EmailAddress = "mark@devig.com", Message = "Hi again!", DateTimeCreated = DateTime.UtcNow.AddHours(-1)});
        guestbook.Entries.Add(new GuestbookEntry() { EmailAddress = "michelle@devig.com", Message = "Hello!" });

        var viewModel = new HomePageViewModel();
        viewModel.GuestbookName = guestbook.Name;
        viewModel.PreviousEntries.AddRange(guestbook.Entries);

        return View(viewModel);
    }
```

```
.cs    ToDoItem.cs    HomePageViewModel.cs    Index.cshtml
@{
    ViewData["Title"] = "DDD Guestbook";
}
@model CleanArchitecture.Web.ViewModels.HomePageViewModel

<h2>Guestbook: @Model.GuestbookName</h2>
<h3>Messages:</h3>
<ol class="round">
    @if (!Model.PreviousEntries.Any())
    {
        <li class="zero">
            <h5>No Messages</h5>
            Nobody has left any messages.  How sad. :(
        </li>
    }
    @foreach (var entry in Model.PreviousEntries.OrderByDescending(p => p.DateTimeCreated))
    {
        <li class="arrow">
            <h5>@entry.EmailAddress - @entry.DateTimeCreated</h5>
            @entry.Message
        </li>
    }
</ol>
```

Run the app. You should see 3 messages on the home page, most recent at the top.

Lab 2: Adding Entries to the Guestbook

Goal

Add a form to the page that allows users to add new entries to the Guestbook.

Topics Used

Repository

Requirements

Display up to 10 of the most recent entries *from persistence* on the home page. For now, there is only ever one instance of a Guestbook. If none exists in persistence, create one when the home page loads.

Add a form that accepts an **EmailAddress** and a **Message** and POSTs to the IndexController. Persist new entries to the Guestbook.

Details

The CleanArchitecture template includes an IRepository<T> and an EFRepository<T> implementation already. You can see an example of it in action in the Web/Controllers/TodoController.cs class. Follow this example to pass an IRepository<Guestbook> into HomeController's constructor and assign it to a private local field.

Determine if there is already a Guestbook in persistence. If not, create and save one in the HomeController's Index() method, using the repository (you can use the code you have from lab 1 that creates a guestbook with several entries). In any case, fetch the first Guestbook from persistence and use it to populate the HomePageViewModel.

Example:

```
using System;
using System.Linq;
using CleanArchitecture.Core.Entities;
using CleanArchitecture.Core.Interfaces;
using CleanArchitecture.Web.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore.Migrations;

namespace CleanArchitecture.Web.Controllers
{
    1 reference | Steve Smith, 30 minutes ago | 1 author, 2 changes
    public class HomeController : Controller
    {
        private readonly IRepository<Guestbook> _guestbookRepository;

        0 references | 0 changes | 0 authors, 0 changes
        public HomeController(IRepository<Guestbook> guestbookRepository)
        {
            _guestbookRepository = guestbookRepository;
        }

        0 references | Steve Smith, 30 minutes ago | 1 author, 2 changes
        public IActionResult Index()
        {
            if (!_guestbookRepository.List().Any())
            {
                var newGuestbook = new Guestbook() {Name = "My Guestbook"};
                newGuestbook.Entries.Add(new GuestbookEntry()
                {
                    EmailAddress = "steve@deviq.com",
                    Message = "Hi!" });
                _guestbookRepository.Add(newGuestbook);
            }

            //var guestbook = _guestbookRepository.List().FirstOrDefault();
            var guestbook = _guestbookRepository.GetById(1);
            var viewModel = new HomePageViewModel();
            viewModel.GuestbookName = guestbook.Name;
            viewModel.PreviousEntries.AddRange(guestbook.Entries);

            return View(viewModel);
        }
```

Implement a Repository for Guestbook

The CleanArchitecture template comes with a generic EF repository, but it requires some updates to support Guestbook. Specifically, the dbcontext needs to have a DbSet of Guestbooks and when a Guestbook is fetched, it needs to include the associated Entries collection with it.

1. Add a DbSet<Guestbook> property 'Guestbooks' to the Infrastructure/Data/AppDbContext.cs class.
2. Modify the EfRepository to make its _dbContext field **`protected`** and its GetById method **`virtual`**.
3. Create a new GuestbookRepository class that inherits from EfRepository<Guestbook>
    a. Provide a constructor that passes a dbContext to the base constructor
    b. Override GetById as shown:
    c. Include `using Microsoft.EntityFrameworkCore;`

```
public class GuestbookRepository : EfRepository<Guestbook>
{
    public GuestbookRepository(AppDbContext dbContext) : base(dbContext)
    {
    }

    // Since Guestbook is an Aggregate Root we need it to include its children
    public override Guestbook GetById(int id)
    {
        return _dbContext.Guestbooks
            .Include(g => g.Entries)
            .FirstOrDefault(g => g.Id == id);
    }

}
```

4. In Startup (around line 47), configure IRepository<Guestbook> to use GuestbookRepository:

```
config.For<IRepository<Guestbook>>().Use<GuestbookRepository>();
```

At this point you should be able to view the home page and see your test data displayed, which is being stored in persistence (on the first request) and then fetched from persistence (on each request).

Implement Adding Entries

To support adding new entries to the guestbook, you need an HTML form in the view and a new controller action method to handle POSTs.

1. Update /Views/Home/Index.cshtml to **include a form above the message list (between the <h2> and <h3> tags)**. ASP.NET Core tag helpers provide a replacement for HTML Helpers used in previous versions of ASP.NET MVC.

Example:

(Note: You can copy the Razor/HTML code below from here: http://bit.ly/2qB7sX5)

```
<form asp-controller="Home" asp-action="Index" method="post" class="form-horizontal">
    <h4>Sign the Guestbook</h4>
    <hr />
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="NewEntry.EmailAddress" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="NewEntry.EmailAddress" class="form-control" />
            <span asp-validation-for="NewEntry.EmailAddress" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="NewEntry.Message" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="NewEntry.Message" class="form-control" />
            <span asp-validation-for="NewEntry.Message" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Save</button>
        </div>
    </div>
</form>
```

Note that the form **posts** to the **Home** controller's **Index** method, and the two values are EmailAddress and Message from a NewEntry property on the viewmodel.

2. Add a **NewEntry** property of type GuestbookEntry to the HomePageViewModel.
3. Add a new action method to /Controllers/HomeController.cs: **public IActionResult Index(HomePageViewModel model)**
    a. Add the **[HttpPost]** attribute to the method
    b. Check if ModelState.IsValid. If it is:
        i. Fetch the appropriate guestbook (in this case the first one, or ID 1)
        ii. Add the NewEntry from the model to the guestbook's Entries collection.
        iii. Update the guestbook using the _guestbookRepository.
        iv. Update the model's PreviousEntries collection to match guestbook.Entries
    c. Return View(model)

Example:

```
[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult Index(HomePageViewModel model)
{
    if (ModelState.IsValid)
    {
        var guestbook = _guestbookRepository.GetById(1);
        guestbook.Entries.Add(model.NewEntry);
        _guestbookRepository.Update(guestbook);

        model.PreviousEntries.Clear();
        model.PreviousEntries.AddRange(guestbook.Entries);
    }
    return View(model);

}
```

Run the application. You should be able to add additional messages to the guestbook, and they should appear at the top of the list.

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 tag. 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 tag. Save (and commit, if you're using git for your own work) you work. Then go to a new folder to clone the sample labs using these commands:

```
git clone https://github.com/ardalis/ddd-guestbook
```

```
git checkout tags/2.0Lab3Start
```

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

You can view a list of available tags using

```
git tag
```

Assuming you're using ASP.NET 2.0, you'll want to use the tags that are prefixed with **`2.0`**.

Lab 3: Notifying Users of New Entries

Goal

When a new entry is added to the guestbook, notify anyone who has previously signed the guestbook, *but only within the last day*.

Topics Used

Domain Events, Services

Requirements

Step 1:

Implement logic in the controller's POST action to loop through previous entries and send an email to any that have been made within the last day. ASP.NET Core 2.0 supports System.Net.Mail for sending messages via SMTP. If you're using an earlier version, you can install the "MailKit" Nuget package instead.

Add the following (non-grey) code before you add the new entry to **`guestbook.Entries`** `in your HomeController Post method`. You shouldn't send an email to the person who just signed the guestbook (which is why we're looping through entities *before* adding the current message).

```
var guestbook = _guestbookRepository.GetById(1);

// notify all previous entries
foreach (var entry in guestbook.Entries)
{
    var message = new MailMessage();
    message.To.Add(new MailAddress(entry.EmailAddress));
    message.From = new MailAddress("donotreply@guestbook.com");
    message.Subject = "New guestbook entry added";
    message.Body =  model.NewEntry.Message ;
    using (var client = new SmtpClient("localhost",25))
    {
        client.Send(message);
    }
}

guestbook.Entries.Add(model.NewEntry);
```

Manually test that your solution works, using Smtp4Dev/Papercut or similar.

Consider:

- What impact does this approach have on testability of the controller?
- Where in the solution does the business logic for this requirement live?
- How valuable is the current domain model in representing this logic?

Refactor to use a domain service

Putting this much code in a controller action is a code smell – there's way too much going on. Let's move what we can into a service for now as a good incremental step to improve the design.

Step 2: Encapsulate Email Sending in an Infrastructure service.

- Create an interface to represent sending email (e.g. IMessageSender)
    - Place in Core/Interfaces
    - Keep as simple as possible(e.g. `void SendGuestbookNotificationEmail(string toAddress, string messageBody);`)
- Implement the interface in Infrastructure, in a Services folder (e.g. EmailMessageSenderService : IMessageSender)

```
using CleanArchitecture.Core.Interfaces;
using System.Net.Mail;

namespace CleanArchitecture.Infrastructure.Services
{
    0 references | 0 changes | 0 authors, 0 changes
    public class EmailMessageSenderService : IMessageSender
    {
        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public void SendGuestbookNotificationEmail(string toAddress, string messageBody)
        {
            var message = new MailMessage();
            message.To.Add(new MailAddress(toAddress));
            message.From = new MailAddress("donotreply@guestbook.com");
            message.Subject = "New guestbook entry added";
            message.Body = messageBody;
            using (var client = new SmtpClient("localhost", 25))
            {
                client.Send(message);
            }
        }
    }
}
```

- In Startup.cs, configure IMessageSender to use EmailMessageSender.
  `config.For<IMessageSender>().Use<EmailMessageSenderService>();`
- Inject IMessageSender into HomeController and re-test your email sending logic

```
// notify all previous entries
foreach (var entry in guestbook.Entries)
{
    _messageSender.SendGuestbookNotificationEmail(entry.EmailAddress, model.NewEntry.Message);
}
```

Step 3: Encapsulate Sending Notifications

We want to pull the logic of deciding who should get a notification out of the POST action method and into a service in Core. **Note**, this approach is better than leaving the logic in the UI project, but not ideal because it splits business functionality related to Guestbook into a service, tending to make Guestbook *anemic*. We'll shortly see how we can make Guestbook itself more responsible for this behavior.

- Create an interface IGuestbookService
  - Place in Core/Interfaces
  - Add method **`void RecordEntry(Guestbook guestbook, GuestbookEntry newEntry);`**
  - (later you could add additional methods, if desired)
- Create a new class, GuestbookService, in Core/Services
  - Implement IGuestbookService
  - Inject a repository to access guestbook (IRepository<Guestbook>)
  - Inject an IMessageSender to send emails
  - Implement RecordEntry, moving the logic for saving the entry and looping through previous entries from HomeController to this method.
  - Inject an IGuestbookService into HomeController and call its RecordEntry() method inside the POST action.
    - You should be able to remove IMessageSender from HomeController after this is done.
  - Note: If you named your interface and implementation correctly, StructureMap will map them to one another automatically by convention (so you won't need to add a line to Startup ConfigureServices).

Test your application again. It should still work as it did before. The controller method should be much smaller now than it was when it was responsible for identifying recipients, building messages, and sending emails.

```
[HttpPost]
0 references | Steve Smith, 7 minutes ago | 1 author, 9 changes | 0 requests | 0 exceptions
public IActionResult Index(HomePageViewModel model)
{
    if (ModelState.IsValid)
    {
        var guestbook = _guestbookRepository.GetById(1);

        _guestbookService.RecordEntry(guestbook, model.NewEntry);

        model.PreviousEntries.Clear();
        model.PreviousEntries.AddRange(guestbook.Entries);
    }
    return View(model);
}
```

Step 4: Use a Domain Event

Eliminate the need for a service that only works with one Entity by putting the logic into the Entity itself and raising a *domain event* to trigger additional behavior. The CleanArchitecture sample project already has built-in support for domain events that are fired when an entity is saved successfully. We just need to add the event and a handler.

- Create a new domain event in Core/Events called EntryAddedEvent.
- Inherit from BaseDomainEvent.
- Include the entry and the guestbook ID in its constructor; expose these as get-only properties.
- Modify Core/Entities/Guestbook.cs:
    - Add a new public void AddEntry(GuestbookEntry entry) method that adds the entry to Entries.
    - Create a new EntryAddedEvent and added it to the Events collection.
    - *(later) Change Entries to make it harder for clients to manipulate it directly – we only want new entries to be added using the AddEntry method.*
- Create a GuestbookNotificationHandler in Core/Handlers *(add a new Handlers folder in the Core project)*
    - Implement IHandle<EntryAddedEvent>
    - Inject IRepository<Guestbook> and IMessageSender (*just like in GuestbookService*)
    - Copy notification logic from GuestbookService to Handle method
    - Realize we missed a requirement! **Only send messages to other guestbook signers who signed within the last day.**
    - Update HomeController POST method to call Guestbook.AddEntry()

Test the application again – its behavior should remain unchanged*.

Samples

```
public class EntryAddedEvent : BaseDomainEvent
{
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public EntryAddedEvent(int guestbookId, GuestbookEntry entry)
    {
        GuestbookId = guestbookId;
        Entry = entry;
    }
    1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
    public int GuestbookId { get; }
    1 reference | 0 changes | 0 authors, 0 changes | 0 exceptions
    public GuestbookEntry Entry { get; }
}
```

In Guestbook:

```
public void AddEntry(GuestbookEntry entry)
{
    Entries.Add(entry);
    Events.Add(new EntryAddedEvent(this.Id, entry));
}
```

GuestbookNotificationHandler:

```
using CleanArchitecture.Core.Entities;
using CleanArchitecture.Core.Events;
using CleanArchitecture.Core.Interfaces;
using System;
using System.Linq;

namespace CleanArchitecture.Core.Handlers
{
    1 reference | 0 changes | 0 authors, 0 changes
    public class GuestbookNotificationHandler : IHandle<EntryAddedEvent>
    {
        private readonly IRepository<Guestbook> _guestbookRepository;
        private readonly IMessageSender _messageSender;

        0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public GuestbookNotificationHandler(IRepository<Guestbook> guestbookRepository,
            IMessageSender messageSender)
        {
            _guestbookRepository = guestbookRepository;
            _messageSender = messageSender;
        }

        3 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public void Handle(EntryAddedEvent entryAddedEvent)
        {
            var guestbook = _guestbookRepository.GetById(entryAddedEvent.GuestbookId);

            // send updates to previous entries made in the last day
            var emailsToNotify = guestbook.Entries
                .Where(e => e.DateTimeCreated > DateTimeOffset.UtcNow.AddDays(-1))
                .Select(e => e.EmailAddress);

            foreach(var emailAddress in emailsToNotify)
            {
                string messageBody = $"{entryAddedEvent.Entry.EmailAddress} left a new message {entryAddedEvent.Entry.Message}.";
                _messageSender.SendGuestbookNotificationEmail(emailAddress, messageBody);
            }
        }
    }
}
```

*There is a small bug in the code above. If you leave it as-is you'll fix it in a later lab. See if you can identify what it is.

Note that GuestbookService can now be deleted. Its existence was tragically cut short by the introduction of the GuestbookNotificationHandler that does the same thing, but automatically whenever an event is raised.

Reference:

https://github.com/ardalis/ddd-guestbook see Tags for different Lab starting points

Lab 4: Testing

Goal

Demonstrate the testability of the solution we've built, and how to test full stack ASP.NET Core MVC apps.

Topics Used

Testing, xUnit

Requirements

The Guestbook needs to support mobile and/or rich client apps, and thus requires an API. The API needs to support two methods initially:

- **ListEntries**: Should list the same entries as the current home page
- **AddEntry**: Should add a new entry (just like the form on the current home page)

Detailed Steps

1. Add a new API Controller called GuestbookController to the web project, in the Api folder (/Api/GuestController.cs).
    a. Use the ToDoItemsController as a reference.
2. Add a method, GetById, that will return a Guestbook or a 404 if the specified id doesn't exist

Example

```
// GET: api/Guestbook/1
[HttpGet("{id:int}")]
public IActionResult GetById(int id)
{
    var guestbook = _guestbookRepository.GetById(id);
    if (guestbook == null)
    {
        return NotFound(id);
    }
    return Ok(guestbook);
}
```

3. Add a new integration test class for this GetById in Tests/Integration/Web.
    a. Use the ApiToDoItemsControllerListShould class as a reference/starting point.
    b. Inherit from BaseWebTest. Modify BaseWebTest.cs as follows:
        i. Change line 77's reference to CleanArchitecture.sln to your solution file name (DDDGuestbook.sln)
    c. Add test data (a sample guestbook with entries) to Web/SeedData.cs in the PopulateTestData method
        i. Be sure to use Entries.Add instead of AddEntry to avoid raising events.
        ii. Add one Guestbook with one GuestbookEntry.
        iii. Use a disposable TestServerFixture (sample at end of this lab)
4. Write a test that confirms the 404 behavior occurs when expected.
5. Write a test that confirms the guest book and its entry are returned correctly when a valid id is supplied.

You should be able to run all of your solution's tests (should be 9 of them) now, and they should all **pass**.

Example

```
public class ApiGuestbookControllerListShould : BaseWebTest
{
    [Fact]
    public void ReturnGuestbookWithOneItem()
    {
        var response = _client.GetAsync("/api/guestbook/1").Result;
        response.EnsureSuccessStatusCode();
        var stringResponse = response.Content.ReadAsStringAsync().Result;
        var result = JsonConvert.DeserializeObject<Guestbook>(stringResponse);
```

```
        Assert.Equal(1, result.Id);
        Assert.Equal(1, result.Entries.Count());
    }

    [Fact]
    public void Return404GivenInvalidId()
    {
        string invalidId = "100";
        var response = _client.GetAsync($"/api/guestbook/{invalidId}").Result;

        Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
        var stringResponse = response.Content.ReadAsStringAsync().Result;

        Assert.Equal(invalidId.ToString(), stringResponse);
    }
}
```

Add to SeedData:

```
// add Guestbook test data; specify Guestbook ID for use in tests
var guestbook = new Guestbook() { Name = "Test Guestbook", Id = 1 };
dbContext.Guestbooks.Add(guestbook);
guestbook.Entries.Add(new GuestbookEntry()
{
    EmailAddress = "test@test.com",
    Message = "Test message"
});
dbContext.SaveChanges();
```

6. Add a new API method to record an entry to a guestbook.
    a. Method should take in a guestbook ID and a GuestbookEntry.
    b. Method should return 404 if no Guestbook exists for the ID.
    c. Return the updated Guestbook if successful.
7. Add a new integration test class for this NewEntry API method (ApiGuestbookControllerNewEntryShould)
8. Write a test to confirm 404 behavior
9. Write a test to confirm entry is created and returned with guestbook correctly
10. Run all of your tests – you should have 11 passing tests.

Example

```
[HttpPost("{id:int}/NewEntry")]
public IActionResult NewEntry(int id, [FromBody] GuestbookEntry entry)
{
    var guestbook = _guestbookRepository.GetById(id);
    if (guestbook == null)
    {
        return NotFound(id);
    }
    guestbook.AddEntry(entry);
    _guestbookRepository.Update(guestbook);

    return Ok(guestbook);
}
```

Example Tests

```
using CleanArchitecture.Core.Entities;
using Newtonsoft.Json;
using System;
using System.Net;
using System.Net.Http;
using System.Text;
using Xunit;

namespace CleanArchitecture.Tests.Integration.Web
{
    public class ApiGuestbookControllerNewEntryShould : BaseWebTest
    {
        [Fact]
        public void Return404GivenInvalidId()
        {
            string invalidId = "100";
            var entryToPost = new { EmailAddress = "test@test.com", Message = "test" };
            var jsonContent = new StringContent(JsonConvert.SerializeObject(entryToPost), Encoding.UTF8,
                "application/json");
            var response = _client.PostAsync($"/api/guestbook/{invalidId}/NewEntry", jsonContent).Result;

            Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
            var stringResponse = response.Content.ReadAsStringAsync().Result;

            Assert.Equal(invalidId, stringResponse);
        }

        [Fact]
        public void ReturnGuestbookWithOneItem()
        {
            int validId = 1;
            string message = Guid.NewGuid().ToString();
            var entryToPost = new { EmailAddress = "test@test.com", Message = message };
            var jsonContent = new StringContent(JsonConvert.SerializeObject(entryToPost), Encoding.UTF8,
                "application/json");
            var response = _client.PostAsync($"/api/guestbook/{validId}/NewEntry", jsonContent).Result;
            response.EnsureSuccessStatusCode();
            var stringResponse = response.Content.ReadAsStringAsync().Result;
            var result = JsonConvert.DeserializeObject<Guestbook>(stringResponse);

            Assert.Equal(validId, result.Id);
            Assert.Contains(result.Entries, e => e.Message == message);
        }
    }
}
```

Notes

Your NewEntry tests will fail if you do not have a local mailserver (Smtp4Dev, Papercut) running. Make sure it's running when you run your tests. Alternately, configure your tests so they use an alternate (fake) implementation of IMessageSender.

Lab 5: Filters

Goal

Demonstrate how to reduce duplication by pulling out cross-cutting concerns and policies and implementing them as filters.

Topics Used

MVC Filters

Requirements / Overview

In the previous lab, the API controller you wrote returns 404 whenever the provided ID doesn't fetch a Guestbook instance. You can imagine that in a real application this kind of behavior will be extremely common and repetitive. It's important especially for APIs that you be consistent, and it would be easy to have different controllers or actions behave differently, sometimes returning NotFound and other times BadRequest or simply throwing NullReferenceException. Filters can encapsulate policies and can be applied without having to touch implementation code. They can be tested using integration tests as described in the previous lab.

Detailed instructions

1. Add a new VerifyGuestbookExistsAttribute.cs file to Web/Filters.
    a. See https://github.com/ardalis/GettingStartedWithFilters for more reference info on filters
2. Have your class inherit from `TypeFilterAttribute`
3. Create a constructor that chains to the base type: `base(typeof(VerifyGuestbookExistsFilter))`
4. Create a private nested class VerifyGuestbookExistsFilter
5. Have this class implement IAsyncActionFilter
6. Inject an `IRepository<Guestbook>` into this class.
7. Implement the OnActionExecutionAsync method. Don't forget to make it `async`:

```
public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
{
    if (context.ActionArguments.ContainsKey("id"))
    {
        var id = context.ActionArguments["id"] as int?;
        if (id.HasValue)
        {
            if ((_guestbookRepository.GetById(id.Value)) == null)
            {
                context.Result = new NotFoundObjectResult(id.Value);
                return;
            }
        }
    }
    await next();
}
```

8. Add an attribute to the API methods that should return a 404 when no guestbook is found.
    a. `[VerifyGuestbookExists]`
    b. Optionally you can apply this to the entire Api/GuestbookController class.
9. Remove the if statements that check guestbook for null and return NotFound.
10. Confirm all of your tests still run successfully (especially the ones expecting NotFound).

Lab 6: Specification

Goal

Extract query logic used when sending notifications into a *Specification*.

Topics Used

Specification pattern

Requirements / Overview

The Guestbook currently (unless you already fixed it) sends an email to the person who last left a message, in addition to others. This is a bug.

Pull the filtering logic into its own type where it can be reused and tested.

Detailed Steps

1. Create an `ISpecification<T>` interface in Core/Interfaces
    a. One property on this interface: **`Expression<Func<T, bool>> Criteria { get; }`**
    b. Extract querying/filtering logic from GuestbookNotificationHandler.cs
        i. Put the lambda expression in a new GuestbookNotificationPolicy class in a new Specifications folder in Core.
    c. GuestbookNotificationPolicy should implement ISpecification<GuestbookEntry>
    d. Implement the Criteria property with the filter lambda expression.
    e. Add a new List<GuestbookEntry> ListEntries(ISpecification<GuestbookEntry> spec) method to GuestbookRepository.
        i. You'll also need to add a new public DbSet<GuestbookEntry> property to AppDbContext
    f. Add the ListEntries method signature to a new IGuestbookRepository interface.
        i. Put this interface in Core/Interfaces.
        ii. Have this interface inherit from IRepository<Guestbook>
        iii. Have GuestbookRepository inherit IGuestbookRepository.
    g. Add unit tests that confirm your criteria works as expected:
        i. Create a list of test entries with different dates.
        ii. Test that entries from the last day are returned.
        iii. Entries older than 24 hours should not be returned.
        iv. The entry that triggered the notification should not be returned. You can pass in the id of the triggering entry to your GuestbookNotificationPolicy constructor to achieve this.
    h. Update GuestbookNotificationHandler to
        i. Use IGuestbookRepository
        ii. Create an instance of GuestbookNotificationPolicy in its Handle method
        iii. Get emailsToNotify by calling ListEntries(policy).Select(e => e.EmailAddress)
    i. Run and test that the application works as before and notifications are sent correctly.

Example: *GuestbookNotificationPolicy*

```
public class GuestbookNotificationPolicy : ISpecification<GuestbookEntry>
{
    private readonly int _entryId;

    public GuestbookNotificationPolicy(int entryId)
    {
        _entryId = entryId;
    }

    public Expression<Func<GuestbookEntry, bool>> Criteria {
        get
        {
            return e =>
                e.DateTimeCreated > DateTimeOffset.UtcNow.AddDays(-1)
                && e.Id != _entryId;
        }
    }
}
```

Example: *IGuestbookRepository*

```
public interface IGuestbookRepository : IRepository<Guestbook>
{
    List<GuestbookEntry> ListEntries(ISpecification<GuestbookEntry> spec);
}
```

Example: *GuestbookRepository*

```
public class GuestbookRepository : EfRepository<Guestbook>, IGuestbookRepository
{
    public GuestbookRepository(AppDbContext dbContext) : base(dbContext)
    {
    }

    // Since Guestbook is an Aggregate Root we need it to include its children
    public override Guestbook GetById(int id)
    {
        return _dbContext.Guestbooks
            .Include(g => g.Entries)
            .FirstOrDefault(g => g.Id == id);
    }

    public List<GuestbookEntry> ListEntries(ISpecification<GuestbookEntry> spec)
    {
        return _dbContext.GuestbookEntries
            .Where(spec.Criteria)
            .ToList();
    }
}
```