JAMES STILL

Software, Design, Architecture and Whatever Else

ARCHITECTURE, C#, WEB DEVELOPMENT

A Simple CQRS Pattern Using C# in .NET

Posted on April 26, 2017 by James Still in Architecture, C#, Web Development

For years in my apps I've used a Data Mapper pattern or even the Repository pattern to mediate between the business domain and the database. One thing I've learned is that something like this interface is no help at all:

```
1.  public interface IEntity
2.  {
3.      int ID { get; set; }
4.  }
5.
6.  public interface IRepository where T : IEntity
7.  {
8.      T Get(int id);
9.      IEnumerable GetAll();
10.     int Add(T item);
11.     bool Update(T item);
12.     bool Delete(T item);
13. }
```

In all but the most trivial app it proves too inflexible to be useful. This is because no two entities are alike. In one case calling a Get() function and passing in an ID is just fine. But I might have a two-part key for another entity. Or I might need GetByLastName() instead. So I end up adding extra functions to the concrete repository class. But If I'm adding functions outside the contract then I might as well not use the contract at all. Another problem with a non-trivial app is the repository even for a single entity quickly becomes a big ball of mud (if you're lucky) or a God class if several developers are working together and there's no discipline. If I have to wade through 30 fetch functions to get to the one I want that's not maintainable. There are other growing pains that emerge a few years down the road. But others — notably Ayende — have documented those problems so I won't rehash that here. Instead I want to describe a simple CQRS pattern as an alternative that I've found to be flexible and maintainable over the long haul.

A robust CQRS pattern would physically separate the query model from the command model, running each in its own process. Sometimes there's even two data stores: an OLTP store for the commands and a denormalized store for the queries with a message queue

supporting eventual consistency between them. This advanced scenario is very complex and not to be undertaken lightly. For current purposes I'm going to describe a logical, rather than a physical, separation between command and queries.

In my .NET solution I've added a top-level folder Domain with Commands and Queries nested in it. Under Command there are two folders Command and Handler. And under Queries there are two folders Query and Handler as in this example:

Even this can be a little too generic for many projects. I might be better off organizing the folder structure by domain entites because we could have a hundred queries in a large system. But let's keep it simple. Under the Domain folder I'm going to add a mock database to simulate a real one and load it with three widgets:

```
1.  public static class MockWidgetDatabase
2.  {
3.      public static IList Widgets { get; }
4.      public static int UniqueWidgetId = 4;
5.
6.      static MockWidgetDatabase()
7.      {
8.          Widgets = new List()
9.          {
10.             new Widget {ID = 1, Name = "Widget 1", Shape = "Shape 1"},
11.             new Widget {ID = 2, Name = "Widget 2", Shape = "Shape 2"},
12.             new Widget {ID = 3, Name = "Widget 3", Shape = "Shape 3"}
13.         };
14.     }
15. }
```

We need two domain entities in the system. The Widget entity and an optional CommandResponse entity:

```
1.  public class Widget
2.  {
3.      public int ID { get; set; }
4.      public string Name { get; set; }
5.      public string Shape { get; set; }
6.  }
7.
8.  public class CommandResponse
9.  {
```

```
10.         public int ID { get; set; }
11.         public bool Success { get; set; }
12.         public string Message { get; set; }
13.     }
```

Let's write a query to fetch all widgets from the database. I want a consistent way of implementing my queries so I'm going to create two interfaces in my Queries folder:

```
1.  public interface IQuery<out TResponse> { }
2.
3.  public interface IQueryHandler<in TQuery, out TResponse> where TQuery :
    IQuery<TResponse>
4.  {
5.      TResponse Get();
6.  }
```

With that done I can write my first query which lives in the \Domain\Queries\Query folder that I'll call AllWidgetsQuery:

```
1.  public class AllWidgetsQuery : IQuery<IEnumerable<Widget>> { }
```

The caller will use this query to get an IEnumerable of all Widget entities in the system. Therefore there are no properties. This is about as basic as you can get. Now we need this handler which I'll name AllWidgetsQueryHandler:

```
1.  public class AllWidgetsQueryHandler : IQueryHandler<AllWidgetsQuery,
    IEnumerable<Widget>>
2.  {
3.      public IEnumerable<Widget> Get()
4.      {
5.          return MockWidgetDatabase.Widgets.OrderBy(w => w.Name);
6.      }
7.  }
```

What if I don't want my caller (probably a Controller method) to know about my handler instance? Let's create a query factory in Domain\Queries to abstract that away from our controller method:

```
1.  public static class WidgetQueryHandlerFactory
2.  {
3.      public static IQueryHandler<AllWidgetsQuery, IEnumerable<Widget>>
    Build(AllWidgetsQuery query)
4.      {
5.          return new AllWidgetsQueryHandler();
6.      }
7.  }
```

Now the controller only needs to create the query it wants to use and pass it into the factory to get back an IQueryHandler interface implementation. The idea is to keep the controller methods uncluttered as in this example:

```
1.  public class WidgetsController : ApiController
```

```
{
    [HttpGet]
    [Route("v1/widgets")]
    public IEnumerable GetAll()
    {
        var query = new AllWidgetsQuery();
        var handler = WidgetQueryHandlerFactory.Build(query);
        return handler.Get();
    }
}
```

Because the factory knows which handler to return for the given query it makes it easier for the controller. Now let's add a little complexity and write a query called OneWidgetQuery that needs a parameter to know which Widget entity to fetch:

```
public class OneWidgetQuery : IQuery<Widget>
{
    public int ID { get; private set; }

    public OneWidgetQuery(int id)
    {
        ID = id;
    }
}
```

The handler will add a constructor so that the query can be passed into the object:

```
public class OneWidgetQueryHandler : IQueryHandler<OneWidgetQuery, Widget>
{
    private readonly OneWidgetQuery _query;

    public OneWidgetQueryHandler(OneWidgetQuery query)
    {
        _query = query;
    }

    public Widget Get()
    {
        return MockWidgetDatabase.Widgets.FirstOrDefault(w => w.ID ==
_query.ID);
    }
}
```

And we need to update our query factory to return the handler:

```
public static class WidgetQueryHandlerFactory
{
    public static IQueryHandler<OneWidgetQuery, Widget> Build(OneWidgetQuery
query)
    {
        return new OneWidgetQueryHandler(query);
    }

    public static IQueryHandler<AllWidgetsQuery, IEnumerable<Widget>>
Build(AllWidgetsQuery query)
    {
        return new AllWidgetsQueryHandler();
    }
}
```

As far as our controller is concerned the code to fetch a single Widget follows the sample pattern as the code to fetch all of the Widgets:

```
[HttpGet]
[Route("v1/widgets/{id}")]
public Widget GetWidget(int id)
{
    var query = new OneWidgetQuery(id);
    var handler = WidgetQueryHandlerFactory.Build(query);
    return handler.Get();
}
```

This pattern can be very powerful. The controller doesn't need to know how the sausage is made. And testing is easier. It is true that instead of wading through dozens of methods in a repository or data mapper class you're wading through dozens of handler classes. But in my experience this is a good trade off. I can find the particular handler I'm looking for a lot easier in a folder structure than I can in a class. I don't like hitting Ctrl-F or the dropdown to search for the method I want. I like looking down through a list of classes instead. And by separating my data logic this way I have more confidence that any changes I make are done in relative isolation. That's because I'm updating a file that has a single responsibility instead of a repository class with dozens of methods inside of it. Developers who have to maintain their own code know that it's easier to work with small lean classes. Refactoring always involves breaking large classes down into smaller ones. So why not design the data layer with that in mind?

Let's add a new command to the system. As with queries we're going to need an ICommand and an ICommandHandler interface:

```
public interface ICommand<out TResult> { }

public interface ICommandHandler<in TCommand, out TResult> where TCommand :
ICommand<TResult>
{
    TResult Execute();
}
```

Now I can implement a new command in Domain\Commands\Command called SaveWidgetCommand that takes a Widget entity in its constructor:

```
public class SaveWidgetCommand : ICommand<CommandResponse>
{
    public Widget Widget { get; private set; }

    public SaveWidgetCommand(Widget item)
    {
        Widget = item;
    }
}
```

I'm choosing to return a CommandResponse object to the caller. I could just as easily have made it a primitive type like boolean. But this way I can add some rich data that the caller might need like the new database key of the object once created or any error messages if something went wrong. My command handler in Domain\Commands\Handler is responsible for upserting the Widget entity in our mock database:

```
 public class SaveWidgetCommandHandler : ICommandHandler<SaveWidgetCommand, CommandResponse>
 {
     private readonly SaveWidgetCommand _command;

     public SaveWidgetCommandHandler(SaveWidgetCommand command)
     {
         _command = command;
     }

     public CommandResponse Execute()
     {
         var response = new CommandResponse()
         {
             Success = false
         };

         try
         {
             var item = MockWidgetDatabase
                 .Widgets
                 .FirstOrDefault(w => w.ID == _command.Widget.ID);

             if (item == null)
             {
                 item.ID = MockWidgetDatabase.UniqueWidgetId;
                 MockWidgetDatabase.UniqueWidgetId++;
                 MockWidgetDatabase.Widgets.Add(item);
             }
             else
             {
                 item.Name = item.Name;
                 item.Shape = item.Shape;
             }

             response.ID = item.ID;
             response.Success = true;
             response.Message = "Saved widget.";
         }
         catch
         {
             // log error
         }

         return response;
     }
 }
```

And as with our queries we need a static factory class to mediate between the controller and the handler itself:

```
 public static class WidgetCommandHandlerFactory
 {
```

```
    public static ICommandHandler<SaveWidgetCommand, CommandResponse>
Build(SaveWidgetCommand command)
    {
        return new SaveWidgetCommandHandler(command);
    }
}
```

The controller method that implements the POST verb follows the same pattern as with the queries. However, it also has a little extra plumbing to return an appropriate IHttpActionResult based on the response from the handler:

```
[HttpPost]
[Route("v1/widgets")]
public IHttpActionResult Post(Widget item)
{
    var command = new SaveWidgetCommand(item);
    var handler = WidgetCommandHandlerFactory.Build(command);
    var response = handler.Execute();
    if (response.Success)
    {
        item.ID = response.ID;
        return Ok(item);
    }

    // an example of what might have gone wrong
    var message = new HttpResponseMessage(HttpStatusCode.InternalServerError)
    {
        Content = new StringContent(response.Message),
        ReasonPhrase = "InternalServerError"
    };

    throw new HttpResponseException(message);
}
```

I'd probably want to add a delete handler. And maybe I don't want a save handler that combines insert and update and I'd rather split those out into two handlers. I had two queries but it wouldn't surprise me if I ended up a half-dozen queries like WidgetsByMachineQuery, WidgetsNeedingMaintenanceQuery, WidgetsByShapeQuery, WidgetsByColorQuery, and so on. As I mentioned the queries and commands are logically separated. As the system grows a few years down the road I might want to pull out the query logic and put it in its own API. It would be far easier to do that with this design than in a big ball of mud repository class.

I hope this gives you some ideas for your next project!

Like this:

Like  

2 bloggers like this.

ASP.NET C# CQRS DDD

PREVIOUS POST
Oregon Star Party 2015

NEXT POST
Microservices with IdentityServer4 and Ocelot Fronting a .NET Core API