Entity Framework Core IN ACTION

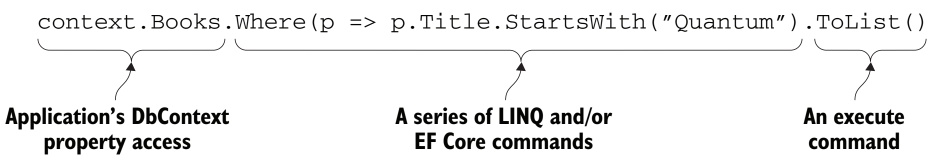
Jon P Smith

Sample Chapter

MANNING

Key topics covered in this book: the primary chapter covering each topic is listed first. Key figures that go with the topic are also listed.

Topics	Chapters	Key figures
Setting up EF Core	1, 2, 6, 7, 8, 5	1.4, 1.5, 2.6
Query the database	2, 5, 10	2.5, 2.9,
Create, update, delete	3, 5, 7, 10	3.1, 3.2, 3.3, 3.4
Business logic	4, 5, 10	4.2, 5.1, 5.4
ASP.NET Core	5, 2	5.1, 5.4
Dependency injection	5, 14, 15	5.2, 5.3
Async/await	5, 12	5.8, 5.9, 5.10
Configure non-relational	6	6.1, 6.2
Configure relationships	7, 8	7.1, 7.2, 7.3, 7.4
Configure table mappings	7	7.10, 7.11
Concurrency issues	8, 13	8.3, 8.4, 8.5, 8.6, 8.7
How EF Core works inside	1, 9, 14	1.6, 1.8, 1.10, 9.1
Design patterns	10, 4, 12	5.1, 10.1, 10.5, 10.6, 14.1, 14.2
Domain-driven design	10, 4	4.2, 10.5, 10.6
Database migrations	11, 5	11.1, 11.2, 11.3, 11.4, 11.5, 11.6
Performance tuning	12, 13, 14	12.1, 11.2, 11.4, 13.7, 14.5
Different databases	14	
Data validation	6, 4, 10	10.7
Unit testing	15	15.2
LINQ language	Appendix A, 2	A.2, A.1



```
context.Books.Where(p => p.Title.StartsWith("Quantum").ToList()
```

Application's DbContext property access
A series of LINQ and/or EF Core commands
An execute command

An example of an Entity Framework Core database query

Entity Framework Core in Action

by Jon P Smith

Chapter 4

brief contents

4 Using EF Core in business logic

This chapter covers

- Understanding business logic and its use of EF Core
- Using a pattern for building business logic
- Working through a business logic example
- Adding validation of data before it's written to the database
- Using transactions to daisy-chain code sequences

Real-world applications are built to supply a set of services, ranging from holding a simple list of things on your computer to managing a nuclear reactor. Every real-world problem has a set of rules, often referred to as *business rules*, or by the more generic name *domain rules* (this book uses *business rules*).

The code you write to implement a business rule is known as *business logic* or *domain logic*. Because business rules can be complex, the business logic you write can also be complex. Just think about all the checks and steps that should be done when you order something online.

Business logic can range from a simple check of status to massive artificial intelligence (AI) code, but in nearly all cases, business logic needs access to a database. Although the approaches in chapters 2 and 3 all come into play, the way you apply those EF Core commands in business logic can be a little different, which is why I've written this chapter.

This chapter describes a pattern for handling business logic that compartmentalizes some of the complexity in order to reduce the load on you, the developer. You'll also learn several techniques for writing business logic that uses EF Core to access the database. These techniques range from using software classes for validation to standardizing your business logic's interface in order to make frontend code simpler. The overall aim is to help you quickly write accurate, understandable, and well-performing business logic.

4.1 Why is business logic so different from other code?

Our CRUD code in chapters 2 and 3 adapted and transformed data as it moved into and out of the database. Some of that code got a little complex, and I showed you the Query Object pattern to make a large query more manageable. Convesely, business logic can reach a whole new level of complexity. Here's a quote from one of the leading books on writing business logic:

> *The heart of software is its ability to solve domain (business)-related problems for its users. All other features, vital though they may be, support this basic purpose. When the domain is complex, this is a difficult task, calling for the concentrated effort of talented and skilled people.*
>
> *Eric Evans, Domain-Driven Design*[1]

Over the years, I've written quite a bit of complex business logic, and I've found Eric Evan's comment "this is a difficult task" to be true. When I came back to software development after a long gap, the first applications I wrote were for geographic modeling and optimization, which have complex business rules. The business code I wrote ended up being hundreds of lines long, all intertwined. The code worked, but it was hard to understand, debug, and maintain.

So, yes, you can write business logic just like any other bit of code, but there's a case for a more thought-through approach. Here are a few of the questions you should ask when writing business logic:

- Do you fully understand the business rule you're implementing?
- Are there any edge cases or exceptions that you need to cover?
- How can you prove that your implementation is correct?
- How easy will it be to change your code if the business rules change?
- Will you, or someone else, understand the code if it needs changing later?

[1] *Domain-Driven Design: Tackling Complexity in the Heart of Software* was published in 2003 by Addison-Wesley Professional.

4.2 Our business need—processing an order for books

Let's start by describing the business issue that we want to implement. The example you'll use is handling a user's order for books. Figure 4.1 shows the checkout page of our book app. You're going to implement the code that runs when the user clicks the Purchase button.

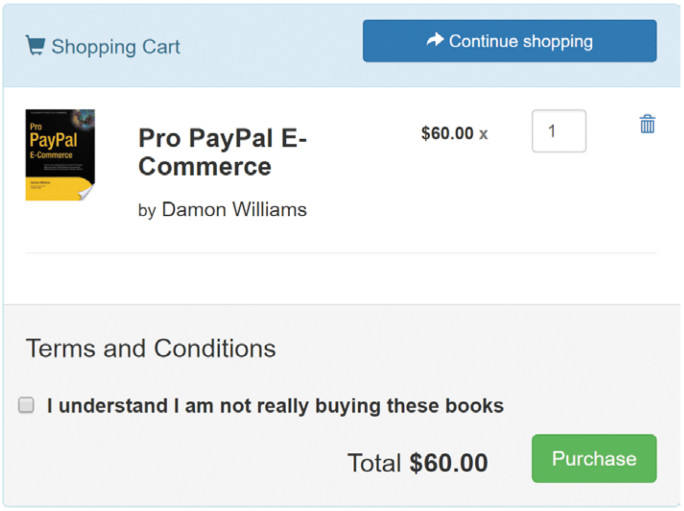


Figure 4.1 The checkout page of the book app. Clicking Purchase calls the business logic to create the order.

> **NOTE** You can try the checkout process on the live site at http://efcoreinaction.com/. The site uses an HTTP cookie to hold your basket and your identity (which saves you from having to log in). No money needed—as the Terms and Conditions says, you aren't actually going to buy a book.

4.2.1 The business rules that you need to implement

The following list gives the rules set for this business need. As I'm sure you can imagine, a real order-processing piece of business logic would have a lot more steps, especially on payment and shipping, but these six rules are enough for this example:

1. The Terms and Conditions box must be ticked.
2. An order must include at least one book.
3. A book must be available for sale, as defined by the price being positive in value.
4. The price of the book must be copied to the order, because the price could change later.
5. The order must remember the person who ordered the books.
6. Good feedback must be provided to the user so they can fix any problems in the order.

The quality and quantity of the business rules will change with the project. The preceding rules aren't bad, but they don't cover things such as what to do if the book selected by the user has been removed (unlikely, but possible), nor how to weed out malicious input. This is where you, as a developer, need to think through the problem and try to anticipate issues.

4.3 Using a design pattern to help implement business logic

Before you start writing code to process an order, you should describe a pattern that you're going to follow. This pattern helps you to write, test, and performance-tune your business logic. The pattern is based on the domain-driven design (DDD) concepts expounded by Eric Evans, but where the business logic code isn't inside the entity classes. This is known as a *transactions script* or *procedural* pattern of business logic because the code is contained in a standalone method.

This procedural pattern is easier to understand and uses the basic EF Core commands you have already seen. But many see the procedural approach as a DDD antipattern, known as an *anemic domain model* (see www.martinfowler.com/bliki/AnemicDomainModel.html). After you have learned about EF Core's *backing field* feature and the DDD entity pattern, you will extend this approach to a fully DDD design in section 10.4.2.

This section, and section 10.4, present my interpretation of Eric Evans' DDD approach, and plenty of other ways for applying DDD with EF. Although I offer my approach, which I hope will help some of you, don't be afraid to look for other approaches.

4.3.1 Five guidelines for building business logic that uses EF Core

The following list explains the five guidelines that make up the business logic pattern you'll be using in this chapter. Most of the pattern comes from DDD concepts, but some are the result of writing lots of complex business logic and seeing areas to improve.

1. *The business logic has first call on how the database structure is defined.* Because the problem you're trying to solve (called the *domain model* by Eric Evans) is the heart of the problem, it should define the way the whole application is designed. Therefore, you try to make the database structure, and the entity classes, match your business logic data needs as much as you can.
2. *The business logic should have no distractions.* Writing the business logic is difficult enough in itself, so you isolate it from all the other application layers, other than the entity classes. When you write the business logic, you must think only about the business problem you're trying to fix. You leave the task of adapting the data for presentation to the service layer in your application.
3. *Business logic should think it's working on in-memory data.* This is something Eric Evans taught me: write your business logic as if the data is in-memory. Of course, you need to have some *load* and *save* parts, but for the core of your business logic, treat the data, as much as is practical, as if it's a normal, in-memory class or collection.

4 *Isolate the database access code into a separate project.* This fairly new rule came out of writing an e-commerce application with complex pricing and delivery rules. Before this, I used EF directly in my business logic, but I found that it was hard to maintain and difficult to performance-tune. Instead, you should use another project, which is a companion to the business logic, to hold all the database access code.

5 *The business logic shouldn't call EF Core's SaveChanges directly.* You should have a class in the service layer (or a custom library) whose job it is to run the business logic. If there are no errors, this class calls `SaveChanges`. The main reason for this rule is to have control of whether to write out the data, but there are other benefits I'll describe later.

Figure 4.2 shows the application structure you'll create to help you apply these guidelines when implementing business logic. In this case, you'll add two new projects to the original book app structure described in chapter 2:

- The pure business logic project, which holds the business logic classes that work on the in-memory data provided by its companion business database access methods.
- The business database access project, which provides a companion class for each pure business logic class that needs database access. Each companion class makes the pure business logic class think it's working on an in-memory set of data.

Figure 4.2 has five numbers, with comments, that match the five guidelines listed previously.

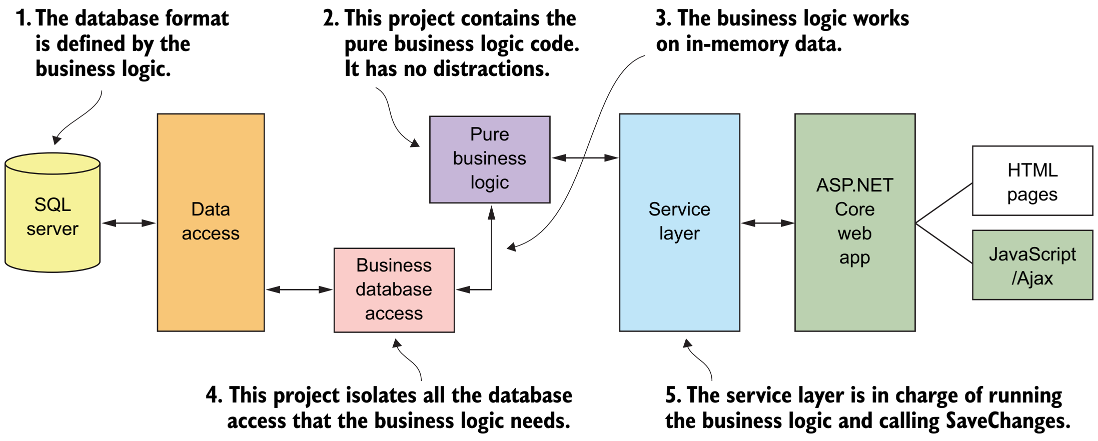

1. The database format is defined by the business logic.
2. This project contains the pure business logic code. It has no distractions.
3. The business logic works on in-memory data.
SQL server
Data access
Pure business logic
Business database access
Service layer
ASP.NET Core web app
HTML pages
JavaScript /Ajax
4. This project isolates all the database access that the business logic needs.
5. The service layer is in charge of running the business logic and calling SaveChanges.

Figure 4.2 The projects inside our book app, with two new projects for handling business logic

Does all business logic in an application live in the BizLogic layer?

In real-world applications, especially ones that interact with a human being, you want the user experience to be as great as possible. As a result, the business logic may move outside the BizLogic layer into other layers, especially the presentation layer. So, no, all business logic in an application doesn't live in the BizLogic layer.

As a developer, I find it useful to separate the distinct parts of the business rules that my clients present into three types:

- *Manipulation of a state or data*—For instance, creating an order
- *Validation rules*—For instance, checking that a book is available to buy
- *A sequence or flow*—For instance, the steps in processing an order

The manipulation of a state or data is the core business logic. The code for this manipulation can be complicated and may require a lot of design and programming effort to write. This chapter focuses on server-side business logic, but with sophisticated frontend JavaScript libraries, some data or state manipulation may move out to the frontend.

Validation of data is ubiquitous, so you find validation code cropping up in every layer of your application. In human-facing applications, I generally move the validation as far forward as possible so that the user gets feedback quickly. But, as you'll see in the examples, plenty of extra validation can exist in the business logic.

A sequence or flow is often shown to a human user as a sequence of pages or steps in a wizard, but backed up by the data manipulations that each stage needs done by some sort of CRUD and/or business logic.

None of this invalidates the approach of having a specific area in your server-side application dedicated to business logic. There's plenty of complex code to write, and having a zone where business rules are the number one focus helps you to write better code.

4.4 Implementing the business logic for processing an order

Now that I've described the business need, with its business rules, and the pattern you're going to use, you're ready to write code. The aim is to break the implementation into smaller steps that focus on specific parts of the problem at hand. You'll see how this business logic pattern helps you to focus on each part of the implementation in turn.

You're going to implement the code in sections that match the five guidelines listed in section 4.3.1. At the end, you'll see how this combined code is called from the ASP .NET Core application that the book app is using.

4.4.1 Guideline 1: Business logic has first call on defining the database structure

This guideline says that the design of the database should follow the business needs—in this case, represented by six business rules. The three rules that are relevant to the database design are as follows:

- An order must include at least one book (implying there can be more).
- The price of the book must be copied to the order, because the price could change later.
- The order must remember the person who ordered the books.

From this, you come up with a fairly standard design for an order, with an `Order` entity class that has a collection of `LineItem` entity classes—a one-to-many relationship. The `Order` entity class holds the information about the person placing the order, while each `LineItem` entity class holds a reference to the book order, how many, and at what price.

Figure 4.3 shows what these two tables, LineItem and Orders, look like in the database. To make the image more understandable, I show the Books table (in gray) that each LineItem row references.

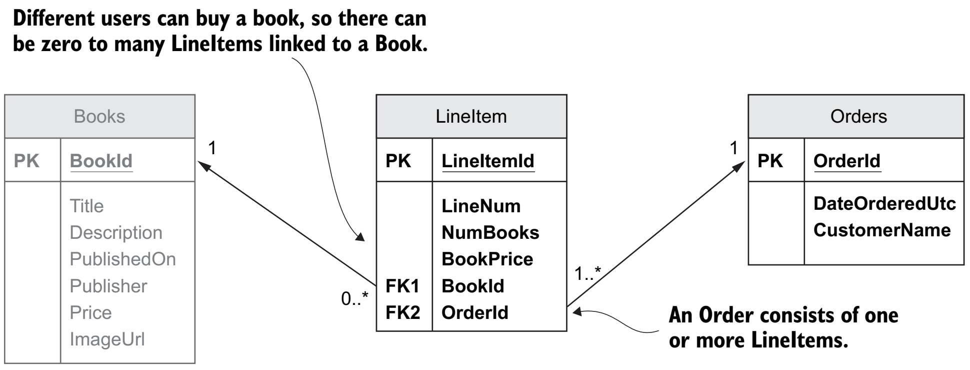


Figure 4.3 The new LineItem and Orders tables added to allow orders for books to be taken

NOTE The Orders table name is plural because you added a `DbSet<Order> Orders` property to the application's DbContext, and EF Core, by default, uses the property name, `Orders`, as the table name. You haven't added a property for the `LineItem` entity class because it's accessed via the Order's relational link. In that case, EF Core, by default, uses the class name, `LineItem`, as the table name.

4.4.2 Guideline 2: Business logic should have no distractions

Now you're at the heart of the business logic code, and the code here will do most of the work. It's going to be the hardest part of the implementation that you write, but you want to help yourself by cutting off any distractions. That way, you can stay focused on the problem.

You do this by writing the pure business code with reference to only two other parts of the system: the entity classes shown in figure 4.3, `Order`, `LineItem`, and `Book`, and your companion class that will handle all the database accesses. Even with this minimization of scope, you're still going to break the job into a few parts.

CHECKING FOR ERRORS AND FEEDING THEM BACK TO THE USER—VALIDATION

The business rules contain several checks, such as "The Terms and Conditions box must be ticked." And they also say you need to give good feedback to the person, so that they can fix any problems and complete their purchase. These sorts of checks, called *validation*, are common throughout an application.

To help, you'll create a small abstract class called `BizActionErrors`, shown in listing 4.1. This provides a common error-handling interface for all your business logic. This class contains a C# method called `AddError` that the business logic can call to add an error, and an immutable list (a list that can't be changed) called `Errors`, which holds all the validation errors found while running the business logic.

You'll use a class called `ValidationResult` for storing each error because it's the standard way of returning errors with optional, additional information on what exact property the error was related to. Using the `ValidationResult` class instead of a simple string fits in with another validation method you'll add later in this chapter.

NOTE You have two main approaches to handling the passing of errors back up to higher levels. One is to throw an exception when an error occurs, and the other is to pass back the errors to the caller. Each has its own advantages and disadvantages; this example uses the second approach—passing the errors back for the higher level to check.

Listing 4.1 Abstract base class providing error handling for your business logic

```
public abstract class BizActionErrors
{
    private readonly List<ValidationResult> _errors
        = new List<ValidationResult>();

    public IImmutableList<ValidationResult>
        Errors => _errors.ToImmutableList();

    public bool HasErrors => _errors.Any();

    protected void AddError(string errorMessage,
        params string[] propertyNames)
    {
        _errors.Add( new ValidationResult
            (errorMessage, propertyNames));
    }
}
```

Holds the list of validation errors privately

Abstract class that provides error handling for business logic

Provides a public, immutable list of errors

Creates a bool HasErrors to make checking for errors easier

Allows a simple error message, or an error message with properties linked to it, to be added to the errors list.

Validation result has an error message and a possibly empty list of properties it's linked to.

Using this abstract class means your business logic is easier to write and all your business logic has a consistent way of handling errors. The other advantage is that you can change the way errors are handled internally without having to change any of your business logic code.

Your business logic for handling an order does a lot of validation; that's typical for an order, because it often involves money. Other business logic may not do any tests, but the base class `BizActionErrors` will automatically return a `HasErrors` of `false`, which means all business logic can be dealt with in the same way.

4.4.3 Guideline 3: Business logic should think it's working on in-memory data

Now you'll start on the main class, `PlaceOrderAction`, that contains the pure business logic. It relies on the companion class, `PlaceOrderDbAccess`, to present the data as an in-memory set (in this case, a dictionary) and to take the created order and write it to the database. Although you're not trying to hide the database from the pure business logic, you do want it to work as if the data were normal .NET classes.

Listing 4.2 shows the `PlaceOrderAction` class, which inherits the abstract class `BizActionErrors` to handle returning error messages to the user. It also uses two methods that the companion `PlaceOrderDbAccess` class provides:

- `FindBooksByIdsWithPriceOffers`—Takes the list of `BookIds` and returns a dictionary with the `BookId` as the key and the `Book` entity class as the value (`null` if no book found), and any associated `PriceOffers`
- `Add`—Adds the `Order` entity class with its `LineItem` collection to the database

Listing 4.2 `PlaceOrderAction` class contains build-a-new-order business logic

Makes the business logic conform to a standard interface for business logic that has an input and an output

Provides all the error handling required for the business logic

```
public class PlaceOrderAction :
    BizActionErrors,
    IBizAction<PlaceOrderInDto,Order>
{
    private readonly IPlaceOrderDbAccess _dbAccess;

    public PlaceOrderAction(IPlaceOrderDbAccess dbAccess)
    {
        _dbAccess = dbAccess;
    }

    public Order Action(PlaceOrderInDto dto)
    {
        if (!dto.AcceptTAndCs)
        {
            AddError(
"You must accept the T&Cs to place an order.");
            return null;
        }
```

Needs the companion PlaceOrderDbAccess class to handle all the database accesses

The method called by BizRunner to execute this business logic.

You start with basic validation.

```
            if (!dto.LineItems.Any())
            {
                AddError("No items in your basket.");
                return null;
            }
```

You start with basic validation.

You ask the PlaceOrderDbAccess class to find all the books you need, with any optional PriceOffers.

Creates the Order entity class. Calls the private method FormLineItemsWithErrorChecking, which creates the LineItems.

```
            var booksDict =
                _dbAccess.FindBooksByIdsWithPriceOffers
                     (dto.LineItems.Select(x => x.BookId));
            var order = new Order
            {
                CustomerName = dto.UserId,
                LineItems =
                    FormLineItemsWithErrorChecking
                         (dto.LineItems, booksDict)
            };

            if (!HasErrors)
                _dbAccess.Add(order);
```

Adds the order to the database only if there are no errors

```
            return HasErrors ? null : order;
        }
```

If there are errors, you return null; otherwise, you return the order.

```
        private List<LineItem>  FormLineItemsWithErrorChecking
            (IEnumerable<OrderLineItem> lineItems,
             IDictionary<int,Book> booksDict)
```

Private method handles the creation of each LineItem entity class for each book ordered.

```
        {
            var result = new List<LineItem>();
            var i = 1;
```

Goes through each book type that the person has ordered

```
            foreach (var lineItem in lineItems)
            {
                if (!booksDict.
                    ContainsKey(lineItem.BookId))
                        throw new InvalidOperationException
       ("An order failed because book, " +
        $"id = {lineItem.BookId} was missing.");
```

Treats a book being missing as a system error and throws an exception.

Calculates the price at the time of the order

```
                var book = booksDict[lineItem.BookId];
                var bookPrice =
                    book.Promotion?.NewPrice ?? book.Price;
                if (bookPrice <= 0)
                    AddError(
       $"Sorry, the book '{book.Title}' is not for sale.");
                else
                {
                    //Valid, so add to the order
```

More validation where you check that the book can be sold

```
                    result.Add(new LineItem
                    {
                        BookPrice = bookPrice,
                        ChosenBook = book,
                        LineNum = (byte)(i++),
                        NumBooks = lineItem.NumBooks
                    });
                }
            }
            return result;
        }
    }
```

All is OK, so now you can create the LineItem entity class with the details.

Returns all the LineItems for this order

You'll notice that you add another check that the book selected by the person is still in the database. This wasn't in the business rules, but this could occur, especially if malicious inputs were provided. In this case, you make a distinction between errors that the user can correct, which are returned by the `Errors` property, and system errors (in this case, a book being missing), for which you throw an exception that the system should log.

You may have seen at the top of the class that you apply an interface in the form of `IBizAction<PlaceOrderInDto,Order>`. This ensures that this business logic class conforms to a standard interface you use across all your business logic. You'll see this later when you create a generic class to run and check the business logic.

4.4.4 Guideline 4: Isolate the database access code into a separate project

Our guideline says to put all the database access code that the business logic needs into a separate, companion class. This ensures that the database accesses are all in one place, which makes testing, refactoring, and performance tuning much easier.

Another benefit that a reader of my blog noted is that this guideline can help if you're working with an existing, older database. In this case, the database entities may not be a good match for the business logic you want to write. If so, you can use the BizDbAccess methods as an *Adapter pattern* that converts the older database structure to a form more easily processed by your business logic.

DEFINITION The *Adapter pattern* converts the interface of a class into another interface that the client expects. This pattern lets classes work together that couldn't otherwise do so because of incompatible interfaces. See https://sourcemaking.com/design_patterns/adapter.

You make sure that your pure business logic, class `PlaceOrderAction`, and your business database access class `PlaceOrderDbAccess` are in separate projects. That allows you to exclude any EF Core libraries from the pure business logic project, which ensures that all database access is done via the companion class, `PlaceOrderDbAccess`. In my own projects, I split the entity classes into a separate project from the EF code. Then my business logic accesses only the project containing the entity classes, and not the project that contains EF Core. For simplicity, the example code holds the entity classes in the same project as the application's DbContext. Listing 4.3 shows our `PlaceOrderDbAccess` class, which implements two methods to provide the database accesses that the pure business logic needs:

1. `FindBooksByIdsWithPriceOffers` method, which finds and loads the `Book` entity class, with any optional `PriceOffer`.
2. `Add` method, which adds the finished `Order` entity class to the application's DbContext property, `Orders`, so it can be saved to the database after EF Core's `SaveChanges` method is called.

Listing 4.3 `PlaceOrderDbAccess`, which handles all the database accesses

BizLogic hands it a collection of BookIds, which the checkout has provided.

```
public class PlaceOrderDbAccess : IPlaceOrderDbAccess
{
    private readonly EfCoreContext _context;

    public PlaceOrderDbAccess(EfCoreContext context)
    {
        _context = context;
    }

    public IDictionary<int, Book>
        FindBooksByIdsWithPriceOffers
            (IEnumerable<int> bookIds)
    {
        return _context.Books
            .Where(x => bookIds.Contains(x.BookId))
            .Include(r => r.Promotion)
            .ToDictionary(key => key.BookId);
    }

    public void Add(Order newOrder)
    {
        _context.Add(newOrder);
    }
}
```

BizDbAccess needs the application's DbContext to access the database, so it's provided via the constructor

Finds all the books that the user wants to buy

Finds a book, if present, for each ID

Returns the result as a dictionary to make it easier for the BizLogic to look them up

Adds the new order that the BizLogic built into the DbContext's Orders DbSet collection

Includes any optional promotion, as the BizLogic needs that for working out the price

The `PlaceOrderDbAccess` class implements an interface called `IPlaceOrderDbAccess`, which is how the `PlaceOrderAction` class accesses this class. In addition to helping with dependency injection, which is covered in chapter 5, using an interface allows you to replace the `PlaceOrderDbAccess` class with a test version, a process called *mocking*, when you're unit testing the `PlaceOrderAction` class. Section 15.8 covers this in more detail.

4.4.5 Guideline 5: Business logic shouldn't call EF Core's SaveChanges

The final rule says that the business logic doesn't call EF Core's `SaveChanges`, which would update the database directly. There are a few reasons for this. First, you consider the service layer as the main orchestrator of database accesses: it's in command of what gets written to the database. Second, the service layer calls `SaveChanges` only if the business logic returns no errors.

To help you run your business logic, I've built a series of simple classes that I use to run any business logic; I call these *BizRunners*. They're generic classes, able to run business logic with different input and output types. Different variants of the BizRunner can handle different input/output combinations and async methods (chapter 5 covers async/await with EF Core), plus some with extra features, which are covered later in this chapter.

Each BizRunner works by defining a generic interface that the business logic must implement. Your `PlaceOrderAction` class in the BizLogic project runs an action that expects a single input parameter of type `PlaceOrderInDto` and returns an object of type `Order`. Therefore, the `PlaceOrderAction` class implements the interface as shown in the following listing, but with its input and output types (`IBizAction<PlaceOrderInDto,Order>`).

Listing 4.4 The interface that allows the BizRunner to execute business logic

```
public interface IBizAction<in TIn, out TOut>
{
    IImmutableList<ValidationResult>
        Errors { get; }
    bool HasErrors { get; }
    TOut Action(TIn dto);
}
```

BizAction has both a TIn and a TOut

Returns the error information from the business logic

The action that the BizRunner will call

By having the business logic class implement this interface, the BizRunner knows how to run that code. The BizRunner itself is small, as you'll see in the following listing, which shows that it called `RunnerWriteDb<TIn, TOut>`. This BizRunner variant is designed to work with business logic that has an input, provides an output, and writes to the database.

Listing 4.5 The BizRunner that runs the business logic and returns a result or errors

```
public class RunnerWriteDb<TIn, TOut>
{
    private readonly IBizAction<TIn, TOut> _actionClass;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors => _actionClass.Errors;
    public bool HasErrors => _actionClass.HasErrors;
```

Error information from the business logic is passed back to the user of the BizRunner

```
    public RunnerWriteDb(
        IBizAction<TIn, TOut> actionClass,
        EfCoreContext context)
    {
        _context = context;
        _actionClass = actionClass;
    }

    public TOut RunAction(TIn dataIn)
    {
        var result = _actionClass.Action(dataIn);
        if (!HasErrors)
            _context.SaveChanges();

        return result;
    }
}
```

Handles business logic that conforms to the IBizAction<TIn, TOut> interface.

Calls RunAction in your service layer, or in your presentation layer if the data comes back in the right form

Runs the business logic you gave it

If there are no errors, it calls SaveChanges to execute any add, update, or delete methods.

Returns the result that the business logic returned

The BizRunner pattern hides the business logic and presents a common interface/API that other classes can use. The caller of the BizRunner doesn't need to worry about EF Core, because all the calls to EF Core are in the BizDbAccess code or in the BizRunner. That in itself is reason enough to use it, but, as you'll see later, this BizRunner pattern allows you to create other forms of BizRunner that add extra features.

> **NOTE** You may want to check out an open source library I created, called EfCore.GenericBizRunner. This library, which is available as a NuGet package, provides a more sophisticated version of the BizRunner described in this chapter; see https://github.com/JonPSmith/EfCore.GenericBizRunner for more information.

One important point about the BizRunner is that it should be the only method allowed to call `SaveChanges` during the lifetime of the application's DbContext. Why? Because some business logic might add/update an entity class before an error is found. To stop these changes from being written to the database, you're relying on `SaveChanges` *not* being called at all during the lifetime of the application's DbContext.

In an ASP.NET application, controlling the lifetime of the application's DbContext is fairly easy to manage, because a new instance of the application's DbContext is created for each HTTP request. In longer-running applications, this is a problem. In the past, I've avoided this by making the BizRunner create a new, hidden instance of the application's DbContext so that I can be sure no other code is going to call `SaveChanges` on that DbContext instance.

4.4.6 *Putting it all together—calling the order-processing business logic*

Now that you've learned all the parts of the business logic pattern, you're ready to see how to call this code. Listing 4.6 shows the `PlaceOrderService` class in the service layer, which calls the BizRunner to execute the `PlaceOrderAction` that does the order processing. If the business logic is successful, the code clears the checkout cookie and returns the `Order` entity class key, so that a confirmation page can be shown to the user. If the order fails, it doesn't clear the checkout cookie, and the checkout page is shown again, with the error messages, so that the user can correct any problems and retry.

Listing 4.6 The `PlaceOrderService` class that calls the business logic

Handles the checkout cookie. This is a cookie, but with a specific name and expiry time.

The BizRunner you'll use to execute the business logic. It's of type RunnerWriteDb<TIn, TOut>.

```
public class PlaceOrderService
{
    private readonly CheckoutCookie _checkoutCookie;
    private readonly
        RunnerWriteDb<PlaceOrderInDto, Order> _runner;

    public IImmutableList<ValidationResult>
        Errors => _runner.Errors;
```

Holds any errors sent back from the business logic. The caller can use these to redisplay the page and show the errors that need fixing.

```
    public PlaceOrderService(
        IRequestCookieCollection cookiesIn,
        IResponseCookies cookiesOut,
        EfCoreContext context)
    {
        _checkoutCookie = new CheckoutCookie(
            cookiesIn, cookiesOut);
        _runner =
            new RunnerWriteDb<PlaceOrderInDto, Order>(
                new PlaceOrderAction(
                    new PlaceOrderDbAccess(context)),
                context);
    }
```

The constructor needs access to the cookies, both in and out, and the application's DbContext.

Creates a CheckoutCookie using the cookie in/out access parts from ASP.NET Core

Creates the BizRunner with the business logic, PlaceOrderAction, that you want to run. PlaceOrderAction needs PlaceOrderDbAccess when it's created.

```
    public int PlaceOrder(bool acceptTAndCs)
```

The method you call from the ASP.NET action that's called when the user clicks the Purchase button

```
    {
        var checkoutService = new CheckoutCookieService(
            _checkoutCookie.GetValue());
```

Encodes/decodes the checkout data into a string that goes inside the checkout cookie.

```
        var order = _runner.RunAction(
            new PlaceOrderInDto(acceptTAndCs,
            checkoutService.UserId,
            checkoutService.LineItems));

        if (_runner.HasErrors) return 0;
```

You're ready to run the business logic, handing it the checkout information in the format that it needs.

If the business logic has any errors, you return immediately. The checkout cookie hasn't been cleared, so the user can try again.

```
        //successful, so clear the cookie line items
        checkoutService.ClearAllLineItems();
        _checkoutCookie.AddOrUpdateCookie(
            checkoutService.EncodeForCookie());

        return order.OrderId;
    }
}
```

The order was placed successfully. You therefore clear the checkout cookie of the order parts.

Returns the OrderId, the primary key of the order, which ASP.NET uses to show a confirmation page that includes the order details

In addition to running the business logic, this class acts as an Adapter pattern: it transforms the data from the checkout cookie into a form that the business logic accepts, and on a successful completion, it extracts the `Order` primary key, `OrderId`, to send back to the ASP.NET Core presentation layer.

This Adapter pattern role is typical of the code that calls the business logic, because a mismatch often occurs between the presentation layer format and the business logic format. This mismatch can be small, as in this example, but you're likely to need to do some form of adaptation in all but the simplest calls to your business logic. That's why my more sophisticated EfCore.GenericBizRunner library has a built-in Adapter pattern feature.

4.4.7 Any disadvantages of this business logic pattern?

I find the business logic pattern I've described useful, yet I'm aware of a few downsides, especially for developers who are new to a DDD approach. This section presents some thoughts to help you evaluate whether this approach is for you.

The first disadvantage is that the pattern is more complicated than just writing a class with a method that you call to get the job done. This business logic pattern relies on interfaces and code/libraries such as the BizRunners, and at least four projects in your solution. For small applications, this can be overkill.

The second disadvantage is, even in medium-sized projects, you can have simple business logic that may be only 10 lines long. In this case, is it worth creating both the pure business logic class and the companion data access class? For small business logic jobs, maybe you should create one class that combines the pure business logic and the EF Core calls. But be aware: if you do this to cut corners, it can come back and bite you when you need to refactor.

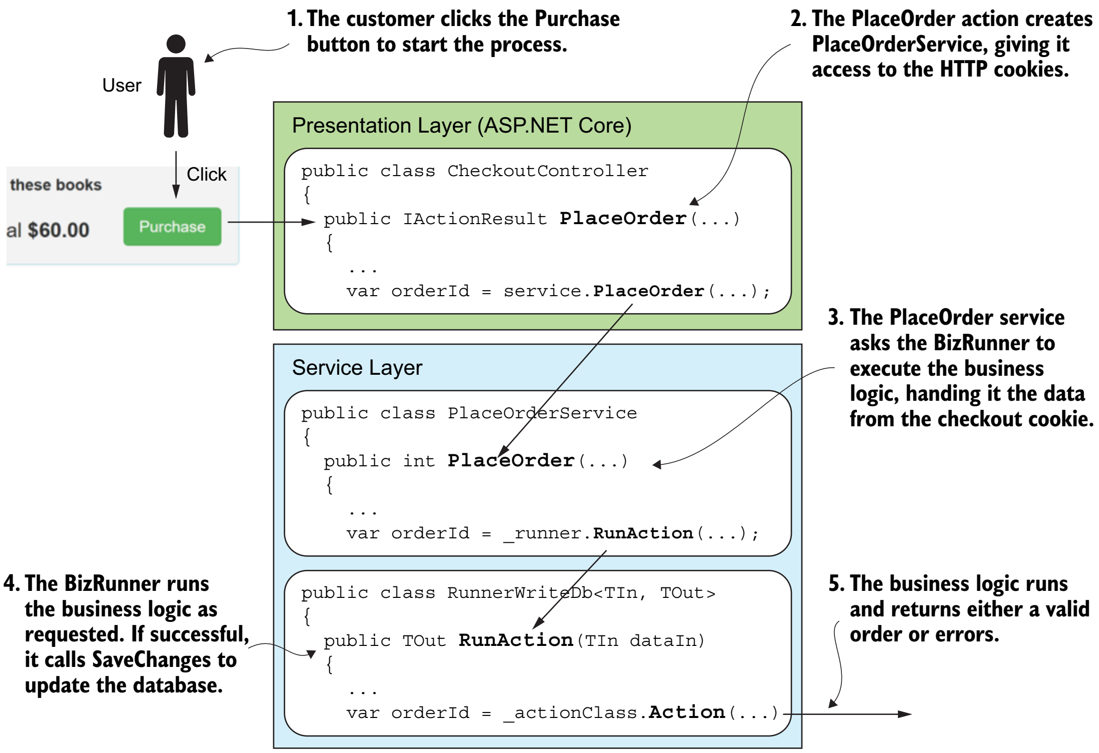
There's also a development cost inherent in the business logic pattern's guideline 2, the "no distraction" rule. The data that the business logic takes in and returns can be different from what the caller of the business logic needs. For instance, in our example, the checkout data was held in an HTTP cookie; the business logic has no concept of what a cookie is (nor should it), so the calling method had to convert the cookie content into the format that the business logic wanted. Therefore, the Adapter pattern is used a lot in the service layer to transform data between the business logic and the presentation layer—which is why I included an Adapter pattern feature in the EfCore .GenericBizRunner library.

Having listed all these disadvantages, I still find this approach far superior to my earlier approach of considering business logic as "just another piece of code." In chapter 10 I further enhance this business logic pattern once you have learned how to apply the DDD principals to the entity classes themselves. DDD-styled entity classes are "locked down"; that is, their properties have private setters and all creates/updates are done via methods inside the entity class. These methods can contain some of your business logic, which improves the overall robustness of your solution because no one can bypass your business logic by simply altering properties in the entity class. After you have learned about the features needed to truly lock down an entity class, I recommend you read about the business logic enhancements in section 10.4.2.

4.5 Placing an order on the book app

Now that we've covered the business logic for processing an order, the BizRunner, and the `PlaceOrderService` that executes the business logic, let's see how to use these in the context of the book app. Figure 4.4 shows the process, from the user clicking the Purchase button through running the business logic and returning a result.

I don't go into the presentation code in detail here, as this chapter is about using EF Core in business logic, but I do cover some of this in the next chapter, which is about using EF Core in ASP.NET Core applications.



1. The customer clicks the Purchase button to start the process.

2. The PlaceOrder action creates PlaceOrderService, giving it access to the HTTP cookies.

3. The PlaceOrder service asks the BizRunner to execute the business logic, handing it the data from the checkout cookie.

4. The BizRunner runs the business logic as requested. If successful, it calls SaveChanges to update the database.

5. The business logic runs and returns either a valid order or errors.

```
public class CheckoutController
{
  public IActionResult PlaceOrder(...)
  {
    ...
    var orderId = service.PlaceOrder(...);
```

```
public class PlaceOrderService
{
  public int PlaceOrder(...)
  {
    ...
    var orderId = _runner.RunAction(...);
```

```
public class RunnerWriteDb<TIn, TOut>
{
  public TOut RunAction(TIn dataIn)
  {
    ...
    var orderId = _actionClass.Action(...)
```

Figure 4.4 The series of steps, from the user clicking the Purchase button, to the service layer, where the BizRunner executes the business logic to process the order

From the click of the Purchase button in figure 4.4, the ASP.NET Core action, `PlaceOrder`, in the `CheckoutController` is executed. This creates a class called `PlaceOrderService` in the service layer, which holds most of the Adapter pattern logic. The caller provides that class with read/write access to the cookies, as the checkout data is held in an HTTP cookie on the user's device.

You've already seen the `PlaceOrderService` class in listing 4.6. Its `PlaceOrder` method extracts the checkout data from the HTTP cookie and creates a DTO in the form that the business logic needs. It then calls the generic BizRunner to run the business logic that it needs to execute. When the BizRunner has returned from the business logic, two routes are possible:

- *The order was successfully placed—no errors.* In this case, the `PlaceOrder` method clears the checkout cookie and returns the `OrderId` of the placed order, so that the ASP.NET Core code can show a confirmation page with a summary of the order.
- *The order was unsuccessful—errors present.* In this case, the `PlaceOrder` method returns immediately to the ASP.NET Core code. That detects that errors occurred and redisplays the checkout page, and adds the error messages so that the user can rectify them and try again.

NOTE You can try the checkout process on the live book app at http://efcore-inaction.com/ and see the results. To try the error path, don't tick the Terms and Conditions (T&C) box.

4.6 Adding extra features to your business logic handling

This pattern for handling business logic makes it easier to add extra features to your business logic handling. In this section, you'll add two features:

- Entity class validation to `SaveChanges`
- Transactions that daisy-chain a series of business logic code

These features use EF Core commands that aren't limited to business logic. Both could be used in other areas, so you might want to keep these features in mind when you're working on your application.

4.6.1 Validating the data that you write to the database

.NET contains a whole ecosystem to *validate* data, to check the value of a property against certain rules (for example, checking that an integer is within the range of 1 to 10, or that a string isn't longer than 20 characters).

EF6 If you're scanning for EF6.x changes, read the next paragraph. EF Core's `SaveChanges` doesn't validate the data before writing to the database, but this section shows how to add this back.

In the previous version of EF (EF6.x), data that was being added or updated was validated by default before writing it out to the database. In EF Core, which is aimed at being lightweight and faster, no validation occurs when adding or updating the database. The idea is that the validation is often done at the frontend, so why repeat the validation?

As you've seen, the business logic contains lots of validation code, and it's often useful to move this into the entity classes as validation checks, especially if the error is related to a specific property in the entity class. This is another case of breaking a complex set of rules into several component parts.

Listing 4.7 moves the test to check that the book is for sale into the validation code, rather than having to do it in the business logic. The listing also adds two new validation checks to show you the various forms that validation checks can take, so that the example is comprehensive.

Figure 4.5 shows the `LineItem` entity class with two types of validation added. The first is a `[Range(min,max)]` attribute, known as `DataAnnotation`, which is added to the `LineNum` property. The second validation method to apply is the `IValidatableObject` interface. This requires you to add a method called `IValidatableObject.Validate`, in which you can write your own validation rules and return errors if those rules are violated.

Listing 4.7 Validation rules applied to the `LineNum` entity class

By applying the IValidatableObject interface, the validation will call the method the interface defines.

```
public class LineItem : IValidatableObject
{
    public int LineItemId { get; set; }

    [Range(1,5, ErrorMessage =
        "This order is over the limit of 5 books.")]
    public byte LineNum { get; set; }

    public short NumBooks { get; set; }

    public decimal BookPrice { get; set; }

    // relationships

    public int OrderId { get; set; }
    public int BookId { get; set; }

    public Book ChosenBook { get; set; }

    IEnumerable<ValidationResult> IValidatableObject.Validate
        (ValidationContext validationContext)
    {
        var currContext =
            validationContext.GetService(typeof(DbContext));

        if (ChosenBook.Price < 0)
            yield return new ValidationResult(
$"Sorry, the book '{ChosenBook.Title}' is not for sale.");
```

A validation DataAnnotation. Shows your error message if the LineNum property isn't in range.

The method that the IValidatableObject interface requires you to create

Moves the Price check out of the business logic

Uses the ChosenBook link to look at the date the book was published. You can also format your own error message.

You can access the current DbContext that this database access is using. In this case, you don't use it, but you could, to get better error feedback information for the user.

```
        if (NumBooks > 100)
            yield return new ValidationResult(
If you want to order a 100 or more books"+
please phone us on 01234-5678-90",
                new[] { nameof(NumBooks) });
    }
}
```

Tests a property in this class so you can return that property with the error

I should point out that in the `IValidatableObject.Validate` method you access a property outside the `LineNum` class: the `Title` of the `ChosenBook`. You need to be careful when doing this, because you can't be sure that the relationship isn't `null`. Microsoft says that EF Core will run the internal *relationship fixup* (see figure 1.6) when `DetectChanges` is called, so this is fine when using the validation code in listing 4.8.

> **NOTE** In addition to using the extensive list of built-in validation attributes, you can create your own validation attributes by inheriting the `ValidationAttribute` class on your own class. See http://mng.bz/9ec for more on the standard validation attributes that are available and how to use the `ValidationAttribute` class.

After adding the validation rule code to your `LineItem` entity class, you need to add a validation stage to EF Core's `SaveChanges` method, called `SaveChangesWithValidation`. Although the obvious place to put this is inside the application's DbContext, you'll create an extension method instead. This will allow `SaveChangesWithValidation` to be used on any DbContext, which means you can copy this class and use it in your application.

The following listing shows this `SaveChangesWithValidation` extension method, and listing 4.9 shows the private method `ExecuteValidation` that `SaveChangesWithValidation` calls to handle the validation.

Listing 4.8 `SaveChangesWithValidation` added to the application's DbContext

```
public static ImmutableList<ValidationResult>
    SaveChangesWithValidation(this DbContext context)
{
    var result = context.ExecuteValidation();
    if (result.Any()) return result;
    context.SaveChanges();
    return result;
}
```

If there are errors, you return them immediately and don't call SaveChanges.

Returns a list of ValidationResults. If it's an empty collection, the data was saved. If it has errors, the data wasn't saved.

Defined as an extension method, which means you can call it in the same way you call SaveChanges.

Creates a private method to do the validation, as you need to apply this in SaveChangesWithValidation and SaveChangesWithValidationAsync

No errors exist, so you're going to call SaveChanges.

Returns the empty set of errors, which tells the caller that everything is OK

Listing 4.9 `SaveChangesWithValidation` calls `ExecuteValidation` method

```
private static ImmutableList<ValidationResult>
    ExecuteValidation(this DbContext context)
{
    var result = new List<ValidationResult>();
    foreach (var entry in
        context.ChangeTracker.Entries()
            .Where(e =>
               (e.State == EntityState.Added) ||
               (e.State == EntityState.Modified)))
    {
        var entity = entry.Entity;
        var valProvider = new
            ValidationDbContextServiceProvider(context);
        var valContext = new
            ValidationContext(entity, valProvider, null);
        var entityErrors = new List<ValidationResult>();
        if (!Validator.TryValidateObject(
            entity, valContext, entityErrors, true))
        {
            result.AddRange(entityErrors);
        }
    }
    return result.ToImmutableList();
}
```

Calls ChangeTracker.DetectChanges, which makes sure all your changes to the tracked entity classes are found.

Filters out only those that need to be added to, or updates the database

Creates an instance of the class that implements the IServiceProvider interface, which makes the current DbContext available in the IValidatableObject.Validate method

Calls method to find any validation errors

If there are errors, you add them to the list.

Returns the list of all the errors found

The main code is in the `ExecuteValidation` method, because you need to use it in sync and async versions of `SaveChangesWithValidation`. The call to `context.ChangeTracker.Entries` calls the DbContext's `DetectChanges` to ensure that all the changes you've made are found before the validation is run. It then looks at all the entities that have been added or modified (updated) and validates them all.

One piece of code I want to point out is that when you create `ValidationContext`, you provide your own class called `ValidationDbContextServiceProvider` (which can be found in the Git repo) that implements the `IServiceProvider` interface. This allows any entity classes that have the `IValidatableObject` interface to access the current DbContext in its `Validate` method, which could be used to gather better error feedback information or do deeper testing.

You design the `SaveChangesWithValidation` method to return the errors rather than throw an exception, which is what EF6.x did. You do this to fit in with the business logic, which returns errors as a list, not an exception. You can create a new BizRunner variant, `RunnerWriteDbWithValidation`, that uses `SaveChangesWithValidation` instead of the normal `SaveChanges`, and returns errors from the business logic or any validation errors found when writing to the database. Listing 4.10 shows the BizRunner class `RunnerWriteDbWithValidation`.

Listing 4.10 BizRunner variant, `RunnerWriteDbWithValidation`

```
public class RunnerWriteDbWithValidation<TIn, TOut>
{
    private readonly IBizAction<TIn, TOut> _actionClass;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors { get; private set; }
    public bool HasErrors => Errors.Any();

    public RunnerWriteDbWithValidation(
        IBizAction<TIn, TOut> actionClass,
        EfCoreContext context)
    {
        _context = context;
        _actionClass = actionClass;
    }

    public TOut RunAction(TIn dataIn)
    {
        var result = _actionClass.Action(dataIn);

        Errors = _actionClass.Errors;

        if (!HasErrors)
        {

            Errors =
                _context.SaveChangesWithValidation()
                    .ToImmutableList();
        }
        return result;
    }
}
```

In this version, you need your own Errors and HasErrors properties, because errors can come from two sources.

Handles business logic that conforms to the IBizAction<TIn, TOut> interface.

Calls RunAction in your service layer, or in your presentation layer if the data comes back in the right form

Runs the business logic you gave it

If there are no errors, you call SaveChangesWithValidation to execute any add, update, or delete methods.

Assigns any errors from the business logic to your local errors list

Extracts the error message part of the ValidationResults and assigns the list to your Errors

Returns the result that the business logic returned

The nice thing about this new variant of the BizRunner pattern is that it has exactly the same interface as the original, nonvalidating BizRunner. You can substitute `RunnerWriteDbWithValidation<TIn, TOut>` for the original BizRunner without needing to change the business logic or the way that the calling method executes the BizRunner.

In the next section, you'll produce yet another variant of the BizRunner that can run multiple business logic classes in such a way that, from the database write point of view, look like one single business logic method, known as a database *atomic unit*. This is possible because of the business logic pattern described at the start of this chapter.

4.6.2 Using transactions to daisy-chain a sequence of business logic code

As I said earlier, business logic can get complex. When it comes to designing and implementing a large or complex piece of business logic, you have three options:

- *Option 1*—Write one big method that does everything.

- *Option 2*—Write a few smaller methods, with one overarching method to run them in sequence.
- *Option 3*—Write a few smaller methods and get the system to run them as one unit.

Option 1 isn't normally a good idea because the method will be so hard to understand and refactor. It also has problems if parts of the business logic are used elsewhere, because you could break the DRY (don't repeat yourself) software principle.

Option 2 can work, but can have problems if later stages rely on database items written by earlier stages, because this could break the *atomic unit* rule mentioned in chapter 1: with multiple changes to the database, either they all succeed, or they all fail.

This leaves option 3, which is possible because of a feature in EF Core (and most relational databases) called *transactions*. When EF Core starts a relational database transaction, the database creates an explicit, local transaction. This has two effects. First, any writes to the database are hidden from other database users until you call the transaction `Commit` command. Second, if you decide you don't want the database writes (say, because the business logic has an error), you can discard all database writes done in the transaction by calling the transaction `RollBack` command.

Figure 4.5 shows three separate pieces of business logic being run by a class called the *transactional BizRunner*. After each piece of business logic has run, the BizRunner calls `SaveChanges`, which means anything it writes is now available for subsequent business logic stages via the local transaction. On the final stage, the business logic, Biz 3, returns errors, which causes the BizRunner to call the `RollBack` command. This has the effect of removing any database writes that Biz 1 and Biz 2 did.

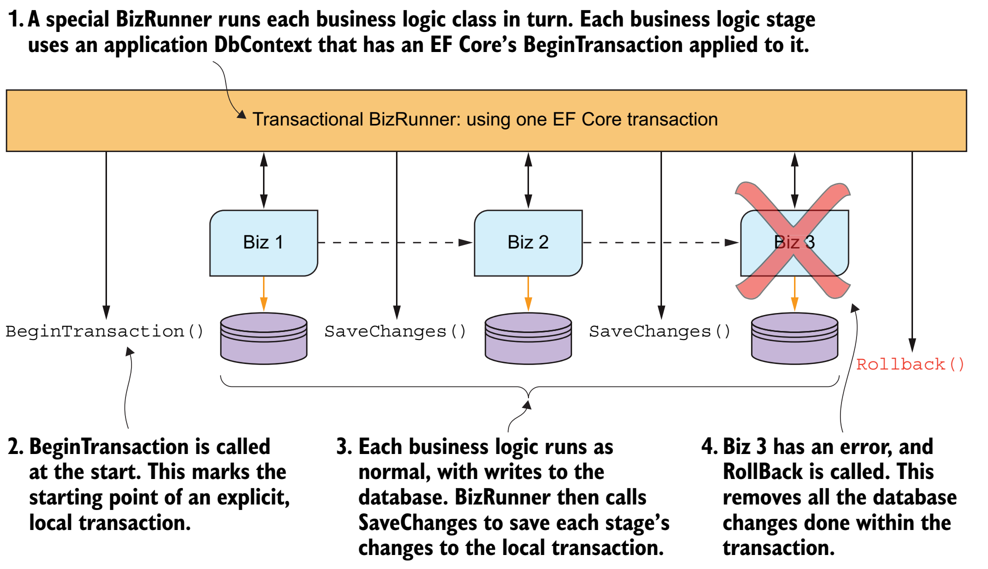

1. A special BizRunner runs each business logic class in turn. Each business logic stage uses an application DbContext that has an EF Core's BeginTransaction applied to it.
Transactional BizRunner: using one EF Core transaction
Biz 1
Biz 2
Biz 3
BeginTransaction()
SaveChanges()
SaveChanges()
Rollback()
2. BeginTransaction is called at the start. This marks the starting point of an explicit, local transaction.
3. Each business logic runs as normal, with writes to the database. BizRunner then calls SaveChanges to save each stage's changes to the local transaction.
4. Biz 3 has an error, and RollBack is called. This removes all the database changes done within the transaction.

Figure 4.5 An example of executing three separate business logic stages under one transaction. When the last business logic stage returns an error, the other database changes applied by the first two business logic stages are rolled back.

Here's the code for the new transactional BizRunner, which starts a transaction on the application's DbContext before calling any of the business logic.

Listing 4.11 `RunnerTransact2WriteDb` runs two business logic stages in series

```
public class RunnerTransact2WriteDb<TIn, TPass, TOut>
    where TOut : class
{
    private readonly IBizAction<TIn, TPass>
        _actionPart1;
    private readonly IBizAction<TPass, TOut>
        _actionPart2;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors { get; private set; }
    public bool HasErrors => Errors.Any();

    public RunnerTransact2WriteDb(
        EfCoreContext context,
        IBizAction<TIn, TPass> actionPart1,
        IBizAction<TPass, TOut> actionPart2)
    {
        _context = context;
        _actionPart1 = actionPart1;
        _actionPart2 = actionPart2;
    }

    public TOut RunAction(TIn dataIn)
    {
        using (var transaction =
            _context.Database.BeginTransaction())
        {
            var passResult = RunPart(
                _actionPart1, dataIn);
            if (HasErrors) return null;
            var result = RunPart(
                _actionPart2, passResult);

            if (!HasErrors)
            {
                transaction.Commit();
            }
            return result;
        }
    }

    private TPartOut RunPart<TPartIn, TPartOut>(
        IBizAction<TPartIn, TPartOut> bizPart,
        TPartIn dataIn)
```

Because the BizRunner returns null if an error occurs, you have to say that the TOut type must be a class.

Generic RunnerTransact2WriteDb takes three types: the initial input, the class passed from Part1 to Part2, and the final output.

Defines the generic BizAction for the two business logic parts

Holds the error information returned from the last business logic code that ran

Takes the two instances of the business logic, and the application DbContext that the business logic is using.

You start the transaction on the application's DbContext within a using statement. When it exits the using statement, unless Commit has been called, it'll RollBack any changes.

You use a private method, RunPart, to run the first business part.

If errors exist, you return null (the rollback is handled by the dispose of the transection).

Because the first part of the business logic was successful, you run the second part of the business logic.

If no errors occur, you commit the transaction to the database.

Returns the result from the last business logic

A private method that handles running each part of the business logic.

```
            where TPartOut : class
        {
            var result = bizPart.Action(dataIn);
            Errors = bizPart.Errors;
            if (!HasErrors)
            {
                _context.SaveChanges();
            }
            return result;
        }
    }
```

Runs the business logic and copies the business logic's Errors property to the local Errors property

If the business logic was successful, you call SaveChanges to apply any add/update/delete commands to the transaction.

Returns the result that the business logic returned

In your `RunnerTransact2WriteDb` class, you execute each part of the business logic in turn, and at the end of each execution, you do one of the following:

- *No errors*—You call `SaveChanges` to save to the transaction any changes the business logic has run. That save is within a local transaction, so other methods accessing the database won't see those changes yet. You then call the next part of the business logic, if there is one.
- *Has errors*—You copy the errors found by the business logic that just finished to the BizRunner error list and exit the BizRunner. At that point, the code steps outside the `using` clause that holds the transaction, which causes disposal of the transaction. The disposal will, because no transaction `Commit` has been called, cause the transaction to execute its `RollBack` method, which discards the database writes to the transaction; they're never written to the database.

If you've run all the business logic with no errors, you call the `Commit` command on the transaction. This does an *atomic update* of the database to reflect all the changes to the database that are contained in the local transaction.

USING THE RUNNERTRANSACT2WRITEDB CLASS

To test the `RunnerTransact2WriteDb` class, you'll split the order-processing code you used earlier into two parts:

- `PlaceOrderPart1`—Creates the `Order` entity, with no `LineItems`
- `PlaceOrderPart2`—Adds the `LineItems` for each book bought to the `Order` entity that was created by the `PlaceOrderPart1` class

`PlaceOrderPart1` and `PlaceOrderPart2` are based on the `PlaceOrderAction` code you've already seen, so I don't repeat the business code here.

Listing 4.12 shows you the code changes that are required to `PlaceOrderService` (shown in listing 4.6) to change over to using the `RunnerTransact2WriteDb` BizRunner. The listing focuses on the part that creates and runs the two stages, Part1 and Part2, with the unchanged parts of the code left out so you can easily see the changes.

Listing 4.12 The `PlaceOrderServiceTransact` class showing the changed parts

```
public class PlaceOrderServiceTransact
{
    //… code removed as the same as in listing 4.5

    public PlaceOrderServiceTransact(
        IRequestCookieCollection cookiesIn,
        IResponseCookies cookiesOut,
        EfCoreContext context)
    {
        _checkoutCookie = new CheckoutCookie(
            cookiesIn, cookiesOut);
        _runner = new RunnerTransact2WriteDb
            <PlaceOrderInDto, Part1ToPart2Dto, Order>(
            context,
            new PlaceOrderPart1(
                new PlaceOrderDbAccess(context)),
            new PlaceOrderPart2(
                new PlaceOrderDbAccess(context)));
    }

    public int PlaceOrder(bool tsAndCsAccepted)
    {
        //… code removed as the same as in listing 4.6
    }
}
```

A version of PlaceOrderService, but using transactions to execute the business logic in two parts

Creates the BizRunner variant called RunnerTransact2WriteDb, which runs the two business logic parts inside a transaction

The BizRunner needs to know the data types used for input, passing from part 1 to part 2, and output.

The BizRunner needs the application's DbContext.

Provides an instance of the first part of the business logic

Provides an instance of the second part of the business logic

The important thing to note is that the business logic has no idea whether it's running in a transaction. You can use a piece of business logic on its own or as part of a transaction. Similarly, listing 4.12 shows that only the caller of transaction-based business logic, what I call the BizRunner, needs to change. This makes it easy to combine multiple business logic classes under one transaction without the need to change any of your business logic code at all.

The advantage of using transactions like this is that you can split up and/or reuse parts of your business logic while still making these multiple business logic calls look to your application, especially its database, like one call. I've used this approach when I needed to create and then immediately update a complex, multipart entity. Because I needed the Update business logic for other cases, I used a transaction to call the Create business logic followed by the Update business logic. That saved me development effort and kept my code DRY.

The disadvantage of this approach is that it adds complexity to the database access. That might make debugging a little more difficult, or the use of database transactions could cause a performance issue. These are normally small issues, but you should be aware of them if you use this approach.

Summary

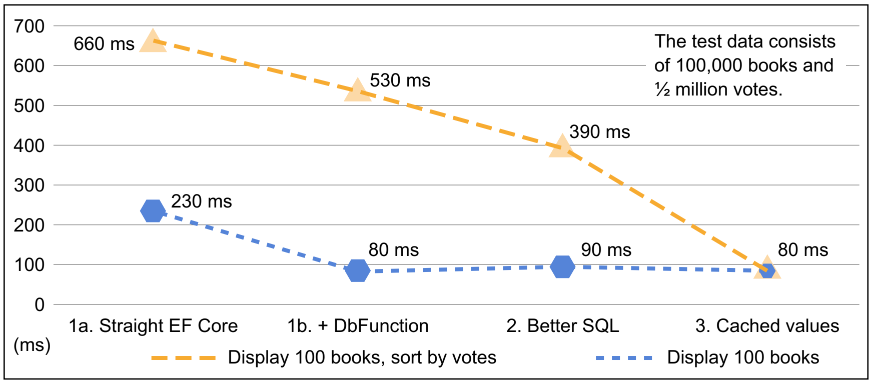
- The term *business logic* describes code written to implement real-world business rules. This type of code can be complex and difficult to write.
- Various approaches and patterns can make business logic easier to write, test, and performance-tune.
- Isolating the database access part of your business logic into another class/project can make the pure business logic simpler to write, and helps when performance tuning.
- Creating a standardized interface for your business logic makes calling and running the business logic much simpler for the frontend.
- Sometimes it's easier to move some of the validation logic into the entity classes and run the checks when that data is being written to the database.
- For business logic that's complex or being reused, it might be simpler to use a database transaction to allow a sequence of business logic parts to be run in sequence, but, from the database point of view, look like one atomic unit.

For readers who are familiar with EF6.x:

- Unlike EF6.x, EF Core's `SaveChanges` method doesn't validate data before it's written to the database. But it's easy to implement a method that provides this feature.

EFC Core performance issue checklist: the section that discusses each issue is listed.

Speed performance issues	Section
Have you picked the right feature to performance tune?	12.1.2
Are you loading too many columns?	12.4.1
Are you loading too many rows?	12.4.2
Are you using lazy loading?	12.4.3
Are you telling EF Core that your query is read-only?	12.4.4
Are you making too many calls to the database?	12.5.1
Are you calling SaveChanged multiple times?	12.5.2
Is part of your query being run in software?	12.5.3
Could you improve the SQL with a DbFunction?	12.5.4
Could pre-compiled queries help?	12.5.5
Have you checked the SQL that EF Core has produced?	12.5.6
Are you using the Find method to load via primary key?	12.5.7
Would an index help with sorting or filtering?	12.5.8
Do you have a mismatch on database types?	12.5.9
Are you making Detect Changes work too hard?	12.6.1
Would turning one DbContext into multiple DbContexts help?	12.6.2



Worked example of performance improvement with four stages, from Chapter 13

MICROSOFT .NET

Entity Framework Core IN ACTION

Jon P Smith

Free eBook
See first page

There's a mismatch in the way OO programs and relational databases represent data. Entity Framework is an object-relational mapper (ORM) that bridges this gap, making it radically easier to query and write to databases from a .NET application. EF creates a data model that matches the structure of your OO code so you can query and write to your database using standard LINQ commands. It will even automatically generate the model from your database schema.

Using crystal-clear explanations, real-world examples, and around 100 diagrams, **Entity Framework Core in Action** teaches you how to access and update relational data from .NET applications. You'll start with a clear breakdown of Entity Framework, along with the mental model behind ORM. Then you'll discover time-saving patterns and best practices for security, performance tuning, and even unit testing. As you go, you'll address common data access challenges and learn how to handle them with Entity Framework.

What's Inside

- Querying a relational database with LINQ
- Using EF Core in business logic
- Integrating EF with existing C# applications
- Applying domain-driven design to EF Core
- Getting the best performance out of EF Core
- Covers EF Core 2.0 and 2.1

For .NET developers with some awareness of how relational databases work.

Jon P Smith is a full-stack developer with special focus on .NET Core and Azure.

"An expertly written guide to EF Core—quite possibly the only reference you'll ever need."
—Stephen Byrne, Action Point

"A solid book that deals well with the topic at hand, but also handles the wider concerns around using EF in real-world applications."
—Sebastian Rogers
Simple Innovations

"This is the next step beyond the basics. It'll help you get to the next level!"
—Jeff Smith, Agilify Automation

"Great book with excellent, real-world examples."
—Tanya Wilke, Sanlam

ISBN-13: 978-1-61729-456-3
ISBN-10: 1-61729-456-X
54999
9 781617 294563

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]