Using CSLA 2019

CSLA .NET Overview

Rockford Lhotka

Using CSLA 2019: CSLA .NET Overview

Copyright © 2019 by Marimer LLC

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner.

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, the author shall not have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book (CSLA .NET version 4.9.0) is available at http:/cslanet.com.

Errata or other comments about this book should be emailed to errata@lhotka.net.

Revision: 0.1

Acknowledgements

Neither this book, nor CSLA .NET, would have been possible without support from Magenic. Magenic helps you get your digital products to market faster.

You can reach Magenic at http://www.magenic.com.

CSLA .NET has attracted a community of very thoughtful, intelligent and dedicated people. You can find many of them at https://github.com/marimerllc/cslaforum.

The bug fixes and feature enhancements described in this book come, in no small part, through the encouragement and feedback provided by this stellar community.

Thank you all!

About the Author

Rockford Lhotka is the author of more than 20 books on developing software using the Microsoft platform and technologies. He is a member of the Microsoft Regional Director and Microsoft MVP programs. Rockford speaks at many conferences and user groups around the world. He is the CTO at Magenic (www.magenic.com), a company that specializes getting your digital products to market faster.

Contents

Copyright © 2019 Marimer LLC

Copyright © 2019 Marimer LLC

Copyright © 2019 Marimer LLC

List of Tables

List of Figures

Copyright © 2019 Marimer LLC

Copyright © 2019 Marimer LLC

Chapter 1:
Introduction and Installation

Welcome to *Using CSLA 2019: CSLA .NET Overview*.

CSLA .NET is an architecture and framework designed to provide you with a home for your business logic. Microsoft's .NET and related tools provide numerous frameworks to create user experiences for the web, Windows, mobile, and other platforms. And they provide you with numerous frameworks for interacting with databases and other data stores. What they don't provide is a uniform or standard way to implement, maintain, and reuse your business logic.

That's where CSLA comes into the picture. Following the CSLA architecture, and leveraging the CSLA framework, allows you to create a true *business layer* in your application. This business layer is platform independent from the UI/presentation layer and the data access/storage layer. It can be deployed to servers, client devices, or both, as needed by your application's deployment model and physical architecture.

This book is an introduction to CSLA in 2019 and will provide you with an overview of the architecture, framework, goals and most basic usage scenarios.

This book is part of a multi-part book series, consisting of several related books. Although each book is separate, they are designed to work together to provide information about CSLA 4.9. This book series will show you how to use the CSLA framework to build powerful and scalable applications for all platforms supported by Microsoft .NET, including Windows, iOS, Android, Linux, and OS X.

CSLA .NET is a comprehensive framework, which means it has many options and can support many more advanced scenarios. Other books in the series provide much more depth for specific technologies and usage scenarios.

Before Reading this Book

CSLA 4.9 is the latest version of CSLA .NET, a software development framework for Microsoft .NET that has been available since 2001. Version 4.9 of the framework provides support for .NET Standard 2.0, .NET Core 2.1, .NET Framework 4.6.1 and higher, the Universal Windows Platform, and Xamarin. It is designed to work with Visual Studio 2017.

> CSLA started out as an acronym standing for Component-based, Scalable, Logical Architecture. These days CSLA is better thought of as an abstract name for the framework.

The CSLA .NET framework is licensed with the MIT open source license available at https://github.com/MarimerLLC/csla/blob/master/license.md.

Organization of the Book

This book is the first in a series of related books that together comprise the *Using CSLA 2019* book. Each book is separate, but they are designed to work together in a flexible manner so you can get just the information you need for your application or project. All subsequent books in the series assume you have read the first book: *Using CSLA 2019: CSLA .NET Overview*.

This book starts by providing an overview of the CSLA .NET framework, including its vision, goals and guiding philosophy. The rest of the book is a very quick walkthrough demonstrating how a very basic application can be constructed using the framework, including UWP, Xamarin Forms, and ASP.NET Core interfaces.

Other books in the series provide more depth in specific technologies and usage scenarios, as well as more in-depth and focused sample applications.

CSLA 4.9 and Previous Versions of CSLA .NET

CSLA .NET 1.0 was released concurrent with the release of Microsoft .NET 1.0 in 2002. Since that time, CSLA .NET has continually evolved and grown to accommodate changes to the .NET framework, and in response to feedback from the CSLA .NET user community.

CSLA 4.9 is the latest release of the framework, with support for Visual Studio 2017, .NET Standard 2.0, .NET Core 2.1, .NET Framework 4.6.1, Xamarin, and the Universal Windows Platform. The most important changes from CSLA .NET 3.8 to CSLA .NET were in response to feedback from the broad CSLA community, including a revamped and powerful business rules engine. From 4.0 to 4.9 most changes have been driven by changes in the .NET platform, including .NET Standard, .NET Core, the Task Parallel Library (TPL), the `async` and `await` keywords, the Microsoft's embrace and support for cross-platform .NET.

Summary of Changes and Enhancements

CSLA 4.9 includes some substantial changes from CSLA .NET which involve parts of the CSLA .NET framework code and enable some powerful capabilities for your business development efforts.

At a high level, the changes can be grouped into a set of functional enhancements as listed in Table 1.

Enhancement	Summary
Unified codebase	The CSLA codebase has been consolidated into a single .NET Standard 2.0 code base that is shared by all platform targets supported by CSLA .NET. This helps ensure consistency in code and implementation across all platforms.
TPL and `async`/`await`	All appropriate code in the CSLA framework now supports the use of the `async` and `await` keywords. Most notably, the data portal now supports async operations, as does the business rules engine.

Universal Windows Platform (UWP)	CSLA may be used to build apps that run on UWP, including Windows 10, Windows 10 Mobile, Hololens, Xbox, and some Internet of Things (IoT) scenarios.
.NET Core	CSLA may be used to build apps that run on .NET Core 2, including .NET Core Console apps as well as ASP.NET Core. This support includes Windows, Linux, and macOS.
Xamarin	CSLA may be used to build apps that run on Xamarin, including iOS, Android, macOS, and Linux desktops.
WebAssembly	CSLA supports the experimental mono on wasm technology, enabling creation of apps that run on Blazor and other .NET on WebAssembly platforms.

Table 1. Functional enhancements in CSLA 4.9

Beyond these highlights, there are numerous other changes, all of which are listed in the change log document available from the CSLA .NET releases page

https://github.com/MarimerLLC/csla/releases

Technical Requirements

CSLA 4.9 is designed to run on any platform that implements .NET Standard 2.0. It also supports .NET Framework 4 or higher to enable a level of backward compatibility. Building CSLA .NET requires Visual Studio 2017.

Windows 10 and the Universal Windows Platform (UWP) SDK must be installed to build the UWP projects in the solution.

Xamarin is required to build the Xamarin projects in the solution. This may be a separate download and install on your development workstation. Additionally, if you want to build the iOS or macOS code you'll need a Mac with Xamarin installed.

ASP.NET MVC 3, 4, and 5 are required to build the ASP.NET MVC projects in the solution. These may be separate downloads and installs on your development workstation.

Entity Framework (EF) 4, 5, and 6 are required to build the EF projects in the solution. These components are available via NuGet.

The .NET Core 2.1 SDK must be installed on your development workstation to build the .NET Core, Entity Framework Core, and ASP.NET Core projects in the solution.

Installing and Building CSLA 4.9 and Samples

CSLA 4.9 is available for download in source form from GitHub. The GitHub repo also includes sample projects demonstrating how to use the framework to build Windows, Web, Xamarin, UWP, and .NET Core applications.

The recommended way to use CSLA .NET in your projects is via NuGet. All CSLA .NET assemblies are available via NuGet for every supported platform.

Downloading CSLA 4.9 Code and Samples

You can download CSLA from the CSLA .NET home page

http://cslanet.com

This page includes links so you can download the code as a zip file, or fork and clone the repo directly from GitHub.

Building CSLA 4.9

It is best to fork and clone the GitHub repo if you want to build CSLA .NET on your development workstation. This will ensure that you have the correct code and folder structure on your workstation.

The `Csla.build` solution in the download can be opened in Visual Studio 2017. It includes the CSLA projects for all supported platforms.

If you do not have the required prerequistes (such as UWP or Xamarin) installed, the projects requiring those technologies will fail to load in Visual Studio and you will see warning dialogs. You can safely ignore those warnings and build the other projects in the solution.

The projects all generate their output in a `Bin` folder one level higher than the `Source` folder that contains the `csla.build.sln` file. When you build the solution, you'll end up with folders like this:

```
\Bin\Release\Analyzers
\Bin\Release\NET4
\Bin\Release\NET45
\Bin\Release\NET46
\Bin\Release\netstandard
```

Each folder contains the compiled assemblies that target the specific platform. The actual code is contained in unified set of Shared Projects in Visual Studio, so each platform's assemblies are largely the same, just compiled for the platform.

Building the Samples

The `Samples` download includes numerous sample applications that illustrate various aspects of CSLA .NET across the supported platforms. The specific samples available change over time based on the development team's ability to maintain them, or desire to illustrate new concepts.

The samples reference CSLA .NET via NuGet.

Some samples use SQL Server database files. Those samples require that you have SQL Server 2012 or later installed (Express or higher). You may need to update database connection strings in `app.config` or `web.config` files for those samples to run, because in some cases the full path to the `mdb` file is specified.

CSLA 4.9 provides a powerful framework for building a rich Business layer for your Windows, mobile, web, and service-oriented applications. If you are coming to CSLA 4.9 from version 3.8 it is important to realize that the new version is largely compatible with 3.8, but does include important

feature enhancements and changes, most notably a new rules engine and support for async operations in the data portal and rules engine.

If you are coming to version 4.9 from version 4, most changes are additive, and your existing code should work on your existing platforms. Some of the new features in version 4.9 are required for some of the newer platforms such as UWP or .NET Core, most notably the use of async operations.

Chapter 2: Architecture and Philosophy

Object-oriented design and programming are big topics—there are entire books devoted solely to the process of object-oriented and domain-driven design, and other books devoted to using object-oriented programming in various languages and on various programming platforms. My focus in this book isn't to teach the basics of object-oriented design or programming, but rather to show how they may be applied to the creation of distributed .NET desktop, web, and mobile applications.

It can be difficult to apply object-oriented design and programming effectively in a physically distributed environment. This chapter is intended to provide a good understanding of the key issues surrounding distributed computing as it relates to object-oriented development. I'll cover several topics, including the following:

- How logical n-tier architectures help address reuse and maintainability
- How physical n-tier architectures impact performance, scalability, security, and fault tolerance
- The difference between data-centric and object-oriented application models
- How object-oriented models help increase code reuse and application maintainability
- The effective use of objects in a distributed environment, including the concepts of anchored and mobile objects
- The relationship between an architecture and a framework

This chapter introduces the concepts and issues surrounding distributed object-oriented architecture. Then, throughout this book, we'll be exploring an n-tier architecture that may be physically distributed across multiple devices. The book will show how to use object-oriented design and programming techniques to implement a framework supporting this architecture. After that, a sample application will be created to demonstrate how the architecture and the framework support development efforts.

Logical and Physical Architecture

In today's world, an object-oriented application must be designed to work in a variety of physical configurations. Even the term "application" has become increasingly blurry due to all the hype around microservices and service-oriented architecture (SOA). If you aren't careful, you can end up building applications by combining several applications, which is obviously very confusing.

When I use the term “application” in this book, I’m referring to a set of code, objects, or components that’s considered to be part of a single, logical unit. Even if parts of the application are in different .NET assemblies or installed on different devices, all the code is viewed as being part of a singular application.

This definition works well when describing most traditional application models, such as 1-, 2-, and n-tier smart client applications (desktop and mobile), web applications and so forth. In all those cases the application consists of a set of objects or components that are designed to work together within the context of the application.

You can contrast this with a microservices or service-oriented (SOA) model, where there are multiple services (each should be viewed as a separate application) that interact through message-based communication. In service-based model the idea is to build an enterprise “system” that is composed of “applications” and “services”. In this context, both applications and services are standalone, autonomous units of functionality – which means they both meet the definition of an “application”. This means a “service” is merely an application that has an XML or JSON interface instead of an HTML or graphical interface.

So, if you are thinking about service-based systems as you read this book, the term “application” means one of two things. The first is that “application” may refer to a service implementation. The second is that “application” may refer to applications on the edge of the system that allow users to interact with the system. These edge applications are much like traditional applications, except they typically interact with services instead of databases for the retrieval and storage of data.

Traditional, service-oriented and workflow applications *might* run on a single device. But it’s very likely that they will run on multiple devices, such as a web server or a smart client and an application server. Given these varied physical environments, we’re faced with the following questions:

- Where do the objects reside?
- Are the objects designed to maintain state, or should they be stateless?
- How is object-to-relational mapping handled when retrieving or storing data in the database?
- How are database transactions managed?

Before getting into discussing some answers to these questions, it’s important to fully understand the difference between a *physical architecture* and a *logical architecture*. After that, I’ll define objects and mobile objects, and see how they fit into the architectural discussion.

When most people talk about n-tier applications, they’re talking about physical models in which the application is spread across multiple machines with different functions: a client, a web server, an application server, a database server, and so on. This isn’t a misconception—these are indeed n-tier systems. The problem is that many people tend to assume there’s a one-to-one relationship between the layers (tiers) in a logical model and the tiers in a physical model, when in fact that’s not always true.

A *physical* n-tier architecture is quite different from a *logical* n-layer architecture. An n-layer architecture has nothing to do with the number of machines or network hops involved in running

the application. Rather, a logical architecture is all about separating different types of functionality: a concept called separation of concerns.

The most common logical separation is into a Presentation layer, a Business layer, and a Data layer that may exist on a single machine, or on three separate machines—the logical architecture doesn't define those details.

> There is a relationship between an application's logical and physical architectures: the logical architecture always has at least as many layers as the physical architecture has tiers. There may be more logical layers than physical ones (because one physical tier can contain several logical layers), but never fewer.

The sad reality is that many applications have no clearly defined logical architecture. Often the logical architecture merely defaults to the number of physical tiers. This lack of a formal, logical design causes problems because it reduces flexibility. If a system is designed to operate in two, three or four physical tiers, then changing the number of physical tiers at a later date is typically very difficult. Conversely, if you start by creating a logical architecture of three or four layers, you can switch more easily between one, two, three or four physical tiers later on.

Additionally, having clean separation between these layers makes your application more maintainable because changing one layer often has minimal impact on the other layers. Nowhere is this more true than with the Presentation layer, where the ability to switch between Windows Presentation Foundation (WPF), Windows Forms, Web Forms, ASP.NET MVC (Model-View-Controller), workflow and service-based interfaces is critical.

The flexibility to choose your physical architecture is important because the benefits gained by employing a physical n-tier architecture are different from those gained by employing a logical n-layer architecture. A properly designed logical n-layer architecture provides the following benefits:

- Logically organized code
- Easier maintenance
- Better reuse of code
- Better team-development experience
- Higher clarity in coding

On the other hand, a properly chosen physical n-tier architecture can provide the following benefits:

- Performance
- Scalability
- Fault tolerance
- Flexibility of deployment
- Security

It goes almost without saying that if the physical or logical architecture of an application is designed poorly, there will be a risk of damaging the things that would have been improved had the job been done well.

N-Tier, Microservices, and SOA

It is important to realize that a physical service-oriented architecture is *not the same* as an n-tier architecture. While the concepts are different, they can be complimentary. It is also important to know that the concept of *logical n-layer* architecture is the same in a service-oriented setting as in any other type of application model.

> Remember that a logical n-layer architecture is about how you create an application. A service-oriented architecture is about how you create a system composed of applications.

When thinking about logical n-layer models, a service should have the same layers as any other application: presentation, business and data. In this case the Presentation layer consists of JSON or XML messages, but that's not a lot different from the HTML used in a web-based Presentation layer. The Business layer is much the same as in any other application, in that it contains the business logic and behaviors that make the service useful. The Data layer is much the same as in any other application, in that it stores and retrieves data as necessary.

But the physical n-tier model may not appear to translate to the service-oriented world at all. Some people would say that service orientation totally obsoletes n-tier concepts, but I disagree. Service orientation has an important set of goals around loose coupling, reuse of functionality and open communication. N-tier client/server has a complimentary set of goals around performance, avoiding duplication of code and targeted functionality. The reality is that *both models are useful,* and they complement each other.

For example, you might use a service-oriented model to create a service that is available on the Internet. But the service *implementation* might be n-tier, with the service interface on the web server, and parts of the business implementation running on a separate application server. The result is a reusable service that enjoys high performance, security and which avoids duplication of code.

Complexity

Experienced designers and developers often view a good n-tier architecture as a way of simplifying an application and reducing complexity, but this isn't necessarily the case. It's important to recognize that n-tier designs are typically *more* complex than single-tier designs. Even novice developers can visualize the design of a form or a page that retrieves data from a file and displays it to the user, but novice developers often struggle with 2-tier designs and are hopelessly lost in an n-tier environment.

With enough experience, architects and developers do typically find that the organization and structure of an n-tier model reduces complexity for large applications. That said, even a veteran n-tier developer will often find it easier to avoid n-tier models when creating a simple form to display some simple data.

The point here is that n-tier architectures only simplify the process for large applications or complex environments. They can easily complicate matters if all you're trying to do is create a small application with a few forms that will be running on someone's desktop computer. (If that desktop computer is one of hundreds or thousands in a global organization, then the *environment* may be so complex that an n-tier solution provides simplicity.)

In short, n-tier architectures help to decrease or manage complexity when *any* of these are true:

- The application is large or complex.
- The application is one of many similar or related applications that *when combined* may be large or complex.
- The environment (including deployment, support, and other factors) is large or complex.

On the other hand, n-tier architectures can increase complexity when *all* of these are true:

- The application is small or relatively simple.
- The application isn't part of a larger group of enterprise applications that are similar or related.
- The environment isn't complex.

Something to remember is that even a small application is likely to grow, and even a simple environment will often become more complex over time. The more successful your application, the more likely that one or both of these will happen. If you find yourself on the edge of choosing an n-tier solution, it's typically best to go with it. You should expect and plan for growth.

This discussion illustrates why n-tier applications are viewed as relatively complex. There are a lot of factors, technical and non-technical, that must be considered. Unfortunately, it isn't possible to say definitively when n-tier does and doesn't fit. In the end, it's a judgment call that, the application architect must make; based on the factors that affect your organization, environment and development team.

Relationship Between Logical and Physical Models

Some architectures attempt to merge logical n-layer and physical n-tier concepts. Such mergers seem attractive because they seem simpler and more straightforward, but typically they aren't good in practice—they can lead people to design applications using a logical or physical architecture that isn't best suited to their needs.

The Logical Model

When you're creating an application, it's important to start with a logical architecture that clarifies the roles of all components, separates functionality so that a team can work together effectively, and simplifies overall maintenance of the system. The logical architecture must also include enough layers so that you have flexibility in choosing a physical architecture later.

Traditionally, you would devise at least a 3-layer logical model that separates the interface, the business logic, and the data-management portions of the application. Today that's rarely enough,

because the “interface” layer is often physically split into two parts (browser and web server), and the “logic” layer is often physically split between a client or web server and an application server. Additionally, there are various application models that have been used to break the traditional Business layer into multiple parts—model-view-controller and facade-data-logic being two of the most popular at the moment.

This means that the logical layers are governed by the following rules:

- The logical architecture includes layers with the purpose of organizing components into discrete roles.
- The logical architecture must have at least as many layers as the anticipated physical deployment will have tiers.

Following these rules, most modern applications have four to six logical layers. As you’ll see, CSLA .NET is based on an architecture defined by five logical layers.

Cross-Layer Communication

Just because an application is organized into layers doesn’t mean those layers can be deployed arbitrarily on different tiers. The code in one layer communicates with the layer immediately above or below itself in the architecture. If that communication is not designed properly, it may be impossible to put a network (tier) boundary between the tiers.

For example, the boundary between the Business layer and Data layer is often highly optimized. Most applications have a network boundary between the Data layer and the rest of the application, and so modern data access technologies are good at optimizing cross-network communication in this scenario.

The boundary between the Presentation layer and Business layer is often not optimized for this purpose. Many applications make use of data binding, which is a very “chatty” technology involving many property, method and event calls between these two layers. The result is that it is often impractical and undesirable to put a network boundary between these layers.

Not all layer boundaries should be designed to enable a tier boundary. An architecture should be designed up front to enable the potential for tier boundaries in certain locations, and to disallow them in other cases. If done properly, the result is a balance between flexibility and capability.

The Physical Model

By ensuring that the logical model has enough layers to provide flexibility, you can configure your application into an appropriate physical architecture that will depend on your performance, scalability, fault tolerance, security, and deployment requirements.

The more physical tiers included, the worse the performance will be; but there is the potential to increase scalability, security, and/or fault tolerance and to meet flexible deployment requirements.

Performance and Scalability

The more physical tiers there are, the *worse* the performance? That doesn’t sound right, but if you think it through, it makes perfect sense: *performance* is the speed at which an application responds to a user. This is different from *scalability*, which is a measure of how performance changes as load

(such as increased users) is added to an application. To get optimal performance—that is, the fastest possible response time for a given user—the ideal solution is to put the client, the logic, and the data on the user's device. This means no network hops, no network latency, and no contention with other users.

If you decide that you need to support multiple users, you might consider putting application data on a central file server. (This was typical with old-fashioned Access and dBASE systems, for example.) However, this immediately affects performance because of contention for access to the data file. Furthermore, data access now takes place across the network, which means you've introduced network latency and network contention, too. To overcome this problem, you could put the data into a managed environment such as SQL Server or Oracle. This will help to reduce file locking issues, but you're still stuck with the network latency and data contention problems. Although improved, performance for a given user is still nowhere near what it was when everything ran directly on that user's computer.

Even with a central database server, scalability is limited. Clients are still in contention for the resources of the server, with each client opening and closing connections, doing queries and updates, and constantly demanding the CPU, memory, and disk resources that are being used by other clients. You can reduce this load by shifting some of the work to another server. An *application server* can provide database connection pooling to minimize the number of database connections that are opened and closed. Application servers are often hosted using IIS on Windows Server or using a *platform as a service* (PaaS) cloud provider such as Azure. Such a server can also perform some data processing, filtering, and even caching to offload some work from the database server.

> It is important to realize that modern database servers can often easily handle hundreds of concurrent users in a 2-tier architecture. For most applications scalability is *not* a good reason to move from a 2- to 3-tier model.

These additional steps provide a dramatic boost to scalability, but again at the cost of performance. The user's request now has *two* network hops, potentially resulting in double the network latency and contention. For a single user, the system gets slower; but it can handle many times more users with acceptable performance levels.

In the end, the application is constrained by the most limiting resource. This is typically the speed of transferring data across the network—but if the database or application server is underpowered, they can become so slow that data transfer across the network isn't an issue. Likewise, if the application does extremely intense calculations and the client devices are slow, then the cost of transferring the data across the network to a relatively idle high-speed server can make sense.

Security

Security is a broad and complex topic, but by narrowing the discussion solely to consider how it's affected by physical n-tier decisions, it becomes more approachable. The discussion is no longer about authentication or authorization as much as it is about controlling physical access to the machines on which portions of the application will run. The number of physical tiers in an application has no impact on whether users can be authenticated or authorized, but physical tiers

can be used to increase or decrease physical access to the machines on which the application executes.

For instance, in a 2-tier Windows or ASP.NET application, the device running the UI code must have credentials to access the database server. Switching to a 3-tier model in which the data access code runs on an application server means that the device running the UI code no longer needs those credentials, potentially making the system more secure.

Security requirements vary radically based on the environment and the requirements of your application. A UWP or Windows Forms application deployed only to internal users may need relatively little security, but a Xamarin or web application accessible by anyone on the Internet may need extensive security.

To a large degree, security is all about surface area: how many points of attack are exposed from the application? The surface area can be defined in terms of domains of trust.

Security and Internal Applications

Internal applications are totally encapsulated within a domain of trust: the client and all servers are running in a trusted environment. This means that virtually every part of the application is exposed to a potential hacker (assuming that the hacker can gain physical access to a machine on the network in the first place). In a typical organization, hackers can attack the client device, the web server, the application server, and the database server if they so choose. Rarely are there firewalls or other major security roadblocks *within* the context of an organization's LAN.

> Obviously, there *is* security. It is common to use Active Directory security on the clients and servers, but there's nothing stopping someone from attempting to communicate directly with any of these machines. Within a typical LAN, users can usually connect through the network to all machines due to a lack of firewall or physical barriers.

Many internal applications are coming under increasing security requirements due to government regulations and other business pressures. The idea of having the database credentials on a client device is rapidly becoming unacceptable, and this is driving organizations to adopt a 3-tier architecture to move those credentials to a separate application server. This is an easy way to quickly improve an application's security.

The result is that the client device has the credentials to the *application* server. If they know how to find and call the application server's services, someone can use an application's own services to access its servers in invalid ways. The services in this book will be designed to prevent casual usage of the objects.

In summary, security has replaced scalability as the primary driver for moving from 2- to 3-tier architectures. But you must be careful when designing your services to ensure you haven't simply shifted the problem down a level.

Security and External Applications

For external applications, things are entirely different.

This is where microservices and SOA come into play. Service orientation is all about assembling a "system" that spans trust boundaries. In a case where part of your system is deployed outside your own network; that certainly crosses at least a security (trust) boundary.

> Microservices are simply "SOA done right". That is to say that individual services are thought of as standalone units of behavior that interact with each other via asynchronous messaging. Essentially, each service is an application with its own interface (the XML or JSON data schema), business logic, and data access.

In a client/server model, this would be viewed as a minimum of two tiers; because the client device is physically separate from any servers running behind the firewall.

Service orientation offers a better way to look at the problem: there are two totally separate applications. The client runs one application, and another application runs on your server. These two applications communicate with each other through clearly defined messages, and neither application is privy to the internal implementation of the other.

This provides a good way to not only deal with the security trust boundary, but also with the *semantic* trust boundary. What I mean by this is that the server application assumes that any data coming from the client application is flawed: either maliciously or due to a bug or oversight in the client. Even if the client has *security* access to interact with your server, the server application cannot assume that the semantic meaning of the data coming from the client is valid.

In short, because the client devices are outside the domain of trust, you should assume that they're compromised and potentially malicious. You should assume that any code running on those clients will run incorrectly or not at all; the client input must be completely validated as it enters the domain of trust, even if the client includes code to do the validation.

> I've had people tell me that this is an overly paranoid attitude, but I've been burned this way too many times. Any time an interface is exposed (Windows, web, JSON, and so on) so that clients outside your control can use it, you should assume that the interface will be misused. Often, this misuse is unintentional—for example, someone may write a buggy macro to automate data entry. That's no different than if they made a typo while entering the data by hand, but user-entered data is always validated before being accepted by an application. The same must be true for automated data entry as well, or your application will fail.

This scenario occurs in three main architectures: Smart client apps, web apps and service-oriented systems.

If you deploy a smart client application, created using something like Xamarin or WebAssembly, to external devices, it should be designed as a stand-alone application that calls your server application through services. A later book in this book series will discuss the use of service-oriented architectures with CSLA .NET in detail.

Web clients, sometimes called *single page apps* (SPA), are created with HTML, JavaScript, and many related technologies. These are basically smart client apps that run in a browser, and they often validate data or otherwise provide a rich experience for the user. Your server code should

assume that the web app didn't do anything it was supposed to. It is far too easy for a user to subvert your client-side JavaScript or otherwise bypass client-side processing—as such, nothing running in a web app can be trusted. The code running in the browser should be viewed as a *separate* application that is not trusted by the server application. Later books in this book series will cover the development of web apps using CSLA .NET.

Service-oriented systems imply one or more (potentially unknown) applications out there consuming your services. The very nature of service-oriented architectures and microservices means that you have no control over those applications, and so it would be foolish to assume they'll provide valid input to your services. A healthy dose of paranoia is critical when building any service for such a system.

As you'll see, the object-oriented concepts and techniques shown in this book can be used to create smart client applications that call services on your servers. The same concepts can be used to create the services themselves. They can also be used to create web applications ranging from simple ASP.NET MVC sites to HTML/JavaScript smart clients.

Fault Tolerance

Fault tolerance is achieved by identifying points of failure and providing redundancy. Typically, applications have numerous points of failure. Some of the most obvious are as follows:

- The network feed to your user's buildings
- The power feed to your user's buildings
- The network feed and power feed to your data center
- The primary DNS host servicing your domain
- Your firewall, routers, switches, etc.
- Your web server
- Your application server
- Your database server
- Your internal LAN

In order to achieve high levels of fault tolerance, you need to ensure that some system will instantly kick in and fill the void if any one of these fails. If the data center power goes out, a generator kicks in. If a bulldozer cuts your network feed, you'll need to have a second network feed coming in from the other side of the building, and so forth.

Considering some of the larger and more well-known outages of major websites in the past couple of years, it's worth noting that most of them occurred due to construction work cutting network or power feeds, or because their ISP or external DNS provider went down or was attacked. That said, there are plenty of examples of websites going down due to local equipment failure. The reason why the high-profile failures are seldom due to this type of problem is because large sites make sure to provide redundancy in these areas.

Clearly, adding redundant power, network, ISP, DNS, or LAN hardware will have little impact on application architecture. Adding redundant servers or using elastic cloud capabilities, on the other

hand, *will* affect the n-tier application architecture—or at least the application design. Each time a physical tier is added, you need to ensure that you add redundancy to the services in that tier. As a result, adding a fault-tolerant physical tier always means adding at least *two* servers to the infrastructure, or leveraging elastic cloud service features.

The more physical tiers, the more redundant servers there are to configure and maintain. Therefore, fault tolerance is typically expensive to achieve. Modern PaaS cloud services often reduce the cost and complexity needed to achieve fault tolerance, which is one reason the use of the cloud is increasingly common.

To achieve fault tolerance through redundancy, all servers in a tier must always also be logically identical. For example, at no time can a user be tied to a specific server, so no single server can ever maintain any user-specific information. As soon as a user is tied to a specific server, that server becomes a point of failure for that user. The result is that the user loses fault tolerance.

Achieving a high degree of fault tolerance isn't easy. It requires a great deal of thought and effort to locate all points of failure and make them redundant. Having fewer physical tiers in an architecture can assist in this process by reducing the number of tiers that must be made redundant.

To summarize, the number of physical tiers used in an architecture is a trade-off between performance, scalability, security, and fault tolerance. Furthermore, the optimal configuration for a web application isn't the same as the one for an intranet application with smart client devices. If an application framework is to have any hope of broad appeal, it needs flexibility in the physical architecture so that it can support web and smart clients effectively; as well as provide both with optimal performance and scalability. Beyond that, it needs to work well in a service-oriented environment to create both client and server applications that interact through message-based communication.

A 5-Layer Logical Architecture

CSLA .NET is based on a 5-layer logical architecture. This book series will show how you can implement the architecture using object-oriented concepts. Once the logical architecture has been created, it will be configured into various physical architectures in order to achieve optimal results for smart client, mobile, web, and service-oriented interfaces.

> If you get any group of architects into a room and ask them to describe their ideal architecture, each one will come up with a different answer. I make no pretense that this architecture is the only one out there, nor do I intend to discuss all the possible options. My aim here is to present a coherent, distributed, object-oriented architecture that supports all these different interfaces.

In the framework used for this book, the logical architecture comprises the five layers shown in Figure 1.

Figure 1. The 5-layer logical architecture

Remember that the benefit of a logical n-layer architecture is the separation of functionality into clearly defined roles or groups, in order to increase clarity and maintainability. Let's define each of the layers more carefully.

Interface

The Interface and Interface Control layers take the place of what is commonly considered the Presentation layer in a traditional 3-layer architecture. I avoid the word Presentation because in modern applications the interface may be XML or JSON instead of some GUI or HTML interface. This is important, because a good architecture needs to be able to describe both interactive applications and service-oriented applications.

At first, it may not be clear why I've broken the "Presentation" layer into two parts: Interface and Interface Control. Certainly, from a smart client perspective, the interface and the code that controls the interface are often one and the same: they are graphical user interface (GUI) forms with which the user can interact.

But with the widespread use of XAML and the rise in popularity of design patterns such as MVVM (Model-View-ViewModel), the distinction between the Interface (XAML) and Interface Control (the viewmodel code) becomes clearer, and important. This impacts smart client apps for Windows and mobile platforms, as well as Xbox, Hololens and other types of client device.

From a web perspective, the distinction is probably quite clear. In legacy web applications, the browser merely presents information to the user, and collects user input. In that case, all the interaction logic—the code written to *generate* the output, or to *interpret* user input—runs on the web server, and not on the client device.

In today's world, a web app in the browser might run a lot of code; these are smart client apps that just happen to run in a browser. But as discussed earlier in the chapter, none of this code can be trusted. It must be viewed as being a *separate* application that interacts with your application as it runs on the server. So even with a web app running in the browser, *your* server application's interface code is running on your web server.

The same is true for a service-oriented system, where the consuming application is clearly separate, and thus can't be trusted. Your application's presentation is XML or JSON messages, and your UI code is running on your server.

Knowing that the logical model must support both smart and web-based clients, it's important to recognize that in many cases, the presentation will be physically separate from the UI logic. In order to accommodate this separation, it is necessary to design the applications around this concept.

> The types of presentation technologies continue to multiply, and each comes with a new and relatively incompatible technology with which we must work. It's virtually impossible to create a programming framework that entirely abstracts presentation concepts. Because of this, the architecture and framework will merely *support the creation* of varied presentations, not automate their creation. Instead, the focus will be on simplifying the other layers in the architecture, for which technology is more stable.

Perhaps the simplest way to understand the nature of the Interface layer is to list the technologies most commonly used to create an interface:

- HTML
- XAML
- JSON
- XML
- Cocoa
- Android XML
- Windows Forms

Ideally, the Interface layer would have little or no code, and is primarily constructed using markup languages or code generation from designers in tools like Visual Studio, Android Studio, or Xcode. There are some cases where code-behind or code-beside is a requirement, but this should be avoided as much as possible.

Interface Control

Now that I've addressed the distinction between Interface and Interface Control, the latter's purpose is probably fairly clear. This layer includes the logic to decide what the user sees, the navigation paths, and how to interpret user input. In a WPF, UWP or Xamarin application this is the code in a viewmodel class (when using the MVVM design pattern), whereas in a Windows Forms application this is the code behind the form.

It's the code behind the form in a Web Forms application, too, but here it can also include code that resides in server-side controls; *logically*, that's part of the same layer. In an ASP.NET MVC application the View and Controller are both part of the Interface Control layer. The HTML, Javascript and other content produced by the View form the Interface. Finally, the Business layer is the model.

In many applications, the Interface Control code is very complex. For a start, it must respond to the user's requests in a nonlinear fashion. (It is difficult to control how users might click controls or enter or leave the forms or pages.) The Interface Control code must also interact with logic in the

Business layer to validate user input, to perform any processing that's required, or to do any other business-related action.

The goal is to write Interface Control code that accepts user input and then provides it to the Business layer, where it can be validated, processed, or otherwise manipulated. The Interface Control code must then respond to the user by displaying the results of its interaction with the Business layer. Was the user's data valid? If not, what was wrong with it? And so forth.

In .NET, the Interface Control code is often event-driven. In most cases, WPF, UWP, Xamarin and Windows Forms code is all about responding to events as the user types and clicks the form, and ASP.NET code is all about responding to events as the browser round-trips the user's actions back to the web server. Although these UI technologies make heavy use of objects, the code that is typically written into the UI isn't object-oriented as much as procedural and event-based.

That said, there's great value in creating frameworks and reusable components that will support a particular type of UI. When creating an interface, developers can make use of numerous object-oriented techniques to simplify the creation, display and management of the forms or pages. For example, by using the MVVM design pattern in XAML-based platforms, and using ASP.NET MVC for web development, the event-driven nature of the interface technologies is abstracted. Although the interface may still be responding to user events or postbacks, your code is contained in a viewmodel or controller class. As such, it is only indirectly event-based.

Because there's such a wide variety of UI styles and approaches, I can't pretend like this book series will cover all options. However, later books in the series will cover the use of MVVM in WPF, UWP and Xamarin, as well as ASP.NET Core. I'll also cover the creation of the Business logic and Data Access layers, which are required for any type of interface.

Business Logic

Business logic includes all business rules, data validation, manipulation, processing, and authorization for the application. One definition from Microsoft is as follows: "The combination of validation edits, login verifications, database lookups, policies, and algorithmic transformations that constitute an enterprise's way of doing business."[1]

> Again, although you may implement validation logic to run in a browser or other external client, that code can't be trusted. You must view the logic that runs under your control in the Business layer as being the only *real* validation logic.

The business logic *must* reside in a separate layer from the presentation code. Although you may choose to duplicate some of this logic in your presentation code to provide a richer user experience, the Business layer must implement all the business logic, because it is the only point of central control and maintainability.

I believe that this particular separation between the responsibilities of the Business layer and Presentation layers is absolutely critical if you want to gain the benefits of increased maintainability and reusability. This is because any business logic that creeps into the UI layer will reside within a *specific* UI and will not be available to any other interfaces that might be created later.

Any business logic written into (for example) a Xamarin UI is useless to a web or service interface and must therefore be written into those as well. This instantly leads to duplicated code, which is a

maintenance nightmare. Separation of these two layers can be done through techniques such as clearly defined procedural models, or object-oriented design and programming. In this book, I'll show how to use object-oriented concepts to help separate the business logic from the UI.

This same focus on *separation of concerns* can be found in design patterns such as Model-View-Controller (MVC), Model-View-Presenter (MVP), and Model-View-ViewModel (MVVM). In all these patterns the common theme is that business logic is encapsulated into the *model* and is therefore separate from any presentation or UI layer elements. This model is the Business layer I am talking about here.

It is important to recognize that a typical application will use business logic in a couple of different ways. Most applications have some user interaction, such as forms in which the user views or enters data into the system. Most applications also have some very non-interactive processes, such as posting invoices, relieving inventory, or calculating insurance rates.

Ideally, the Business layer will be used in a rich and interactive way when the user is directly entering data into the application. For instance, when a user is entering a sales order, he or she expects that the validation of data, the calculation of tax, and the subtotaling of the order will happen literally as they type. This implies that the Business layer can be physically deployed on the client device or on the web server to provide the high levels of interactivity users desire.

To support non-interactive processes, on the other hand, the Business layer often needs to be deployed onto an application server, or as close to the database server as possible. For instance, the calculation of an insurance rate can involve extensive database lookups along with quite a bit of complex business processing. This is the kind of thing that should occur behind the scenes on a server, not on a user's desktop.

Fortunately, it is possible to deploy a Logical layer on multiple physical tiers. Doing this does require some up-front planning and technical design, as you'll see in Chapter 3. The result is a single Business layer that is potentially deployed on both the client device (or web server) and on the application server. This allows the application to provide high levels of interactivity when the user is directly working with the application, and efficient back-end processing for non-interactive processes.

Data Access

Data access code interacts with the Data Management layer to retrieve, insert, update, and remove information. The Data Access layer doesn't manage or store the data; it merely provides an interface between the business logic and the database.

Data access gets its own logical layer for much the same reason that the Interface is split from the Interface Control. In some cases, data access will occur on a machine that's physically separate from the one on which the presentation and/or business logic is running. In other cases, data access code will run on the same machine as the business logic (or even the presentation) in order to improve performance or fault tolerance.

It may sound odd to say that putting the Data Access layer on the same machine as the business logic can *increase* fault tolerance, but consider the case of web farms or PaaS cloud services, in which each web server instance is identical to all the others. Putting the data access code on the web servers provides automatic redundancy of the Data Access layer along with the business logic and UI layers.

Adding an extra physical tier just to do the data access makes fault tolerance harder to implement, because it increases the number of tiers in which redundancy needs to be implemented. As a side effect, adding more physical tiers also reduces performance for a single user, so it's not something that should be done lightly.

Logically defining data access as a separate layer enforces a separation between the business logic and any interaction with a database (or any other data source). This separation provides the flexibility to choose later whether to run the data access code on the same machine as the business logic, or on a separate machine. It also makes it much easier to change data sources without affecting the application. This is important because it enables switching from one database vendor to another at some point.

This separation is useful for another reason: Microsoft has a habit of changing data access technologies every three years or so, meaning that it is necessary to rewrite the data access code to keep up (remember DAO, RDO, ADO 1.0, ADO 2.0, ADO.NET, DataSet/TableAdapter, LINQ to SQL, and the various Entity Framework versions). By isolating the data access code into a specific layer, the impact of these changes is limited to a smaller part of the application.

Data access mechanisms are typically implemented as a set of services, with each service being a procedure that's called by the business logic to retrieve, insert, update, or delete data. Although these services are often constructed using objects, it's important to recognize that the designs for an effective Data Access layer are quite procedural in nature. Attempts to force more object-oriented designs for relational database access often result in increased complexity or decreased performance. I think the best approach is to implement the data access as a set of methods but encapsulate those methods within objects to keep them logically organized.

Sometimes the Data Access layer can be as simple as a series of methods that use ADO.NET directly to retrieve or store data. In other circumstances, the Data Access layer is more complex, providing a more abstract or even metadata-driven way to get at data. In these cases, the Data Access layer can contain a lot of complex code to provide this more abstract data access scheme. The CSLA .NET framework doesn't restrict how you implement your Data Access layer. The examples in the book series demonstrate how to use ADO.NET (connection, command and datareader objects), and Entity Framework, but you could also use some other metadata-driven Data Access layer if you prefer.

Another common role for the Data Access layer is to provide mapping between the object-oriented business logic and the relational data in a data store. A good object-oriented model is almost never the same as a good relational database model. Objects often contain data from multiple tables, or even from multiple databases. Conversely, multiple objects in the model can represent a single table. The process of taking the data from the tables in a relational model and

getting it into the object-oriented model is called *object-relational mapping* (ORM), and I'll have more to say on the subject in Chapter 3.

Data Storage and Management

Finally, there's the Data Storage and Management layer. Database servers such as SQL Server and Oracle often handle these tasks, but increasingly, other applications may provide this functionality as well, via technologies such as XML or JSON services. NoSQL databases, document databases, and blob storage are also becoming quite common.

What's key about the Data Management layer is that it handles the physical creation, retrieval, update, and deletion of data. This is different from the Data Access layer, which *requests* the creation, retrieval, update, and deletion of data. The Data Management layer *implements* these operations within the context of a database or a set of files, and so on.

The business logic (via the Data Access layer) invokes the Data Management layer, but the layer often includes additional logic to validate the data and its relationship to other data. Sometimes, this is true relational data modeling from a database. Other times, it's the application of business logic from an external application. What this means is that a typical Data Management layer will include business logic that is also implemented in the Business layer. This time, the duplication is unavoidable because relational databases are designed to enforce relational integrity; and that's just another form of business logic.

In summary, whether you're using stored procedures in SQL Server, or service calls to another application; data storage and management is typically handled by creating a set of services or procedures that can be called as needed. Like the Data Access layer, it's important to recognize that the designs for data storage and management are typically very procedural.

Table 2 summarizes the five layers and their roles.

Layer	Roles
Interface	Renders display and collects user input.
Interface control	Acts as an intermediary between the user and the business logic, taking user input and providing it to the business logic, then returning results to the user.
Business logic	Provides all business rules, validation, manipulation, processing, and security for the application.
Data access	Acts as an intermediary between the business logic and data management. Also encapsulates and contains all knowledge of data access technologies (such as LINQ to SQL), databases, and data structures.
Data storage and management	Physically creates, retrieves, updates, and deletes data in a persistent data store.

Table 2. The Five Logical Layers and the Roles They Provide

Everything I've talked about to this point is part of a *logical* architecture. Now it's time to move on and see how it can be applied in various *physical* configurations.

Applying the Logical Architecture

Given this 5-layer logical architecture, it is theoretically possible to configure it into one, two, three, four, or five physical tiers in order to gain performance, scalability, security, or fault tolerance to various degrees, and in various combinations. It is also possible to apply this architecture into service-oriented scenarios.

> In this discussion, I assume that there is total flexibility to configure which logical layer runs where. In some cases, there are technical issues that prevent the physical separation of some layers. As I noted earlier, you need to strike a balance between flexibility and capability.

The CSLA .NET framework supports a subset of the theoretical combinations. This set of deployment models meets common requirements around performance, security and scalability:

- Optimal Performance Smart Client
- High Scalability Smart Client or Mobile App
- Optimal Performance Web Client
- High Security Web Client

- Service-oriented Edge Application
- Microservice or Service-oriented Application

These are common and important deployment models that are encountered on a day-to-day basis.

Optimal Performance Smart Client

When so much focus is placed on distributed systems, it's easy to forget the value of a single-tier solution. Point of sale, sales force automation, and many other types of application often run in stand-alone environments. Even so, the benefits of the logical n-layer architecture are still desirable in terms of maintainability and code reuse.

It probably goes without saying that everything can be installed on a single client device. An optimal performance smart client is usually implemented using Xamarin, WPF, UWP, or Windows Forms for the presentation and UI, with the business logic and data access code running in the same process and talking to Microsoft SQL Server Express, SQLLite, or other locally installed database engine.

The fact that the system is deployed on a single physical tier doesn't compromise the logical architecture and separation, as shown in Figure 2.

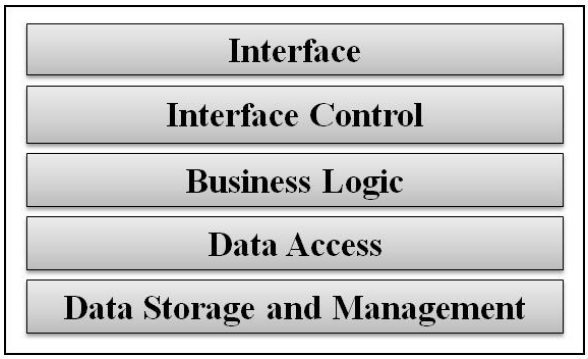

Figure 2. The five logical layers running on a single device

I think it's important to remember that n-layer systems can run on a single device in order to support the wide range of applications that require stand-alone devices.

It's also worth pointing out that this is basically the same as 2-tier, "fat-client" physical architecture. The only difference in that case is that the Data Storage and Management tier would be running on a central database server, such as SQL Server or Oracle, as shown in Figure 3.

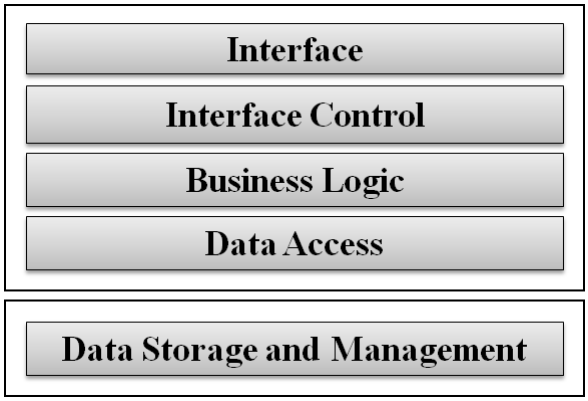

Figure 3. The five logical layers with a separate database server

Other than the location of the data storage, this is identical to the single-tier configuration, and typically the switch from single-tier to 2-tier revolves around little more than changing the database connection string.

High-Scalability Smart Client or Mobile App

Single-tier configurations are good for stand-alone environments, but they don't scale well. To support multiple users, it is common to use 2-tier configurations. I've seen 2-tier configurations support hundreds of concurrent users against SQL Server with acceptable performance.

Going further, it is possible to trade performance to gain scalability by moving the Data Access layer to a separate machine. Single or 2-tier configurations give the best performance, but they don't scale as well as a 3-tier configuration. A good rule of thumb is that if you have more than 100 concurrent users, you can benefit by making use of a separate server to handle the Data Access layer.

It is also the case that most apps written using Xamarin, WebAssembly, or UWP will need access to a shared remote database server. Because those platforms are unable to directly interact with a database server, they must have an application server as part of the overall architecture.

Other than to support mobile apps, perhaps the most common reason for moving the Data Access layer to an application server is security. Because the Data Access layer contains the code that directly interacts with the database, the machine on which it runs must have credentials to access the database server. Rather than having those credentials on the client device, they can be moved to an application server. This way, the user's computer won't have the credentials to interact directly with the database server, thereby increasing security.

It is also possible to put the Business layer on the application server. This is useful for non-interactive processes such as batch updates or data-intensive business algorithms. At the same time, most applications allow for user interaction, and so there is adefinite need to have the Business layer running on the client device to provide high levels of interactivity for the user.

As discussed earlier in the chapter, it is possible to deploy the same logical layer onto multiple physical tiers. Using this idea, the Data Access layer can be put on an application server, and the Business layer on *both* the client device and the application server, as shown in Figure 4.

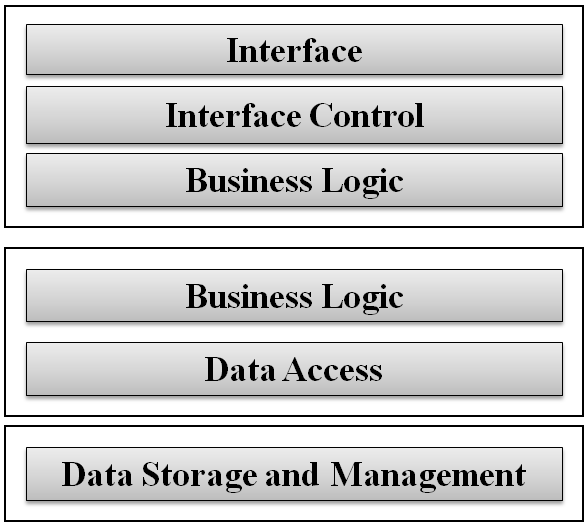

Figure 4. The five logical layers with separate application and database servers

Putting the Data Access layer on the application server centralizes all access to the database on a single machine. In .NET, if the connections to the database for all users are made using the same user ID and password, you'll get the benefits of *connection pooling* for all your users. What this means immediately is that there will be far fewer connections to the database than there would be if each client device connected directly. The actual reduction depends on the specific application, but often it means supporting 150 to 200 concurrent users with only two or three database connections!

Of course, all user requests now go across an extra network hop, thereby causing increased latency (and therefore decreased performance). This performance cost translates into a huge scalability gain, however, because this architecture can handle many more concurrent users than a 2-tier physical configuration.

With the Business layer deployed on both the client and server, the application can fully exploit the strengths of both machines. Validation and a lot of other business processing can run on the client device to provide a rich and highly interactive experience for the user, while non-interactive processes can efficiently run on the application server.

If well designed, such an architecture can support *thousands* of concurrent users with adequate performance.

Optimal Performance Web Client

As with a smart client or mobile app, the best performance is received from a web-based application by minimizing the number of physical tiers. Notably, the trade-off in a web scenario is different: in this case, it is possible to improve performance and scalability at the same time, but at the cost of security, as I will demonstrate.

To get optimal performance in a web application, it is desirable to run most of the code in a single process on a single machine, as shown in Figure 5.

The Interface layer must be physically separate because it's running in a browser, but the Interface control, Business Logic, and Data Access layers can all run on the same machine, in the

same process. In some cases, you might even put the Data Management layer on the same physical machine, though this is only suitable for smaller applications.

This minimizes network and communication overhead and optimizes performance. Figure 6 shows how it is possible to get good scalability, because the web server can be part of a web farm in which all the web servers are running the same code.

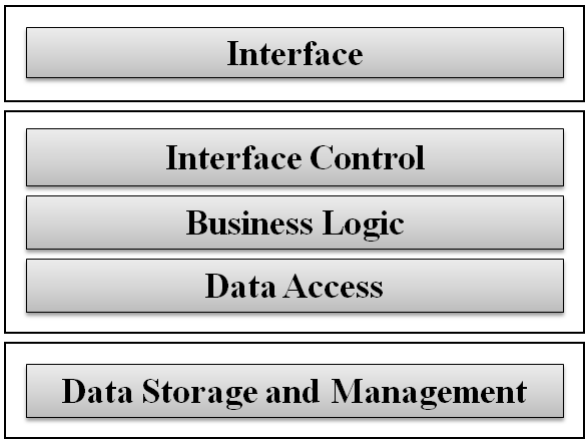

Figure 5. The five logical layers as used for web applications

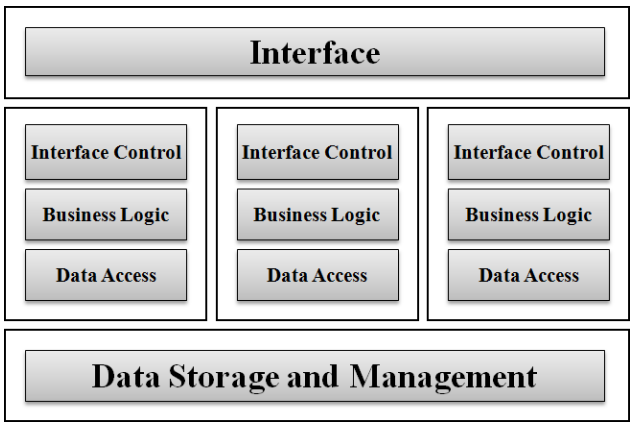

Figure 6. The five logical layers deployed on a load-balanced web farm

This setup provides good database-connection pooling because each web server will be (potentially) servicing hundreds of concurrent users, and all database connections on a web server are pooled.

Unless the database server is getting overwhelmed with connections from the web servers in the web farm, a separate application server will rarely provide gains in scalability. If a separate application server is needed, there will be a reduction in performance because of the additional physical tier. (Hopefully, there will be a gain in scalability, because the application server can consolidate database connections across all the web servers.) It is important to consider fault tolerance in this case, because redundant application servers may be needed in order to avoid a point of failure.

Another reason for implementing an application server is to increase security, and that's the topic of the next section.

High-Security Web Client

As discussed in the earlier section on security, there will be many projects in which it's dictated that a web server can never talk directly to a database. The web server must run in a "demilitarized zone" (DMZ), sandwiched between the external firewall and a second internal firewall. The web server must communicate with another server through the internal firewall in order to interact with the database or any other internal systems.

As with the 3-tier smart client scenario, there is tremendous benefit to also having the Business layer deployed on both the web server and the application server. Such a deployment allows the web UI code to interact closely with the business logic when appropriate, whereas non-interactive processes can run on the application server.

This is illustrated in Figure 7, in which the dashed lines represent the firewalls.

Splitting out the Data Access layer and running it on a separate application server increases the security of the application. This comes at the cost of performance—as discussed, this configuration will typically cause a performance degradation of around 50 percent. Scalability, on the other hand, is fine: like the first web configuration, it can be achieved by implementing a web farm in which each web server runs the same UI and business logic code, as shown in Figure 8.

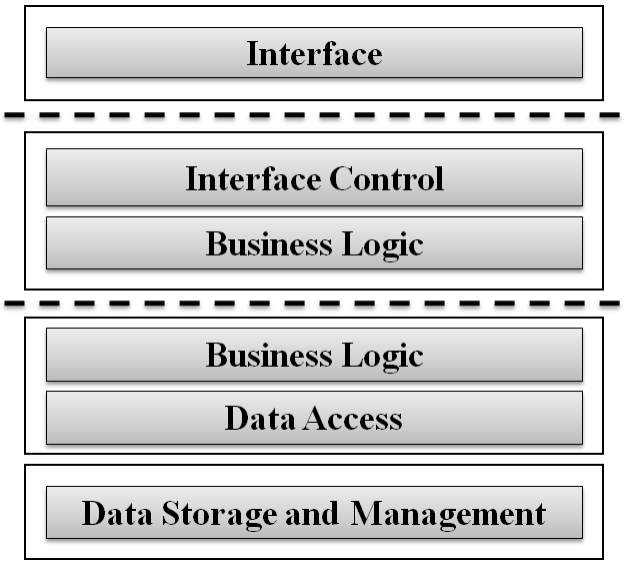

Figure 7. The five logical layers deployed in a secure web configuration

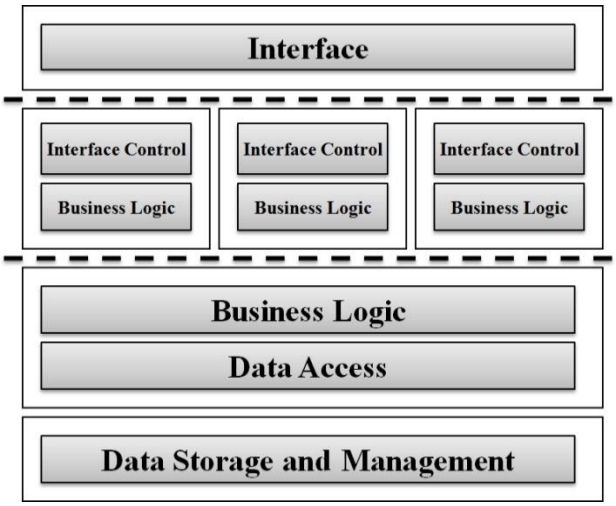

Figure 8. The five logical layers in a secured environment with a web farm

Service-oriented Edge Application

The deployment models I've discussed so far have been n-tier models, assuming that the 5 layers of the logical architecture are being deployed across one to four physical machines or tiers. Service-oriented architecture is a fundamentally different concept, because it is about communication *between applications* instead of between layers or tiers of one application.

This means the 5-layer architectural model should be applied to each application in a service-oriented system. The most visible parts of any service-oriented system are the applications that sit on the "edge" of the system to enable user interaction. These are sometimes called *edge applications*, and they exist primarily to interact with end users.

Figure 9 shows an edge application interacting through a service bus with numerous services. The focus of this diagram is not on the services and how they are implemented, but rather on the application of the 5-layer architecture into the edge application.

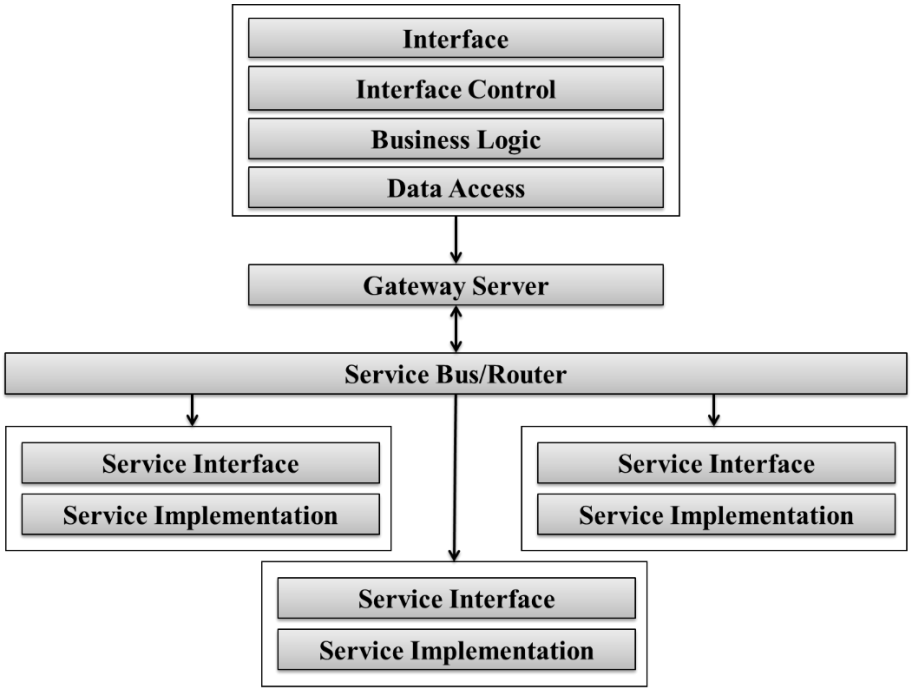

Figure 9. Service-oriented Edge Application

The first thing you should notice is that there's no explicit Data Storage and Management layer in this diagram. That is because the services take the place of that layer, providing and consuming data from the edge application.

This has a direct impact on the implementation of the Data Access layer, which interacts with services instead of calling stored procedures or otherwise directly interacting with a database. But from an architectural perspective, the Data Access layer is still responsible for getting and saving data by interacting with the logical Data Storage and Management layer: the services.

In most cases, an edge application will be created using a smart client technology such as Xamarin, WebAssembly, UWP, or WPF. The edge application could also be constructed using HTML and JavaScript, in which case the logical layers will be implemented using HTML and JavaScript instead of any .NET technology.

Client apps do not normally directly interact with the queued messaging service running inside a physical data center. Instead they interact with a *gateway server* that provides a web-friendly public API, typically via http/https. That gateway server acts as a proxy between the outside world, and the world inside the service-oriented system itself. This allows the services to use advanced queue messaging technologies like a service bus, Azure Service Bus, RabbitMQ, or ActiveMQ without adding complexity to the client apps.

Microservice or Service-oriented Application

Although microservice (service-oriented) systems typically have edge applications, they are primarily composed of services. Because a microservice based system is all about applications interacting with each other through messages, it is a good idea to think about each service as being a separate application, so when I use the term *microservice* I'm really talking about a specific application that exposes a service interface (such as accepting and returning JSON).

> I also frequently use the term *domain service* when referring to an individual service, because "micro" implies a specific size, but really your services should be designed to solve a discrete problem within your business domain. Sometimes that means they aren't so "micro".

Like the other application models discussed in this chapter, it is both possible and desirable, to apply the 5-layer architecture to a microservice. When doing this, it is important to remember that the Interface layer consists of your service contract, which means it is usually XML or JSON.

The focus of the diagram in Figure 10 is on the service implementation. Through the service bus, the service can be invoked from edge applications or other services, but this particular service is a standalone application.

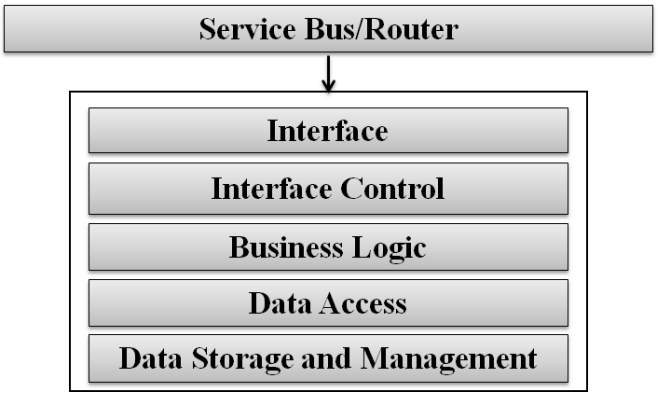

Figure 10. Microservice

If you are creating services and edge applications, you should consider applying the 5-layer architecture to each of these applications. In every case, having a logically layered model helps promote separation of concerns and provides all the benefits of a layered architecture as discussed earlier in this book.

This extends to flexibility in terms of physical deployment. For example, a microservice might be deployed into n-tier models as shown in Figure 11.

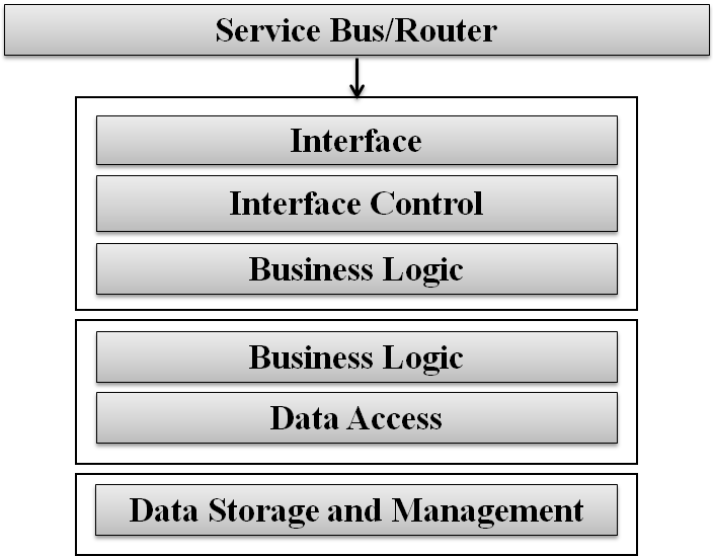

Figure 11. Microservice with separate application and database servers.

You might choose to combine n-tier and service-oriented architectures in this manner to improve the scalability or security of a service, in the same way you'd use n-tier deployments to achieve those goals with a smart client or web application.

The Way Ahead

Through this book series, I will create sample applications demonstrating how to use a Business layer implemented using CSLA .NET to leverage this 5-layer architecture. These applications will demonstrate how to create interfaces using .NET based smart client, mobile, web, and service-oriented technologies. This will give you the opportunity to see firsthand how the framework supports the deployment models I discussed.

Due to the way the CSLA .NET framework is implemented, switching between the deployment models discussed here requires only configuration file changes on the client and configuration of your servers. The result is that you can easily adapt your application to any of the physical configurations without having to change your code.

Managing Business Logic

At this point, you should have a good understanding of logical and physical architectures, and how a 5-layer logical architecture can be configured into various n-tier physical architectures. In one way or another, all of these layers will use or interact with the application's data. That's obviously the case for the Data Management and Data Access layers, but the Business layer must validate, calculate, and manipulate data; the Interface Control layer transfers data between the Business Logic and Interface layers (often performing formatting or using the data to make navigational choices); and the Interface layer displays data to the user and collects new data as it's entered.

In an ideal world, all of the business logic would exist in the Business layer, but in reality, this is virtually impossible to achieve. In a web-based UI, validation logic is often included in the Interface layer, so that the user gets a more interactive experience in the browser. Unfortunately, any validation that's done in the web browser is unreliable, because it's too easy for a malicious user to

bypass that validation. Because of this, any validation done in the browser must be rechecked in the Business layer as well.

Similarly, most databases enforce referential integrity, and often some other rules, too. Furthermore, the Data Access layer will very often include business logic to decide when and how data should be stored or retrieved from databases and other data sources. In almost any application, to a greater or a lesser extent, business logic gets scattered across all the layers.

There's one key truth here: for each piece of application data, there's a fixed set of business logic associated with that data. If the application is to function properly, the business logic must be applied to that data at least once. Why "at least"? In most applications, some of the business logic is applied more than once. For example, a validation rule applied in the Interface layer can be reapplied in the Interface Control layer or Business layer before data is sent to the database for storage. In some cases, the database will include code to recheck the value as well.

Now, I'd like to look at some of the more common options. I'll start with three popular (but flawed) approaches. Then I'll discuss a compromise solution that's enabled using mobile objects; such as the ones supported by the CSLA .NET framework.

Potential Business Logic Locations

Figure 12 illustrates common locations for validation and manipulation business logic in a typical application. Most applications have the same logic in at least a couple of these locations.

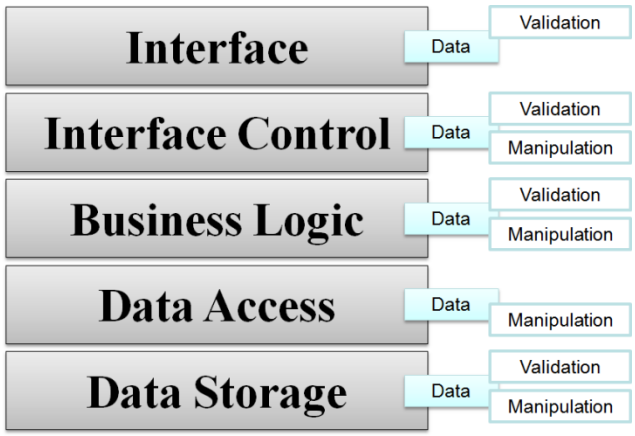

Figure 12. Common locations for business logic in applications

Business logic is put in a web interface layer to give the user a more interactive experience—and put into a Windows UI for the same reason. The business logic is rechecked in the web Interface Control layer (on the web server) because the browser isn't trustworthy. And database administrators put the logic into the database (via stored procedures and other database constructs) because they don't trust any application developers!

The result of all this validation is a lot of duplicated code, all of which must be debugged, maintained, and somehow kept in sync as the business needs (and therefore logic) change over time. In the real world, the logic is almost never *really* kept in sync, and so developers must constantly debug and maintain the code in a near-futile effort to make all these redundant bits of logic agree with each other.

One solution is to force all the logic into a single layer, thereby making the other layers as "dumb" as possible. There are various approaches to this, although (as you'll see) none of them provide an optimal solution.

Another solution is to dynamically generate the validation logic for the Interface or Interface Control layer based on metadata provided from the Business layer. This requires more work in the UI layer but can increase maintainability overall.

Business Logic in the Data Management Tier

The classic approach is to put all logic into the database as the single, central repository. The presentation and UI then allow the user to enter absolutely anything (because any validation would be redundant), and the Business layer now resides inside the database. The Data Access layer does nothing but move the data into and out of the database, as shown in Figure 13.

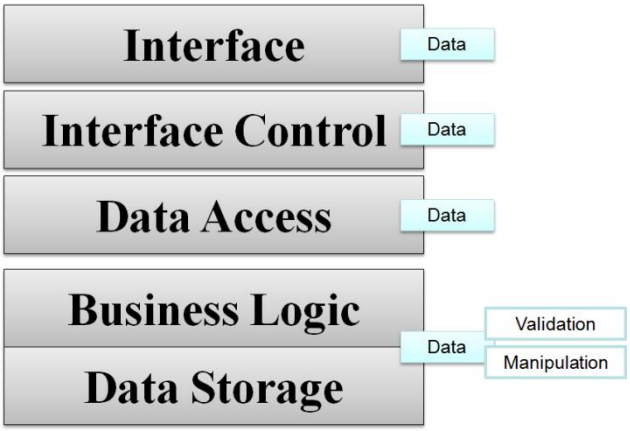

Figure 13. Validation and business logic in the Data Management tier

The advantage of this approach is that the logic is centralized, but the drawbacks are plentiful. For starters, the user experience is totally non-interactive. Users can't get any results, or even confirmation that their data is valid, without round-tripping the data to the database for processing. The database server becomes a performance bottleneck, because it's the only thing doing any substantial work. Unfortunately, the hardest physical tier to scale up for more users is the database server, because it is difficult to use load balancing techniques on it. The only real alternative is to buy a bigger and bigger server machine.

Business Logic in the UI Tier

Another common approach is to put all of the business logic into the UI. The data is validated and manipulated in the UI, and the Data Storage layer just stores the data. This approach, as shown in Figure 14, is common in both Windows and web environments, and has the advantage that the business logic is centralized into a single tier (and of course, one can write the business logic in a language such as C#).

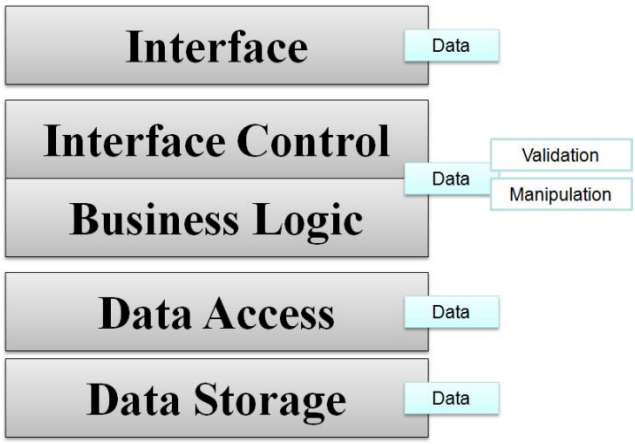

Figure 14. Business logic deployed with only the UI

Unfortunately, in practice, the business logic ends up being scattered throughout the UI and intermixed with the UI code itself, thereby decreasing readability and making maintenance more difficult. Even more importantly, business logic in one form or page isn't reusable when subsequent pages or forms are created that use the same data. Furthermore, in a web environment, this architecture also leads to a totally non-interactive user experience, because no validation can occur in the browser. The user must transmit his or her data to the web server for any validation or manipulation to take place.

> ASP.NET validation controls at least allow for basic data validation in the UI, with that validation automatically extended to the browser by the ASP.NET technology itself. Though not a total solution, this is a powerful feature that does help.
>
> Similarly, ASP.NET MVC and Razor Pages can automatically project script into the web page based on DataAnnotations attributes. This is a more complete solution because CSLA .NET also supports these attributes as business rules, so there's a real reduction in effort on the part of the developer.

Business Logic in the Middle (Business and Data Access) Tier

Still another option is the classic UNIX client-server approach, whereby the Business Logic and Data Access layers are merged, keeping the presentation, UI, and Data Storage tiers as "dumb" as possible (see Figure 15).

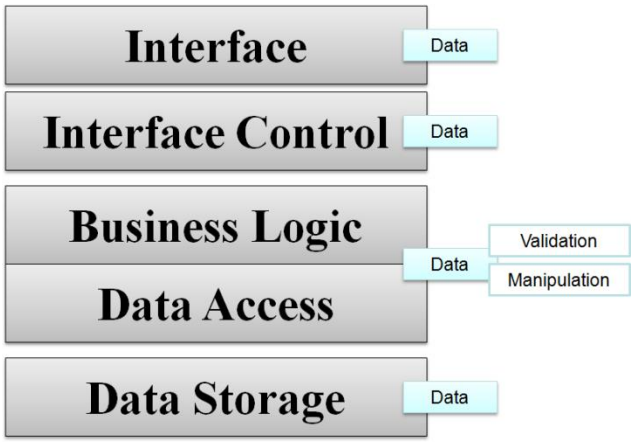

Figure 15. Business logic deployed on only the application server

Unfortunately, once again, this approach falls afoul of the non-interactive user experience problem: the data must round-trip to the Business Logic/Data Access tier for any validation or manipulation. This is especially problematic if the Business Logic/Data Access tier is running on a separate application server, because then you're faced with network latency and contention issues, too. Also, the central application server can become a performance bottleneck, because it's the only machine doing any work for all the users of the application.

Sharing Business Logic Across Tiers

I wish this book included the secret that allows you to write all your logic in one central location, thereby avoiding all of these awkward issues. Unfortunately, that's not possible with today's technology: putting the business logic only on the client, application server, or database server is problematic, for all the reasons given earlier. But something needs to be done about it, so what's left?

What's left is the possibility of centralizing the business logic in a Business layer that's deployed on the client (or web server), so that it's accessible to the Interface Control layer; and in a Business layer that's deployed on the application server, so that it's able to interact efficiently with the Data Access layer. The result is the best of both worlds: a rich and interactive user experience and efficient high-performance back-end processing when interacting with the database (or other data source).

In the simple cases in which there is no application server, the Business layer is deployed only once: on the client device or web server, as shown in Figure 16.

Ideally, this business logic will run on the same device as the Interface control code when interacting with the user, but on the same machine as the data access code when interacting with the database. (As discussed earlier, all of this could be on one machine or several different machines, depending on your physical architecture.) It must provide a friendly interface that the UI developer can use to invoke any validation and manipulation logic, and it must also work efficiently with the Data Access tier to get data in and out of storage.

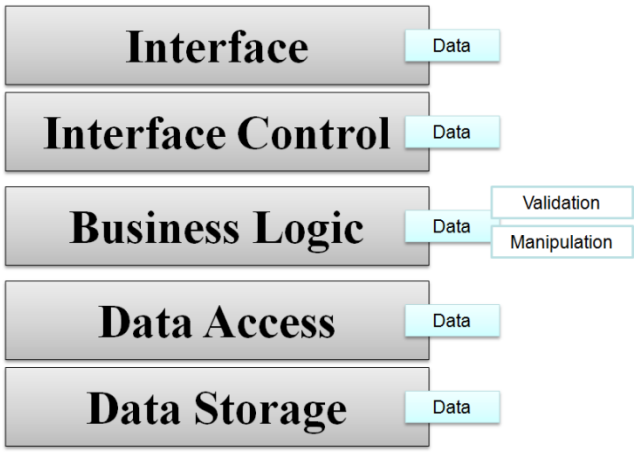

Figure 16. Business logic centralized in the Business layer

The tools for addressing this seemingly intractable set of requirements are *mobile business objects* that encapsulate the application's data along with its related business logic. It turns out that a properly constructed business object can move around the network from machine to machine with almost no effort on your part. The .NET Framework and CSLA .NET handle the details, and you can focus on the business logic and data.

By properly designing and implementing mobile business objects, you allow the .NET Framework to pass your objects across the network *by value*, thereby automatically copying them from one machine to another. This means that with little extra code, you can have your business logic and business data move to the device where the Interface Control layer is running, and then shift to the machine where the Data Access layer is running when data access is required.

At the same time, if you're running the Interface Control and Data Access layers on the same device, then the .NET Framework doesn't move or copy your business objects. They're used directly by both layers on the same device with little performance cost or extra overhead. You don't have to do anything to make this happen, either—CSLA .NET automatically detects that the object doesn't need to be copied or moved, and thus takes no extra action.

The Business layer becomes portable, flexible, and mobile, and adapts to the physical environment in which you deploy the application. Due to this, you're able to support a variety of physical n-tier architectures with one code base, whereby your business objects contain no extra code to support the various possible deployment scenarios. What little code you need to implement to support the movement of your objects from machine to machine will be encapsulated in a framework, leaving the business developer to focus purely on the development of business logic.

Business Domain Objects

Having decided to use business objects and take advantage of CSLA .NET's ability to move objects around the network automatically, it's now time to discuss business objects in more detail. I will discuss exactly what they are and how they can help you to centralize the business logic pertaining to your data.

The primary goal when designing any kind of software object is to create an abstract representation of some entity or concept. In ADO.NET, for example, a `DataTable` object represents

a tabular set of data. `DataTable`s provide an abstract and consistent mechanism by which you can work with *any* tabular data. Likewise, a Windows Forms `TextBox` control is an object that represents the concept of displaying and entering data. From the application's perspective, there is no need to have any understanding of how the control is rendered on the screen, or how the user interacts with it. It's simply an object that includes a `Text` property and a handful of interesting events.

Key to successful object design is the concept of *encapsulation*. This means that an object is a black box: it contains logic and data, but the user of the object doesn't know *what* data or *how* the logic works. All they can do is interact with the object.

> Properly designed objects encapsulate both behavior or logic and the data required by that logic.

If objects are abstract representations of entities or concepts that encapsulate both data and its related logic, what then are *business objects* (sometimes also known as *domain objects* or *business domain objects*)?

> Business objects are different from regular objects only in terms of what they represent.

Object-oriented applications are created to address problems of one sort or another. In the course of doing so, a variety of different objects are often used. Some of these objects will have no direct connection with the problem at hand (`DataTable` and `TextBox` objects, for example, are just abstract representations of computer concepts). There will be others that are closely related to the area or *domain* in which you're working. If the objects are related to the business for which you're developing an application, then they're business objects.

For instance, if you're creating an order entry system, your business domain will include things such as customers, orders, and products. Each of these will likely become business objects within your order entry application—the `OrderEdit` object, for example, will provide an abstract representation of the order being created or edited by a customer.

> Business objects provide an abstract representation of entities or concepts that are part of the business or problem domain.

Business Domain Objects as Smart Data

I've already discussed the drawbacks of putting business logic into the UI tier, but I haven't thoroughly discussed the drawback of keeping the data in a generic representation such as an array, collection, JSON, or XML document. The data in an array or XML document is unintelligent, unprotected, and generally unsafe. There's nothing to prevent anyone from putting invalid data into any of these containers, and there's nothing to ensure that the business logic behind one form in the application will interact with the data in the same way as the business logic behind another form.

An XML document with an XSD (XML Schema Definition) might ensure that text cannot be entered where a number is required, or that a number cannot be entered where a date is required. At best, it might enforce some basic relational-integrity rules. There's no way to ensure that the

values match other criteria, or that calculations or other processing is done properly against the data, without involving other objects. The data in an array, or JSON document isn't self-aware; it's not able to apply business rules or handle business manipulation or processing of the data.

The data in a business object is what I like to call "smart data." The object not only contains the data, but also includes all the business logic that goes along with that data. Any attempt to work with the data must go through this business logic. In this arrangement, there is much greater assurance that business rules, manipulation, calculations, and other processing will be executed consistently everywhere in the application. In a sense, the data has become self-aware, and can protect itself against incorrect usage.

In the end, an object doesn't care whether it's used by a mobile UI, a batch-processing routine, or a web service. The code using the object can do as it pleases; the object itself will ensure that all business rules are always obeyed.

Contrast this with a collection or XML document, in which the business logic doesn't reside in the data container, but somewhere else—often in the code-behind in a Xamarin page or in a web controller. If multiple forms or pages use this data container, there is no assurance that the business logic is applied consistently. Even if you adopt a standard that says that UI developers must invoke methods from a centralized class to interact with the data, there's nothing preventing them from interacting with the data directly. This may happen accidentally, or because it was easier or faster to modify the data than to go through some centralized routine.

> With consistent use of business objects, there's no way to bypass the business logic. The only way to the data is through the object, and the object always enforces the rules.

So, a business object that represents an invoice will include not only the data pertaining to the invoice, but also the logic to calculate taxes and amounts due. The object should understand how to post itself to a ledger, and how to perform any other accounting tasks that are required. Rather than passing raw invoice data around, and having the business logic scattered throughout the application, it is possible to pass an `InvoiceEdit` object around. The entire application can share not only the data, but also its associated logic. Smart data through objects can dramatically increase the ability to reuse code and can decrease software maintenance costs.

Anatomy of a Business Object

Putting all of these pieces together, you get an object that has an interface (a set of properties and methods), some implementation code (the business logic behind those properties and methods), and state (the data). This is illustrated in Figure 17.

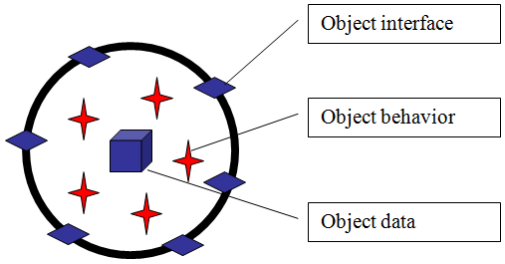

Figure 17. A business object composed of state, implementation, and interface

The hiding of the data and the implementation code behind the interface are keys to the successful creation of a business object. If the users of an object are allowed to "see inside" it, they will be tempted to cheat, and to interact with the logic or data in unpredictable ways. This danger is the reason why it will be important to take care when using the `public` keyword as you build your classes.

Any property, method, event, or field marked as `public` will be available to the users of objects created from the class. For example, you might create a simple class such as the following:

```
public class ProjectEdit
{
  private Guid _id = Guid.NewGuid();
  public Guid Id
  {
    get { return _id; }
  }

  private string _name = string.Empty;
  public string Name
  {
    get { return _name; }
    set
    {
      if (value == null) value = string.Empty;
      if (value.Length > 50)
        throw new Exception("Name too long");
      _name = value;
    }
  }
}
```

This defines a business object that represents a project of some sort. All that is known at the moment is that these projects have an ID value and a name. Notice, though, that the fields containing this data are `private`—you don't want the users of your object to be able to alter or access them directly. If they were `public`, the values could be changed without the object's knowledge or permission. (The `_name` field could be given a value that's longer than the maximum of 50 characters, for example.)

The properties, on the other hand, are `public`. They provide a controlled access point to the object. The `Id` property is read-only, so the users of the object can't change it. The `Name` property allows its value to be changed, but enforces a business rule by ensuring that the length of the new value doesn't exceed 50 characters.

> None of these concepts are unique to business objects—they're common to all objects and are central to object-oriented design and programming.

Mobile Objects

Unfortunately, directly applying the kind of object-oriented design and programming I've been talking about so far is often quite difficult in today's complex computing environments. Object-oriented programs are almost always designed with the assumption that all the objects in an application can interact with each other with no performance penalty. This is true when all the objects are running in the same process on the same computer, but it's not at all true when the objects might be running in different processes, or even on different computers.

Earlier in this chapter, I discussed various physical architectures in which different parts of an application might run on different machines. With a high-scalability smart client architecture, for example, there will be a client, an application server, and a data server. With a high-security web client architecture, there will be a client, a web server, an application server, and a data server. Parts of the application will run on each of these machines, interacting with each other as needed.

In these distributed architectures, you can't use a straightforward object-oriented design, because any communication between classic fine-grained objects on one machine and similar objects on another machine will incur network latency and overhead. This translates into a performance problem that can't be ignored. To overcome this problem, most distributed applications haven't used object-oriented designs. Instead, they consist of a set of procedural code running on each machine, with the data kept in a data container such as an array, or a JSON or XML document that's passed around from machine to machine.

This isn't to say that object-oriented design and programming are irrelevant in distributed environments—just that it becomes complicated. To minimize the complexity, most distributed applications are object-oriented *within a tier*, but between tiers they follow a procedural or service-based model. The result is that the application as a whole is neither object-oriented nor procedural, but a blend of both.

Perhaps the most common architecture for such applications is to have the Data Access layer retrieve the data from the database into a JSON or XML blob. The data container is then returned to the client (or the web server). The code in the forms or pages then interacts with the data object directly, as shown in Figure 18.

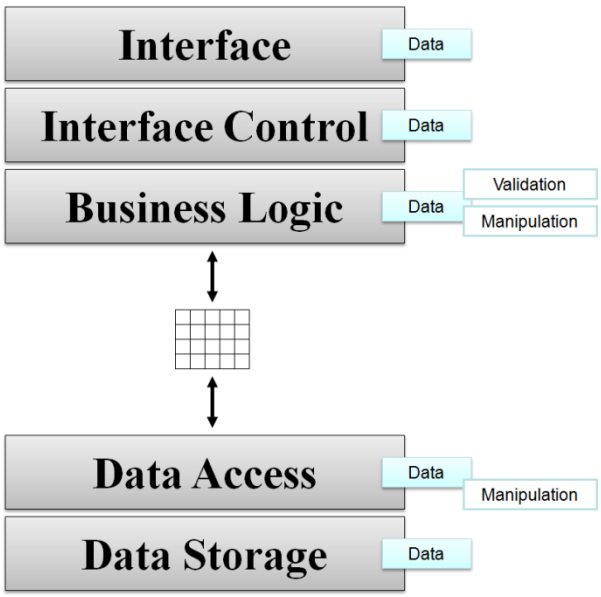

Figure 18. Passing a Data Container between the Business Logic and Data Access layers

This approach has the maintenance and code-reuse flaws that I've talked about, but the fact is that it gives pretty good performance in most cases. Also, it doesn't hurt that most programmers are familiar with the idea of writing code to manipulate a JSON or XML document or an old-

fashioned `DataSet`, so the techniques involved are well understood, thereby speeding up development.

A decision to stick with an object-oriented approach should be undertaken carefully. It's all too easy to compromise the object-oriented design by taking the data out of the objects running on one machine, sending the raw data across the network and allowing other objects to use that data outside the context of the objects and business logic. Such an approach would break the encapsulation provided by the logical Business layer.

Mobile objects are all about logically sending smart data (object graphs) from one device to another, rather than sending raw data.

The .NET Framework contains direct support for the concept of mobile objects through the `BinaryFormatter` and `NetDataContractSerializer` components. Those aren't available on all modern .NET platforms however, so the CSLA .NET framework includes its own `MobileFormatter` component that works consistently across every platform. The `MobileFormatter` is also optimized for the scenarios where CSLA .NET is used, and so is usually more efficient than other serializers.

There are also third party serializers such as Json.Net that have the capability to operate in a manner comparable to the `BinaryFormatter`.

Given the ability to send object graphs from one device to another, you can have your Data Access layer (running on an application server) create a business object and load it with data from the database. You can then logically send that business object to the client device (or web server), where the UI code can use the object (as shown in Figure 19).

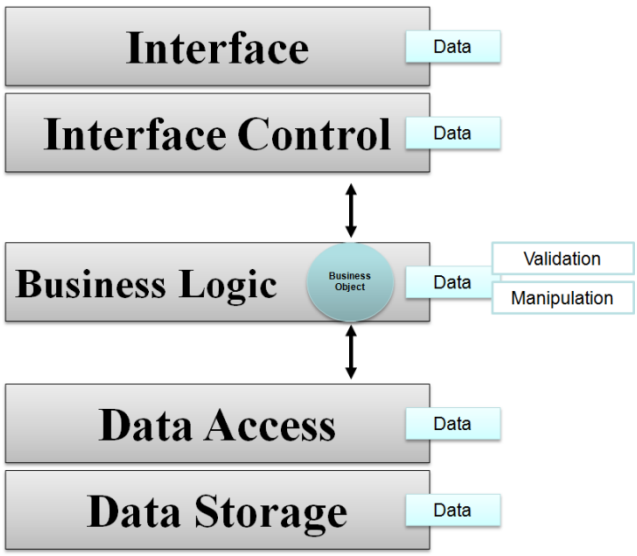

Figure 19. Using a business object to centralize business logic

In this architecture, smart data, in the form of a business object, is sent to the client rather than raw data. Then the UI code can use the same business logic as the data access code. This reduces maintenance, because you're not writing some business logic in the Data Access layer, and some other business logic in the UI layer. Instead, all the business logic is consolidated into a real, separate layer composed of business objects. These business objects will move across the network

just like the JSON or XML document did earlier, but they'll include the data *and* its related business logic—something simple data blobs can't easily offer.

> The business logic isn't transferred over the network, only the data. But the business logic is *available* everywhere the data exists in memory, so the data can never be manipulated without the business rules being enforced.

Effectively, you're sharing the Business layer between the machine running the Data Access layer and the device running the Interface Control layer. If there is support for mobile objects, this is an ideal solution: it provides code reuse, low maintenance costs, and high performance.

A New Logical Architecture

Being able to directly access the Business layer from both the Data Access layer and the Interface Control layer opens up a new way to view the logical architecture. Though the Business layer remains a separate concept, it's directly used by and tied into both the UI and Data Access layers, as shown in Figure 20.

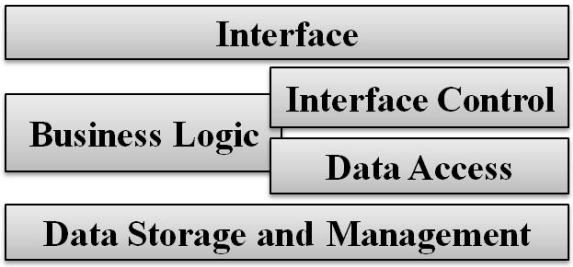

Figure 20. The Business layer tied to the UI and Data Access layers

The Interface Control layer can interact directly with the objects in the Business layer, thereby relying on them to perform all validation, manipulation, and other processing of the data. Likewise, the Data Access layer can interact with the objects as the data is retrieved or stored.

If all the layers are running on a single device (such as a smart client), then these parts will run in a single process and interact with each other with no network or cross-processing overhead. In more distributed physical configurations, the Business layer will run on both the client *and* the application server. Both models are shown in Figure 21.

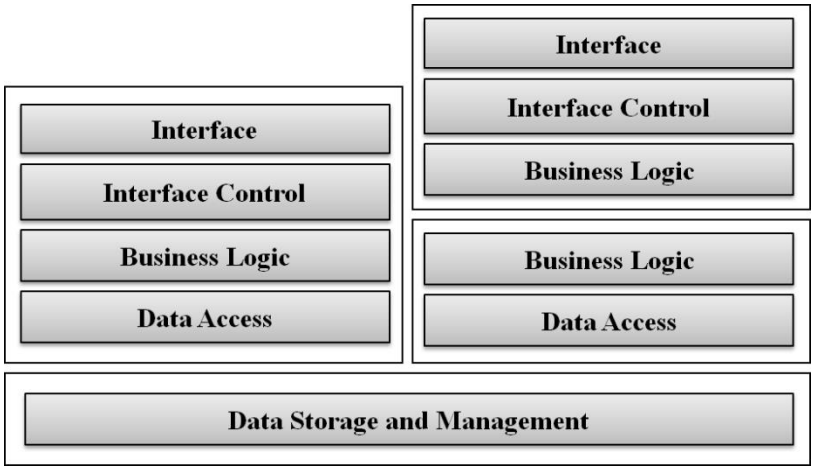

Figure 21. Business logic shared between the UI and Data Access layers

Understanding Mobile Objects

Normally, one might think of objects as being part of a single application, running on a single machine in a single process. A distributed application requires a broader perspective. Some of the objects might only run in a single process on a single machine. Others may run on one machine but may be called by code running on another machine. Still others may be mobile objects: moving from machine to machine.

Local Objects

By default, .NET objects are *local*. This means that ordinary .NET objects aren't accessible from outside the process in which they were created. Without taking extra steps in your code, it isn't possible to pass objects to another process or another machine (a procedure known as *marshaling*), either by value or by reference.

Anchored Objects

When .NET was still young there was a commonly used technology called *Remoting*. This technology allowed you to pass a reference to an object over the network to another machine. The result is that you were able to call methods on an object, even though that object was running on another machine, as shown in Figure 22.

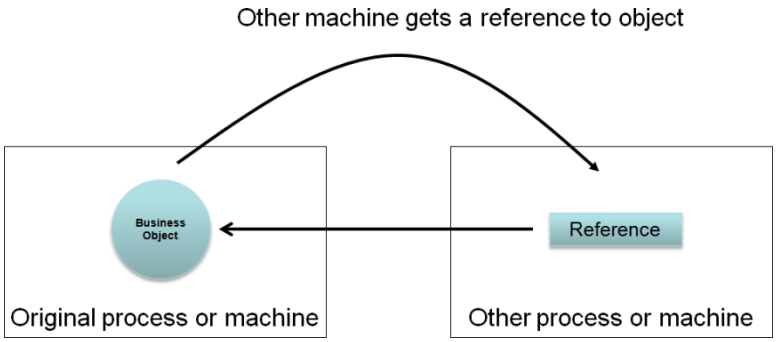

Figure 22. Calling an object by reference

This capability still exists in .NET but hasn't been a recommended approach since .NET 3.0 became available. These days the common approach is to expose a remote object through a web or REST API. Though it is an imprecise analogy, you can think of the REST API endpoint as a reference to the remote object.

By using this reference, the other machine can interact with the object. Because the object is still on the original machine, any property or method calls are sent across the network, and the results are returned across the network. This scheme is only useful if the object is designed so it can be used with few method calls; just one is ideal (and required to implement a REST API)!

The best practice designs for remote objects or services call for each method on the object to do all its work in a single method call for precisely this reason, thereby sacrificing "proper" object-oriented design in order to reduce latency.

These types of objects are stuck, or *anchored*, on the original machine or process where they were created. An anchored object never moves; it's accessed via references.

Mobile Objects

The concept of mobile objects relies on the idea that an object can be passed from one process to another, or from one machine to another, *by value*. This means that the object is physically copied from the original process or machine to the other process or machine, as shown in Figure 23.

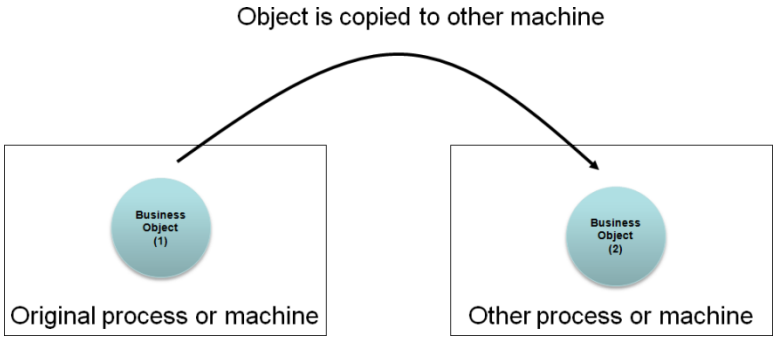

Figure 23. Passing a physical copy of an object across the network

Because the other machine gets a copy of the object, it can interact with the object locally. This means that there's effectively no performance overhead involved in calling properties or methods on the object—the only cost was in copying the object's data across the network in the first place.

> One caveat here is that transferring a large object across the network can cause a performance problem. Returning a JSON or XML document that contains a great deal of data can take a long time. This is true of all mobile objects, including business objects. You need to be careful in your application design in order to avoid retrieving large sets of data.

Objects that can move from process to process or from machine to machine are *mobile objects*. Examples of mobile objects include JSON and XML documents, and the business objects created in this book. Mobile objects aren't stuck in a single place but can move to where they're most needed.

To create one in .NET, add the `Serializable` attribute to your class definition. You may also optionally implement the `ISerializable` interface. I'll discuss this further in Chapter 2, but the following illustrates the start of a class that defines a mobile object:

```
[Serializable]
public class MyMobileClass
{
  private string _data;
}
```

When using WCF you might choose instead to use the `DataContract` and `DataMember` attributes:

```
[DataContract]
public class MyMobileClass
{
  [DataMember]
  private string _data;
}
```

Either approach works, but the `Serializable` attribute is often better for this scenario because it uses an opt-out approach. All fields of the object are serialized unless you explicitly prevent the serialization. The `DataContract` approach is opt-in, so you must remember to mark every field for serialization. Forget one, and you've introduced a hard-to-find bug.

Again, the .NET Framework takes care of the details, so an object of this type can be passed as a parameter to a method call or as the return value from a function. The object will be copied from the original machine to the machine where the method is running.

The CSLA framework provides a third alternative, optimized for use by CSLA. This `MobileFormatter` type requires the use of the `Serializable` attribute, and that the type implement an `IMobileObject` interface. I'll discuss the use of `MobileFormatter` (and assume its use) through this book series.

It is important to understand that the *code* for the object isn't automatically moved across the network. Before an object can move from machine to machine, both machines must have the .NET assembly containing the object's code installed. Only the object's serialized data is moved across the network by .NET. Installing the required assemblies is handled by standard .NET deployment and app update technologies.

When to Use Which Mechanism

The .NET Framework supports all the mechanisms discussed, so you can choose to create your objects as local, anchored, or mobile, depending on the requirements of your design. As you might guess, there are good reasons for each approach.

Most .NET types, including the various UI types used by UWP, Xamarin, WPF, Windows Forms and ASP.NET objects are all local—they're inaccessible from outside the processes in which they were created. The assumption is that other applications shouldn't be allowed to just reach into your program and manipulate your UI objects.

Anchored objects are important because they will always run on a specific machine. If you write an object that interacts with a database, you'll want to ensure that the object always runs on a machine that has access to the database. Because of this, anchored objects are typically used on application servers.

Many business objects, on the other hand, will be more useful if they *can* move from the application server to a client or web server, as needed. By creating business objects as mobile objects, you can pass smart data from machine to machine, thereby reusing your business logic anywhere the business data is sent.

Typically, anchored and mobile objects are used in concert. Later in the book, I'll show how to use an anchored object on the application server to ensure that specific methods are run *on that server*. Then mobile objects will be passed as parameters to those methods, which will cause those mobile objects to move from the client to the server. Some of the anchored server-side methods will return mobile objects as results, in which case the mobile object will move from the server back to the client.

Passing Mobile Objects by Reference

There's a piece of terminology here that can get confusing. So far, I've loosely associated anchored objects with the concept of "passing by reference," and mobile objects as being "passed by value." This makes sense on an intuitive level, because anchored objects provide a reference, though mobile objects provide the object (and its values). However, the terms "by reference" and "by value" have come to mean other things over the years.

The original idea of passing a value "by reference" was that there would be only one set of data—one object—and any code could get a reference to that single entity. Any changes made to that entity by any code would therefore be immediately visible to any other code.

The original idea of passing a value "by value" was that a copy of the original value would be made. Any code could get a copy of the original value, but any changes made to that copy weren't reflected in the original value. That makes sense, because the changes were made to a copy, not to the original value.

In distributed applications, things get a little more complicated, but the previous definitions remain true. An object can be passed by reference so that all machines have a reference to the same object on a server. An object can be passed by value, so that a copy of the object is made. So far, so good. Now, what happens if you mark an object as `Serializable` (that is, mark it as a mobile object), and then *intentionally* pass it by reference? It turns out that the object is passed by value, but the .NET Framework attempts to provide the illusion that the object was passed by reference.

To be more specific, in this scenario, the object is copied across the network as if it were being passed by value. The difference is that the object is then returned to the calling code when the method is complete, and the reference to the original object is replaced with a reference to this new version, as shown in Figure 24.

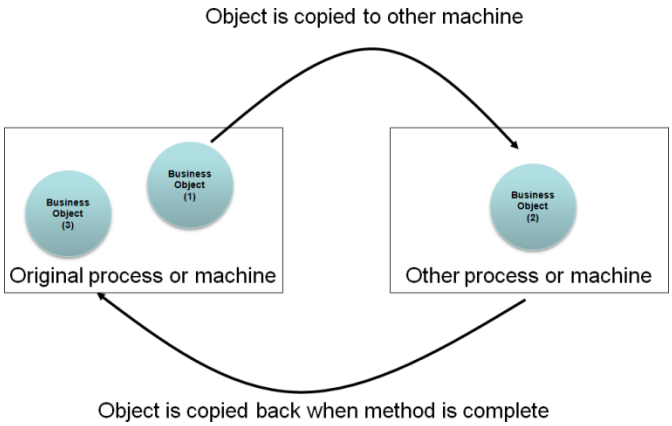

Figure 24. Passing a copy of the object to the server and getting a copy back

This is potentially dangerous, because *other* references to the original object continue to point to that original object—only this one particular reference is updated. You can end up with two different versions of the same object on the machine, with some references pointing to the new one and some to the old one.

CSLA includes the ability to merge this newly returned object back into the original object, so the original object then includes any field or property value changes that were made by the other machine.

Complete Encapsulation

Hopefully, at this point, your imagination is engaged by the potential of mobile objects. The flexibility of being able to choose between local, anchored, and mobile objects is powerful, and opens new architectural approaches.

I've already discussed the idea of sharing the Business layer across machines, and it's probably obvious that the concept of mobile objects is exactly what's needed to implement such a shared layer. But what does this all mean for the *design* of the layers? Given a set of mobile objects in the Business layer, what's the impact on the Interface control and Data Access layers with which the objects interact?

Impact on the UI Layer

What it means for the Interface Control layer, is that the business objects will contain all the business logic. The UI developer can code each form, page, service or workflow activity using the business objects, thereby relying on them to perform any validation or manipulation of the data. This means that the UI code can focus entirely on displaying the data, interacting with the user, and providing a rich, interactive experience.

More importantly, because the business objects are mobile, they'll end up running in the same process as the Interface control code. Any property or method calls from the UI code to the business object will occur locally without network latency, marshaling, or any other performance overhead.

Impact on the Data Access Layer

A traditional Data Access layer consists of a set of methods or services that interact with the database, and with the objects that encapsulate data. The data access code itself is typically outside the objects, rather than being encapsulated within the objects. Notice that this breaks encapsulation, because it means that the objects' data must be externalized to be handled by the data access code.

The framework created in this book allows for the data access code to be encapsulated within the business objects or externalized into a separate set of objects. As you'll see in later books, there are both performance and maintainability benefits to including the data access code directly inside each business object. These benefits must be considered along with the security and manageability benefits of having the code external.

Either way, the concept of a Data Access layer is of key importance. Maintaining a strong logical separation between the data access code and business logic is highly beneficial, as discussed earlier in this chapter. Obviously, having a totally separate set of data access objects is one way to clearly implement a Data Access layer.

Architectures and Frameworks

The discussion so far has focused mainly on architectures: logical architectures that define the separation of responsibilities in an application, and physical architectures that define the locations where the logical layers will run in various configurations. I've also discussed the use of object-oriented design and the concepts behind mobile objects.

Although all of these are important and must be thought through in detail, you don't want to have to go through this process every time you need to build an application. It would be preferable to have the architecture and design solidified into reusable code that could be used to build all your applications. What you want is an *application framework*. A framework codifies an architecture and design in order to promote reuse and increase productivity.

The typical development process starts with requirements gathering and analysis, followed by a period of architectural discussion and decision making. Next comes the application design: first, the low-level concepts to support the architecture, and then the business-level concepts that matter to the end users. With the design completed, developers typically spend a fair amount of time implementing the low-level functions that support the business coding that comes later.

All the architectural discussions, decision making, designing, and coding can be a lot of fun. Unfortunately, it doesn't directly contribute anything to the end goal of writing business logic and providing business functionality. This low-level supporting technology is merely "plumbing" that must exist in order to create business applications. It's an overhead that in the long term you should be able to do once, and then reuse across many business application–development efforts.

In the software world, the easiest way to reduce overhead is to increase reuse, and the best way to get reuse out of an architecture (both design and coding) is to codify it into a framework.

This doesn't mean that *application* analysis and design are unimportant—quite the opposite! People typically spend far too little time analyzing business requirements and developing good application designs to meet those business needs. Part of the reason is that they often end up

spending substantial amounts of time analyzing and designing the “plumbing” that supports the business application, and then run out of time to analyze the business issues themselves.

What I’m proposing here is to reduce the time spent analyzing and designing the low-level plumbing by creating a framework that can be used across many business applications. Is the framework created in this book ideal for every application and every organization? Certainly not!

Before choosing any software framework, you should fully understand the philosophy and goals behind the framework. If you agree with the philosophy, you’ll probably find the framework fits well into your development process. If you disagree with the philosophy, you’ll probably end up fighting the framework at every turn.

By this point in the book you should have a good understanding of the thought process and philosophy behind the architecture for CSLA .NET. If you find this philosophy and its benefits to be compelling, you’ll probably find the CSLA .NET framework useful and productive. On the other hand, if you’ve read this far and vehemently disagree with my thought process, you may find CSLA .NET to be quite frustrating, because it exists to help people implement applications based on the ideas I’ve discussed in this book.

Conclusion

In this chapter, I’ve focused on the theory behind distributed systems—specifically, those based on mobile objects. The key to success in designing a distributed system is to keep clear the distinction between a logical and a physical architecture.

Logical architectures exist to define the separation between the different types of code in an application. The goal of a good logical architecture is to make code more maintainable, understandable, and reusable. A logical architecture must also define enough layers to enable any physical architectures that may be required.

A physical architecture defines the machines on which the application will run. An application with several logical layers can still run on a single machine. You also might configure that same logical architecture to run on various client and server machines. The goal of a good physical architecture is to achieve the best trade-off between performance, scalability, security, and fault tolerance within your specific environment.

The trade-offs in a physical architecture for a smart client application are very different from those for a web application. A smart client application will typically trade performance against scalability, and a web application will typically trade performance against security.

The next chapter will dive deeper into the CSLA .NET framework to show how it supports the architecture and ideas I discussed in this chapter.

Chapter 3:
CSLA .NET Framework

In Chapter 2, I discussed some general concepts about physical and logical n-tier architecture, including a 5-layer model for describing systems logically. In this chapter, I'll take that 5-layer logical model and explain how that architecture is supported by the CSLA .NET framework. Specifically, CSLA .NET maps the architectural layers into the technologies illustrated in Table 3.

Layer	Technologies
Interface	Xamarin, ASP.NET, WebAssembly, UWP, WPF, Windows Forms
Interface Control	Viewmodel object, Controller object, Code-behind, Service implementation
Business	CSLA .NET business domain objects
Data Access	ADO.NET, Entity Framework, Dapper, and many others
Data Storage	SQL Server, Oracle, PostgreSQL, NoSQL, REST services, and many others

Table 3. Mapping the logical layers to technologies

The CSLA .NET framework is primarily focused on the Business layer, though it does include some functionality to help support various types of interface and data access models.

There are already powerful technologies for building Windows, web, mobile and service-based interfaces. However, there is often room to streamline interaction between those technologies and the business objects, and so CSLA .NET does provide some limited functionality to support each type of interface technology.

Also, there are already powerful data-storage options available, including SQL Server, Oracle, MySQL, XML documents, and so forth, along with quite an array of different data access technologies available in the .NET Framework. Supporting the concept of mobile objects does mean that CSLA .NET provides a broad structure within which your data access code must exist, and so the framework does provide some functionality in the data access area.

Recognizing that these preexisting technologies are ideal for building the Interface and Interface Control layers, as well as for handling data storage, allows business developers to focus on the parts of the application that have the least technological support, where the highest return on investment occurs through reuse. Analyzing, designing, implementing, testing, and maintaining business logic is incredibly expensive. The more reuse achieved, the lower long-term application costs become. The easier it is to maintain and modify this logic, the lower costs will be over time.

> This is not to say that additional frameworks for UI creation or simplification of data access are bad ideas. On the contrary, such frameworks can be quite complementary to the ideas presented in this book; and the combination of several frameworks can help lower costs even further.

When I set out to create the architecture and framework discussed in this book, I started with the following set of high-level guidelines:

- Simplify the task of creating object-oriented applications in a distributed .NET environment.
- Consistent programming model and reuse of Business layer code across different interface technologies and .NET platforms (anywhere you can write code with C# or VB).
- The interface developer (Windows, web, mobile, or service) should never see or be aware of SQL, ADO.NET, or other raw data concepts, but should instead rely on a purely object-oriented model of the problem domain.
- The business classes should provide total encapsulation of business logic, including validation, manipulation, calculation and authorization. Everything pertaining to a concept or actor in a use case of the problem domain should be found within a single class.
- It should be possible to achieve clean separation between the business logic code and the data access code.
- It should be relatively easy to create code generators, or templates for existing code generation tools, to assist in the creation of business classes.
- Provide an n-layer logical architecture that can be easily reconfigured to run on one to four physical tiers.
- Use complex features in .NET—but those should be largely hidden and automated (serialization, network transport technologies, security, deployment, and so forth).
- The concepts present in the framework from its inception should carry forward, including validation, authorization, n-level undo, and object-state tracking (`IsNew`, `IsDirty`, `IsBusy`, `IsDeleted`).

In this chapter, I'll focus on the design of CSLA .NET, a framework that allows business developers to make use of object-oriented design and programming with these guidelines in mind. Before I get into the design of the framework, let's discuss some of the specific goals I was attempting to achieve.

Basic Design Goals

When creating object-oriented applications, the ideal situation is that any nonbusiness objects will already exist. This includes UI controls, data access objects, and so forth. In that case, all developers need to do is focus on creating, debugging, and testing the business objects themselves, thereby ensuring that each one encapsulates the data and business logic needed to make the application work.

As rich as the .NET Framework is, it doesn't provide all the nonbusiness objects needed in order to create most applications. All the basic tools are there, but there's a fair amount of work to be done before you can sit down and write business logic. There's a set of higher-level functions and capabilities that are often needed but aren't provided by .NET right out of the box.

These include the following:

- Validation and maintaining a list of broken business rules
- Standard implementation of business rules
- Integrated authorization rules at the object and property levels
- Tracking whether an object's data has changed (is it "dirty"?)
- Tracking whether async rules are running for an object (is it "busy"?)
- Strongly typed collections of child objects (parent-child relationships)
- N-level undo capability
- A simple and abstract model for the UI developer
- Full support for data binding in all .NET interface technologies
- Saving objects to a database and getting them back again
- Custom authentication
- Enable appropriate extensibility

In all these cases, the .NET Framework provides all the pieces of the puzzle, but they must be put together to match your specialized requirements. What you *don't* want to do, is put them together for every business object or application. The goal is to put them together *once*, so that all these extra features are automatically available to all the business objects and applications.

Moreover, because the goal is to enable the implementation of *object-oriented* business systems, the core object-oriented concepts must also be preserved:

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

The result will be a framework consisting of several classes, organized into namespaces and assemblies.

Before getting into the details of the framework's design, let's discuss the desired set of features in more detail.

Business Rules

A lot of business logic involves the enforcement of *validation rules*. The fact that a given piece of data is required is a validation rule. The fact that one date must be later than another date is a validation rule. Some validation rules involve calculations, and others are merely toggles. You can think about validation rules as being either broken or not. And when one or more rules are broken the object is invalid.

A similar concept is the idea of *business rules* that might alter the state of the object. The fact that a given piece of text data must be all upper case is a business rule. The calculation of one

property value based on other property values is a business rule. Most business rules involve some level of calculation.

It is also a good idea to think about some authorization rules as being business rules. Any authorization rule that applies to a specific object instance, or properties of an object instance, is really a type of business rule. Even authorization rules that apply to a *type of object*, or a UI form or page are still business logic and would ideally be implemented in the Business layer.

CSLA .NET has had a business rules subsystem since version 2.0. The rules engine included starting in version 4.0 is flexible and powerful, while still being simple and approachable for many common scenarios.

The CSLA .NET business rules engine recognizes that consistency across all types of business rule is desirable, including:

- Hand-coded validation rules
- `DataAnnotation` attribute rules
- Business rules that affect the state or properties of the object
- Flexible authorization rules for each property or method
- Rules that run synchronously
- Rules that run asynchronously

Although there are certain differences between business, validation and authorization rules, the basic structure of each type of rule is consistent. Similarly, there are obviously differences between synchronous and asynchronous code, but the structure of the code and capabilities of a rule are comparable between those two models.

The result of a business rule is that the state or properties of the object (or another object in the object graph) are altered. Synchronous rules can directly make these changes, although asynchronous rules provide CSLA with a list of property values to update. Synchronous rules can also use the asynchronous approach, so generally it is best for a rule to allow CSLA to make the property changes.

The results of a validation rule are that the object, and usually a specific property, is valid or invalid. Additionally, the results of a validation rule can have different severities: error, warning or information, and the results include human-readable text providing some message about the result. Validation rules can be synchronous or asynchronous.

In Microsoft .NET Framework 3.5 we were introduced to the `System.ComponentModel.DataAnnotations` namespace and the concept of validation attributes. Microsoft has evolved these capabilities over time, and on some .NET platforms they provide a consistent and extensible model for building validation rules and applying them to properties or objects.

CSLA .NET transparently integrates the validation attribute concept, so these attributes work as part of the overall CSLA .NET business rule subsystem. You can use the attributes, or custom CSLA rules, or both at the same time.

CSLA .NET ensures that `DataAnnotations` attributes work properly on all platforms, even those (such as Xamarin, WebAssembly, WPF, and Windows Forms) where the attributes would otherwise be ignored.

Whether your validation rules are attributes or custom rules, when a validation rule returns a result (error, warning or information) that result is maintained by the business object in a list of broken rules. This list of broken rules is available to the object, as well as Presentation layer code, so it is possible to easily determine of the object is valid at any point in time. Perhaps even more importantly, having access to the list of broken rules allows the Presentation layer to show the user why an object is invalid.

This broken rules list provides the underlying data required to implement the `System.ComponentModel.IDataErrorInfo` and `System.ComponentModel.INotifyDataErrorInfo` interfaces defined by the various implementations of .NET across platforms. These interfaces are used by the data binding infrastructure in some .NET UI frameworks to automate the display of validation error results to the user.

Validation warning and information messages are not supported by some UI technologies, and on those platforms, you are responsible for displaying those messages in your interface. CSLA .NET provides helper controls for several UI frameworks, such as ASP.NET MVC, WPF, UWP and Xamarin, to make it easy to display these messages.

The result of a failed authorization rule is typically a security exception, whereas success means the property value is returned, or set, or the method is invoked as requested. CSLA .NET provides authorization rules that work against the Microsoft .NET role-based authorization model, but you can create your own authorization rules that work against any user or application metadata you choose.

It is important to understand that authorization rules are merely another type of business logic. There are many reasons why a user might be blocked from changing a specific property value, including the user's roles, but also including the state of other properties on the object. The extensible authorization rules feature of the business rules subsystem allows you to write rules tailored to the specific requirements of each of your business objects.

Tracking Whether the Object Has Changed

Another concept is that an object should keep track of whether its state data has been changed. This is important for the performance and efficiency of data updates. Typically, data should only be updated into the database if the data has changed. It's a waste of effort to update the database with values it already has! Although the UI developer *could* keep track of whether any values have changed, it's simpler to have the object take care of this detail, and it allows the object to better encapsulate its behaviors.

This can be implemented in several ways, ranging from keeping the previous values of all fields (allowing comparisons to see if they've changed) to saying that *any* change to a value (even "changing" it to its original value) will result in the object being marked as having changed.

The default implementation provided by CSLA .NET is to maintain a Boolean value that is set to `true` if any property is changed to a new value. It is possible to replace the default implementation

with a more sophisticated implementation, such as tracking the previous values of all changed properties.

Not only does this business object need access to this authorization information, but the Presentation layer does as well. Ideally, a good UI will change its display based on how the current user can interact with an object. To support this concept, the business framework will help the business objects expose the authorization rules such that they are accessible to the Presentation layer without duplicating the authorization rules themselves.

Strongly Typed Collections of Child Objects

The .NET Framework includes the `System.Collections.Generic` namespace, which contains a number of powerful collection objects, including `List<T>`, `Dictionary<TKey, TValue>`, and others. There's also `System.ComponentModel.BindingList<T>`, which provides data binding collection behaviors for Windows Forms (and some WPF controls) and `System.ComponentModel.ObservableCollection<T>` that provides data binding collection support for most WPF, UWP and Xamarin controls.

The wide variation in collection or list base types is a cause of confusion for many .NET developers, at least if they ever build applications for different interface technologies or use WPF controls that require `BindingList<T>`. The key issue is support for data binding across interface technologies that have been created over many years. The requirements of older UI technologies, including Web Forms, Windows Forms and some WPF controls, are quite different from modern UI technologies such as UWP, Xamarin, ASP.NET MVC, and most WPF controls. There's no common collection base class that works well across the old and new UI platforms.

From a CSLA .NET perspective, a collection of child objects needs to be able to indicate if any of the objects it contains have been changed. Although the business object developer could easily write code to loop through the child objects to discover whether any are marked as dirty, it makes a lot more sense to put this functionality into the framework's collection object. That way the feature is simply available for use. The same is true with validity: if any child object is invalid, then the collection should be able to report that it's invalid. If all child objects are valid, then the collection should report itself as being valid.

CSLA .NET includes a set of collection base classes that provide the business functionality I'm describing here around change tracking, validity and so forth. In fact, there are two sets of base classes; one that derives from `ObservableCollection<T>` and another that derives from `BindingList<T>`. They are the same from a CSLA perspective, but you'll need to choose the right ones to use based on your specific UI technology.

N-Level Undo Capability

Many older-style Windows applications (Windows Forms or WPF) provide their users with an interface that includes OK and Cancel buttons (or some variation on that theme). When the user clicks an OK button, the expectation is that any work the user has done will be saved. Likewise, when the user clicks a Cancel button, she expects that any changes she's made will be reversed or undone.

Simple applications can often deliver this functionality by saving the data to a database when the user clicks OK and discarding the data when they click Cancel. For slightly more complex

applications, the application must be able to undo any editing on a single object when the user presses the Esc key. (This is the case for a row of data being edited in a DataGridView: if the user presses Esc, the row of data should restore its original values.)

When applications become much more complex, these approaches won't work. Instead of undoing the changes to a single row of data in real time, you may need to be able to undo the changes to a row of data at some later stage.

> It is important to realize that the n-level undo capability implemented in the framework is *optional* and is designed to incur no overhead if it is not used.

Consider the case of an Invoice object that contains a collection of LineItem objects. The Invoice itself contains data that the user can edit, plus data that's derived from the collection. The TotalAmount property of an Invoice, for instance, is calculated by summing up the individual Amount properties of its LineItem objects. Figure 25 illustrates this arrangement.

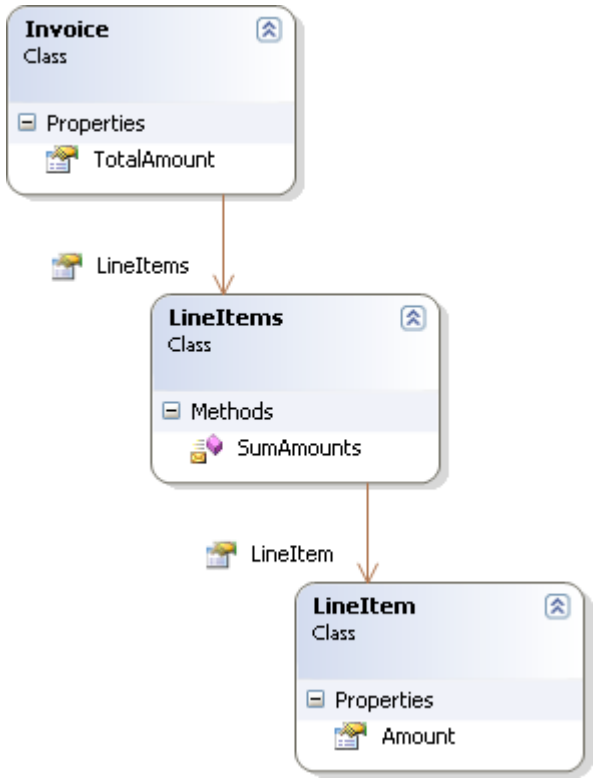

Figure 25. Relationship between the Invoice, LineItems, and LineItem classes

The UI may allow the user to edit the LineItem objects, and then press Enter to accept the changes to the item, or Esc to undo them. Even if the user chooses to accept changes to some LineItem objects, they can still choose to cancel the changes on the Invoice itself. Of course, the only way to reset the Invoice object to its original state is to restore the states of the LineItem objects as well; including any changes to specific LineItem objects that might have been "accepted" earlier.

As if this weren't enough, many applications have more complex hierarchies of objects and subobjects (which I'll call *child objects*). Perhaps the individual `LineItem` objects each have a collection of `Component` objects beneath them. Each one represents one of the components sold to the customer that make up the specific line item, as shown in Figure 26.

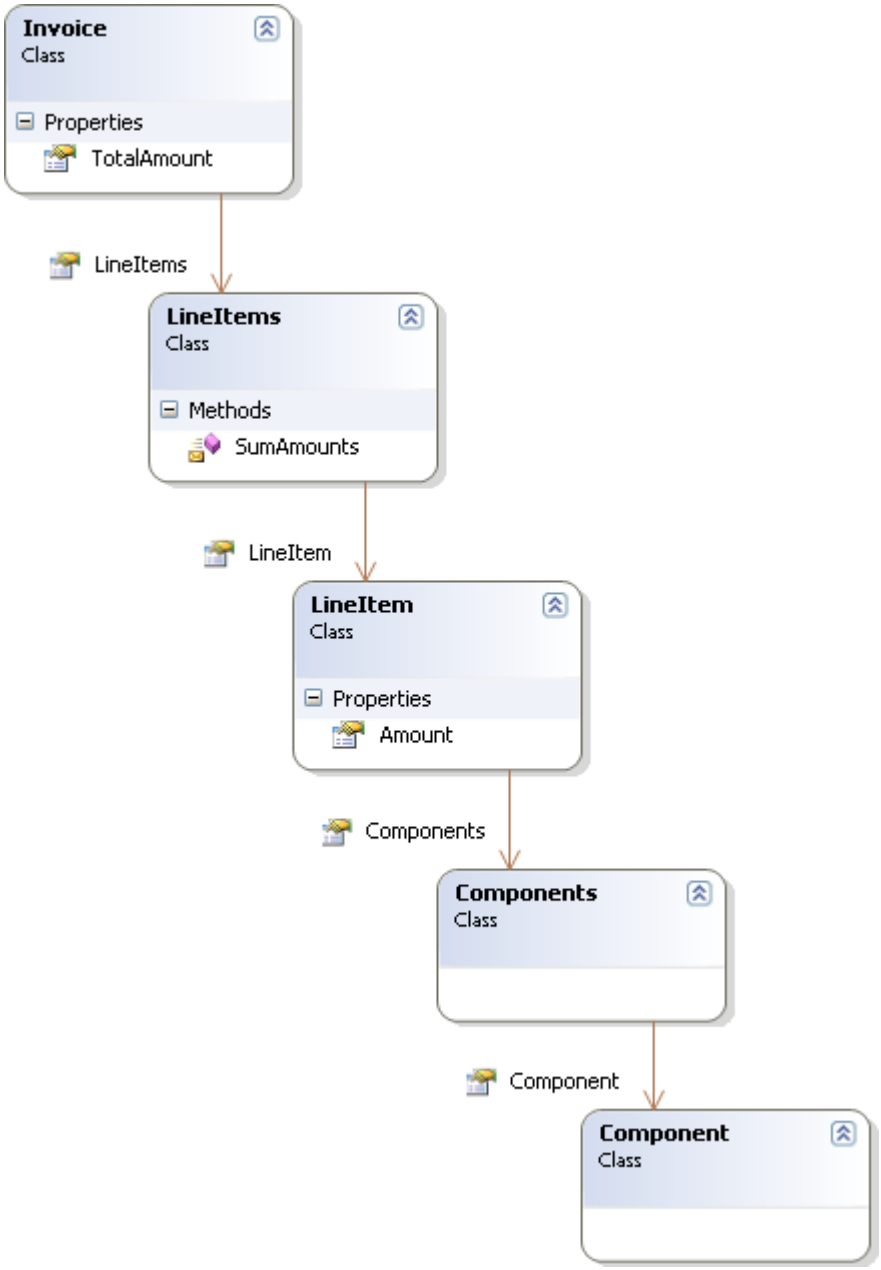

Figure 26. Class diagram showing a more complex set of class relationships

Now things get even more complicated. If the user edits a `Component` object, those changes ultimately impact the state of the `Invoice` object itself. Of course, changing a `Component` also changes the state of the `LineItem` object that owns the `Component`.

The user might accept changes to a `Component`, but cancel the changes to its parent `LineItem` object, thereby forcing an undo operation to reverse *accepted* changes to the `Component`. Or in an even more complex scenario, the user may accept the changes to a `Component` and its parent `LineItem`, only to cancel the `Invoice`. This would force an undo operation that reverses all those changes to the child objects.

Implementing an undo mechanism to support such n-level scenarios isn't trivial. The application must implement code to take a snapshot" of the state of each object before it's edited, so that changes can be reversed later. The application might even need to take more than one snapshot of an object's state at different points in the editing process, so that the object can revert to the appropriate point, based on when the user chooses to accept or cancel any edits.

> This multilevel undo capability flows from the user's expectations. Consider a typical word processor, where the user can undo multiple times to restore the content to ever-earlier states.

And the collection objects are every bit as complex as the business objects themselves. The application must handle the simple case in which a user edits an existing `LineItem`, but it must also handle the case in which a user adds a new `LineItem` and then cancels changes to the parent or grandparent, resulting in the new `LineItem` being discarded. Equally, it must handle the case in which the user *deletes* a `LineItem` and then cancels changes to the parent or grandparent, thereby causing that deleted object to be restored to the collection as though nothing had ever happened.

Things get even *more* complex if you consider that the framework keeps a list of broken validation rules for each object. If the user changes an object's data so that the object becomes invalid, but then cancels the changes, the original state of the object must be restored. The reverse is true as well: an object may start out invalid (perhaps because a required field is blank), so the user must edit data until it becomes valid. If the user later cancels the object (or its parent, grandparent, etc.), then the object must become *invalid* once again, because it will be restored to its original invalid state.

Fortunately, this is easily handled by treating the broken rules and validity of each object as part of that object's state. When an undo operation occurs, not only is the object's core state restored, but so is the list of broken rules associated with that state. The object and its rules are restored together.

N-level undo is a perfect example of complex code that shouldn't be written into every business object. Instead, this functionality should be written *once*, so that all business objects support the concept and behave the way we want them to. This functionality will be incorporated directly into the business object framework—but at the same time, the framework must be sensitive to the different environments in which the objects will be used. Although n-level undo is of high importance when building sophisticated Windows user experiences, it's virtually useless in a typical web environment.

In most modern applications, the user typically doesn't have a Cancel button. They either accept the changes, or navigate away to another task, allowing the application to discard the changed object. In this regard, these environments are much simpler, so if n-level undo isn't useful to the developer, it doesn't incur any overhead if it isn't used.

Abstracted Object Persistence for the UI Developer

At this point, I've discussed some of the business object features that the framework will support. One of the key reasons for providing these features is to make the business object support Windows, mobile, and web-style user experiences with minimal work on the part of the UI developer. In fact, this should be an overarching goal when you're designing business objects for a system. The UI developer should be able to rely on the objects to provide business logic, data, and related services in a consistent manner.

Beyond all the features already covered, is the issue of creating new objects, retrieving existing data, and updating objects in some data store. I'll discuss the *process* of object persistence later in the chapter, but this topic should be considered from the UI developer's perspective. Should the UI developer be aware of any application servers? Should they be aware of any database servers? Or should they interact with a set of abstract objects?

In summary, the UI developer should not be aware of any details around application servers, how to communicate with application servers, network protocols and libraries, or any of those technical details. Those have no business value and detract from a focus on the business problem being solved.

CSLA .NET provides a component called the data portal to help provide an abstract mechanism by which the UI developer can create, retrieve, and update domain objects. The data portal uses configuration to determine whether there's a separate application server, and what network technologies and protocols to use when communicating with any remote application server.

From the UI developer's perspective, they can use the data portal directly, or interactions with the data portal can be abstracted within `static` factory methods in each business class.

Directly using the data portal, the code might be something like this:

```
var result = await Csla.DataPortal.CreateAsync<Customer>();
```

The data portal examines its configuration to determine whether there is a remote application server, and if so how to communicate with that server over the network. Regardless, a new instance of the `Customer` type is created, loaded with any default values, and returned to the calling code.

This may seem like a lot of work just to create a new, empty object, but it's the retrieval of default values that makes it necessary. If the application has objects that don't need default values, or if you're willing to hard-code the defaults, you can avoid some of the work by having the UI create the object with the `new` keyword. Be aware that many business applications have configurable default values for objects that must be loaded from the database; and that means the application server must load them.

Retrieving an *existing* object follows the same basic procedure. The UI passes criteria to the application server, which uses the criteria to create a new object and load it with the appropriate data from the database. The populated object is then returned to the UI for use. The UI code might be something like this:

```
var result = await Csla.DataPortal.FetchAsync<Customer>(criteria);
```

Updating an object happens when the UI calls the application server and passes the object to the server. The server can then take the data from the object and store it in the database. Because the

update process may result in changes to the object's state, the newly saved and updated object is then returned to the UI. The UI code might be something like this:

```
obj = await obj.SaveAsync();
```

CSLA does include the ability to merge the result of an update operation into the existing object graph. For example:

```
await obj.SaveAndMergeAsync();
```

The result of this code is that the original object's data is persisted through the data portal, and the resulting changes (if any) are then merged back into the original object graph.

This code seems pretty straightforward and provides a consistent pattern for all UI developers to create, retrieve, and update business domain objects. The only drawback is that the UI developer is forced to interact with the data portal directly, rather than only interacting with abstract business types.

An alternative is to wrap the use of the data portal within `static` factory methods in the business classes. For example:

```
[Serializable]
public class Customer
{
  public static async Task<Customer> NewCustomerAsync()
  {
    return await Csla.DataPortal.CreateAsync<Customer>();
  }
}
```

Then the UI code could use this method without first creating a `Customer` object, as follows:

```
Customer cust = await Customer.NewCustomerAsync();
```

A common example of this tactic within the .NET Framework itself is the `Guid` class, whereby a `static` method is used to create new `Guid` values, as follows:

```
Guid myGuid = Guid.NewGuid();
```

This accomplishes the goal of making the UI code only deal with business domain types, and not with any "plumbing" concepts like the data portal.

However, the use of `static` factory methods does prevent the use of dependency injection and can make unit testing more difficult. To that end, it is common for UI code to interact with the data portal directly via the `IDataPortal` interface, enabling the use of dependency injection and mocking techniques.

Supporting Data Binding

Data binding is an important feature of each UI technology on the Microsoft platforms. Data binding allows developers to create forms and populate them with data with almost no custom code. The controls on a form are "bound" to specific fields from a data source (such as an entity object, `DataSet` or a business object).

Data binding is provided in UWP, Xamarin.Forms, WPF, ASP.NET MVC, Windows Forms, and Web Forms. The primary benefits or drivers for using data binding in .NET development include the following:

- Data binding offers good performance, control, and flexibility.
- Data binding can be used to link controls to properties of business objects.
- Data binding can dramatically reduce the amount of code in the UI.
- Data binding is sometimes *faster* than manual coding, especially when loading data into list boxes, grids, or other complex controls.

Of these, the biggest single benefit is the dramatic reduction in the amount of UI code that must be written and maintained. Combined with the performance, control, and flexibility of .NET data binding, the reduction in code makes it an attractive technology for UI development.

Enabling the Objects for Data Binding

Although data binding can be used to bind against any object or any collection of homogeneous objects, there are some things that object developers can do to make data binding work better. Implementing these "extra" features enables data binding to do more work for us and provide the user with a superior experience. The .NET `DataSet` object, for instance, implements these extra features in order to provide full data binding support to Windows Forms, WPF, and ASP.NET developers.

The IEditableObject Interface

All editable business objects should implement the interface called `System.ComponentModel.IEditableObject`. This interface is designed to support a simple, one-level undo capability, and is used by simple forms-based data binding and complex grid-based data binding alike.

In the forms-based model, `IEditableObject` allows the data binding infrastructure to notify the business object before the user edits it, so that the object can take a snapshot of its values. Later, the application can tell the object whether to apply or cancel those changes, based on the user's actions. In the grid-based model, each of the objects is displayed in a row within the grid. In this case, the interface allows the data binding infrastructure to notify the object when its row is being edited, and then whether to accept or undo the changes based on the user's actions. Typically, grids perform an undo operation if the user presses the `Esc` key, and an accept operation if the user presses `Enter` or moves off that row in the grid by any other means.

The INotifyPropertyChanged Interface

Editable business objects need to raise events to notify data binding any time their data values change. Changes that are caused directly by the user editing a field in a bound control are supported automatically—with the exception that if the object updates a property value through *code* rather than by direct user editing, the object needs to notify the data binding infrastructure that a refresh of the display is required.

The .NET Framework defines `System.ComponentModel.INotifyPropertyChanged`, which should be implemented by any bindable object. This interface defines the `PropertyChanged` event that data binding can handle to detect changes to data in the object.

It is important to realize that Windows Forms treats the PropertyChanged event differently from other UI technologies. In Windows Forms, every PropertyChanged event causes a refresh of all

bound controls on the form. Whereas, in other technologies only controls bound to the changed property are updated. CSLA .NET includes code to help you optimize the use of PropertyChanged for your UI technology without impacting the code in your business classes or UI.

The INotifyPropertyChanging Interface

In .NET Framework 3.5 Microsoft introduced the `System.ComponentModel.INotifyPropertyChanging` interface so business objects can indicate when a property is about to be changed. Strictly speaking, this interface is optional and isn't (currently) used by data binding. For completeness, it is nonetheless recommended that this interface be implemented when implementing `INotifyPropertyChanged`.

The `INotifyPropertyChanging` interface defines the `PropertyChanging` event that is raised before a property value is changed, as a complement to the `PropertyChanged` event that is raised *after* a property value has changed.

The INotifyCollectionChanged Interface

In .NET Framework 3.0, Microsoft introduced a new option for building lists for data binding. This new option only works with all UI technologies other than Windows Forms and some WPF controls. The `System.ComponentModel.INotifyCollectionChanged` interface defines a `CollectionChanged` event that is raised by any list implementing the interface. The simplest way to do this is to have the collection classes inherit from `System.ComponentModel.ObservableCollection<T>`. This generic class implements the interface and related behaviors.

This interface is used by datagrid and other list type controls, so the UI can automatically update as elements are added, removed, and edited within the collection. The most commonly used CSLA .NET collection base classes ultimately subclass `ObservableCollection<T>`.

The IBindingList Interface

All business collections that need to support Windows Forms (and some WPF controls) should implement the interface called `System.ComponentModel.IBindingList`. The simplest way to do this is to have the collection classes inherit from `System.ComponentModel.BindingList<T>`. This generic class implements all the collection interfaces required to support data binding for Windows Forms.

This interface is used in grid-based binding, in which it allows the control that's displaying the contents of the collection to be notified by the collection any time an item is added, removed, or edited, so that the display can be updated. Without this interface, there's no way for the data binding infrastructure to notify the grid that the underlying data has changed, so the user won't see changes as they happen.

When implementing a list or collection you must choose to implement either `IBindingList` or `INotifyCollectionChanged`. If you implement both then data binding in WPF will become confused, as it honors *both interfaces* and will always get duplicate events for any change to the list.

CSLA .NET includes a set of collection base classes that ultimately subclass `BindingList<T>`, and you should use these base classes when building business objects that will be used by Windows Forms.

Events and Serialization

The events that are raised by business collections and business objects are all valuable. Events support the data binding infrastructure and enable utilization of its full potential. Unfortunately, there's a conflict between the idea of objects raising events and the use of .NET serialization via the `Serializable` attribute.

When an object is marked as `Serializable`, the .NET Framework is told that it can pass the object across the network by value. As part of this process, the object will be automatically converted into a byte stream by the .NET runtime. It also means that any other objects *referenced* by the object will be serialized into the same byte stream, unless the field representing it is marked with the `NonSerialized` attribute. What may not be immediately obvious is that *events create an object reference behind the scenes*.

When an object declares and raises an event, that event is delivered to *any* object that has a handler for the event. WPF forms and Windows Forms often handle events from objects, as illustrated in Figure 27.

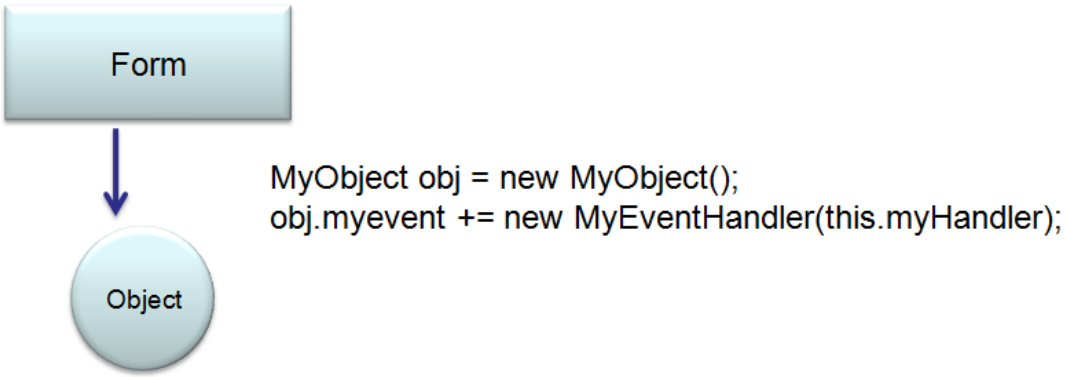

Figure 27. A Windows form referencing a business object

How does the event get delivered to the handling object? Well, it turns out that behind every event is a *delegate*—a strongly typed reference that points back to the handling object. This means that any object that raises events can end up with bidirectional references between the object and the other object/entity that is handling those events, as shown in Figure 28.

Even though this back reference isn't visible to developers, it's completely visible to the .NET serialization infrastructure. When serializing an object, the serialization mechanism will trace this reference and attempt to serialize any objects (including forms) that are handling the events! Obviously, this is rarely desirable. In fact, if the handling object is a form, this will fail outright with a runtime error, because forms aren't serializable.

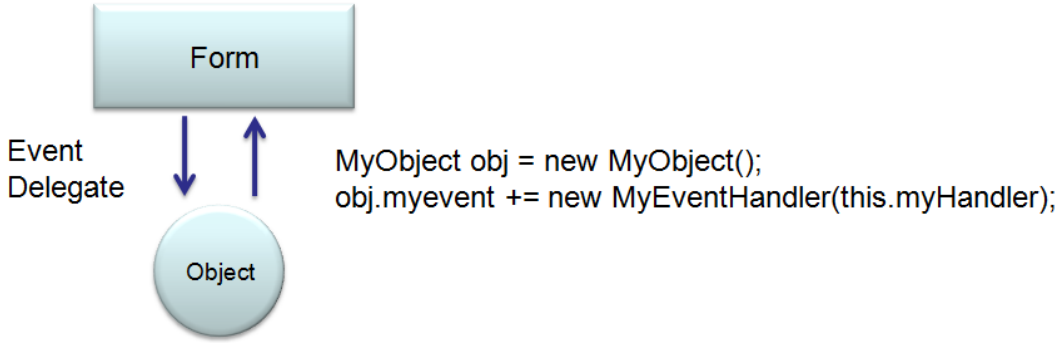

Figure 28. Handling an event on an object causes a back reference to the form.

> If any non-serializable object handles events that are raised by a serializable object, you'll be unable to serialize the object because the .NET runtime serialization process will error out.

Solving this means marking the events as `NonSerialized`. It turns out that this requires a bit of special syntax when dealing with events. Specifically, a more explicit block structure must be used to declare the event. This approach allows manual declaration of the delegate field so it is possible to mark that field as `NonSerialized`. The `ObservableCollection<T>` and `BindingList<T>` classes already declare their events in this manner, so this issue only pertains to the implementation of `INotifyPropertyChanged` and `INotifyPropertyChanging` (or any custom events you choose to declare in your business classes).

The IDataErrorInfo and INotifyDataErrorInfo Interfaces

Earlier, I discussed the need for objects to implement business rules and expose information about broken rules to the UI. The `System.ComponentModel.IDataErrorInfo` and `System.ComponentModel.INotifyDataErrorInfo` interfaces are designed to allow data binding to request information about broken validation rules from a data source. The `IDataErrorInfo` interface has existed since the start of Windows Forms, and `INotifyDataErrorInfo` exists only in more modern UI frameworks. It works with asynchronous validation rules.

Given that the object framework will already help the objects manage a list of all currently broken validation rules, we'll already have the tools needed to easily implement these interfaces.

By implementing this interface, the objects will automatically support the feedback mechanisms built into the Windows Forms `DataGridView` and `ErrorProvider` controls.

Object Persistence and Object-Relational Mapping

One of the biggest challenges facing a business developer building an object-oriented system is that a good object model is almost never the same as a good relational data model. Because most data is stored in relational databases using a relational model, we're faced with the significant problem of translating that data into an object model for processing, and then changing it back to a relational model later on to persist the data from the objects back into the data store.

> CSLA .NET doesn't *require* a relational model, but because that is the most common data storage technology, I focus on it quite a bit. You should remember that the

> concepts and code shown in this chapter can be used against XML files, object databases, or almost any other data store you are likely to use.

Relational vs. Object Modeling

Before going any further, let's make sure we're in agreement that object models aren't the same as relational models. Relational models are primarily concerned with the efficient storage of data, so that replication is minimized. Relational modeling is governed by the rules of normalization, and almost all databases are designed to meet at least the third normal form. In this form, it's quite likely that the data for any given business concept or entity is split between multiple tables in the database in order to avoid any duplication of data.

Object models, on the other hand, are primarily concerned with modeling *behavior*, not data. It's not the data that defines the object, but the role the object plays within your business domain. Every object should have one clear responsibility and a limited number of behaviors focused on fulfilling that responsibility.

> I recommend the book *Object Thinking*, by David West (DV-Microsoft Professional, 2004), for some good insight into behavioral object modeling and design. Though my ideas differ somewhat from those in *Object Thinking*, I use many of the concepts and language from that book in my own object-oriented design work and in this book.

For instance, a `CustomerEdit` object may be responsible for *adding and editing customer data*. A `CustomerInfo` object in the same application may be responsible for *providing read-only access to customer data*. Both objects will use the same data from the same database and table, but they provide different behaviors.

Similarly, an `InvoiceEdit` object may be responsible for *adding and editing invoice data*. But invoices include some customer data. A naïve solution is to have the `InvoiceEdit` object make use of the aforementioned `CustomerEdit` object, but that's not a good answer. That `CustomerEdit` object should only be used in the case where the application is adding or editing customer data—something that isn't occurring while working with invoices. Instead, the `InvoiceEdit` object should directly interact with the customer data it needs to do its job.

Through these two examples, it should be clear that sometimes multiple objects will use the same relational data. In other cases, a single object will use relational data from different data entities. In the end, the same customer data is being used by three different objects. The point, though, is that each one of these objects has a clearly defined responsibility that defines the object's *behavior*. Data is merely a resource the object needs to implement that behavior.

Behavioral Object-Oriented Design

It is a common trap to think that data in objects needs to be normalized like it is in a database. A better way to think about objects is to say that *behavior* should be normalized. The goal of object-oriented design is to avoid replication of *behavior*, not data.

> In object-oriented design, *behavior* should be normalized, not data.

At this point, most people are struggling. Most developers have spent years programming their brains to think relationally, and this view of object-oriented design flies directly in the face of that conditioning. Yet the key to the successful application of object-oriented design is to divorce object thinking from relational or data thinking.

Perhaps the most common objection at this point is this: if two objects (say, `CustomerEdit` and `InvoiceEdit`) both use the same data (say, the customer's name), how do you make sure that consistent business rules are applied to that data? And this is a good question.

The answer is that the behavior must be normalized. Business rules are merely a form of behavior. The business rule specifying that the customer name value is required, for instance, is just a behavior associated with that particular value.

Earlier in the chapter, I discussed the idea that a business or validation rule can be standardized into a consistent structure. Following this train of thought, every rule can be thought of as being an object, and in CSLA .NET this is literally true: your rules are implemented as classes.

Behavioral object-oriented design relies heavily on the concept of *collaboration*. Collaboration is the idea that an object should collaborate with other objects to do its work. If an object starts to become complex, you can break the problem into smaller, more digestible parts by moving some of the sub-behaviors into other objects that collaborate with the original object to accomplish the overall goal.

In the case of a required customer name value, there's a `BusinessRule` object that defines that behavior. Both the `CustomerEdit` and `InvoiceEdit` objects can collaborate with that `BusinessRule` object to ensure that the rule is consistently applied. As you can see in Figure 29, the rule is only implemented once, but is used as appropriate—effectively normalizing that behavior.

It could be argued that the `CustomerName` concept should become an object of its own, and that this object would implement the behaviors common to the field. Although this sounds good in an idealistic sense, it has serious performance and complexity drawbacks when implemented on development platforms such as .NET. Creating a custom object for every field in your application can rapidly become overwhelming, and such an approach makes the use of technologies like data binding very complex.

My approach of normalizing the rules themselves provides a workable compromise; providing a high level of code reuse while still offering good performance and allowing the application to take advantage of all the features of the .NET platform.

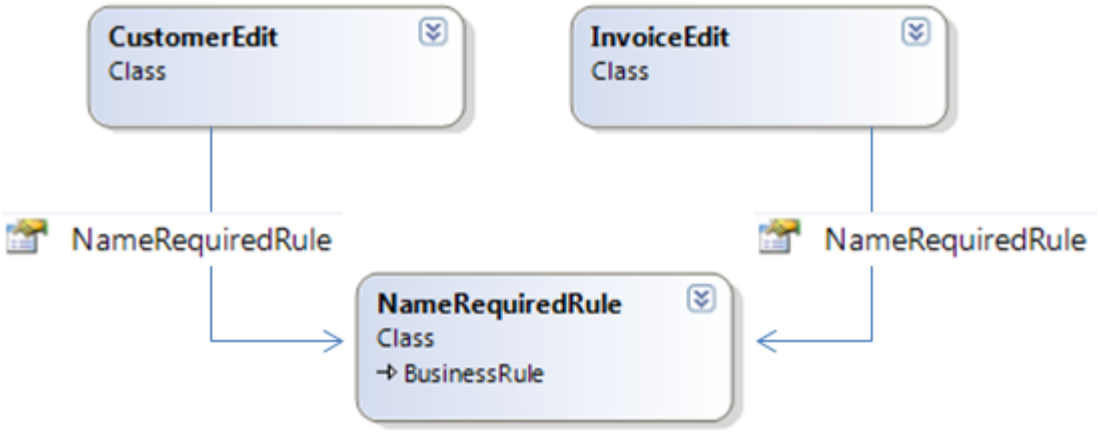

Figure 29. Normalizing the customer name required behavior

In fact, the idea that a string value is required is so pervasive that it can be normalized to a general `StringRequired` rule that can be used by any object with a required property anywhere in an application. In Chapter 5, I'll implement a `CommonRules` class containing several common validation rules of this nature.

Object-Relational Mapping

If object models aren't the same as relational models (or some other data models that we might be using), some mechanism is needed by which data can be translated from the Data Storage and Management layer up into the object-oriented Business Logic layer.

> This is a well-known issue within the object-oriented community. It is commonly referred to as the *impedance mismatch problem*, and one of the best discussions of it can be found in David Taylor's book, Object-Oriented Technology: A Manager's Guide (Addison-Wesley, 1991).

Several object-relational mapping (ORM) products exist for the .NET platform from various vendors, including Microsoft. In truth, most ORM tools have difficulty working against object models defined using behavioral object-oriented design. Unfortunately, most of the ORM tools tend to create entity models that are closely modeled after the database, rather than true behavioral business objects. They create a data-centric representation of the business data and perhaps wrap it with business logic.

The differences between such a data-centric object model and what I am proposing in this book are subtle but important. Responsibility-driven object modeling creates objects that are focused on the object's behavior, not on the data it contains. The fact that objects contain data is merely a side effect of implementing behavior; the data is not the identity of the object. Most ORM tools, by contrast, create objects based around the data, with the *behavior* being a side effect of the data in the object.

> It is important to understand that CSLA .NET is not an ORM. It does not manage object-relational mapping, but you can use an ORM tool along with CSLA .NET to get

many ORM benefits, while leveraging the abilities of CSLA .NET to help you build a powerful, behavioral business layer.

Beyond the philosophical differences, the wide variety of mappings that you might need, and the potential for business logic driving variations in the mapping from object to object, make it virtually impossible to create a generic ORM product that can meet everyone's needs.

Consider the `CustomerEdit` object example discussed earlier. Although the customer data may come from one database, it is totally realistic to consider that some data may come from SQL Server while other data comes through screen-scraping a mainframe screen. It's also quite possible that the business logic will dictate that some of the data is updated in some cases, but not in others. Issues like these are virtually impossible to solve in a generic sense, and so solutions almost always revolve around custom code. The most a typical ORM tool can do is provide support for simple cases, in which objects are updated to and from standard, supported, relational data stores. At most, they'll provide hooks by which you can customize their behavior. Rather than trying to build a generic ORM product as part of this book, I'll aim for a much more attainable goal.

CSLA .NET defines a standard set of five methods for creating, retrieving, updating, and deleting objects: Create, Fetch, Update, Delete and Execute. Business developers will implement these methods to work with the underlying Data Access layer, which is implemented using the ADO.NET Entity Framework, raw ADO.NET, the XML support in .NET, XML services, or any other technology required to accomplish the task. In fact, if you have an ORM (or some other generic data access) product, you'll often be able to invoke that tool from these five methods just as easily as using ADO.NET directly.

The point is that CSLA .NET simplifies object persistence to the point at which all developers need to do is implement these five methods in order to retrieve or update data. This places no restrictions on the object's ability to work with data, and provides a standardized persistence and mapping mechanism for all objects.

Preserving Encapsulation

As I noted at the beginning of the chapter, one of my key goals is to design this framework to provide powerful features while following the key object-oriented concepts, including *encapsulation*.

Encapsulation is the idea that all the logic and data pertaining to a given business entity is held within the object that represents that entity. Of course, there are various ways in which one can interpret the idea of encapsulation—nothing is ever simple!

One approach is to encapsulate business data and logic in the business object, and then encapsulate data access and ORM behavior in some other object: a persistence object. This provides a nice separation between the business logic and data access, and encapsulates both types of behavior, as shown in Figure 30.

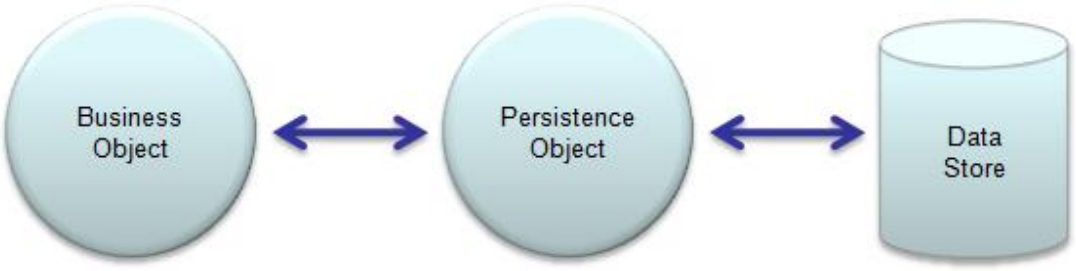

Figure 30. Separation of ORM logic into a persistence object

Although there are certainly some advantages to this approach, there are drawbacks, too. The most notable of these is that it can be challenging to efficiently get the data from the persistence object into or out of the business object. For the persistence object to load data into the business object, it must be able to bypass business and validation processing in the business object, and somehow load raw data into it directly. If the persistence object tries to load data into the object using the object's public properties, you'll run into a series of issues:

- The data already in the database is presumed valid, so a lot of processing time is wasted unnecessarily revalidating data. This can lead to a serious performance problem when loading a large group of objects.
- There's no way to load read-only property values. Objects often have read-only properties for things such as the primary key of the data, and such data obviously must be loaded into the object, but it can't be loaded via the normal interface (if that interface is properly designed).
- Sometimes properties are interdependent due to business rules, which means that some properties must be loaded before others or errors will result. The persistence object would need to know about all these conditions so that it could load the right properties first. The result is that the persistence object would become *very* complex, and changes to the business object could easily break the persistence object.

On the other hand, having the persistence object load raw data into the business object breaks encapsulation in a big way, because one object ends up directly tampering with the internal fields of another. This could be implemented using reflection, or by designing the business object to expose its private fields for manipulation. But the former is slow, and the latter is just plain bad object design: it allows the UI developer (or any other code) to manipulate these fields, too. That's almost asking for the abuse of the objects, which will invariably lead to code that's impossible to maintain.

Regardless, this is a popular approach to solving this problem, and CSLA .NET does support creation of a persistence object, called an `ObjectFactory`, that can break encapsulation and directly manipulate the private state of business objects.

But what's needed is a workable compromise, where the data access code is in one object, while the code to load the business object's fields is in the business object itself. This can be accomplished by creating a separate data container class, often called a data transfer object (DTO). This DTO acts as a temporary data container to carry the data from the persistence object into the business object, or from the business object into the persistence object.

Usually your DTO types will be defined in a separate data access interface assembly, as shown in Figure 31.

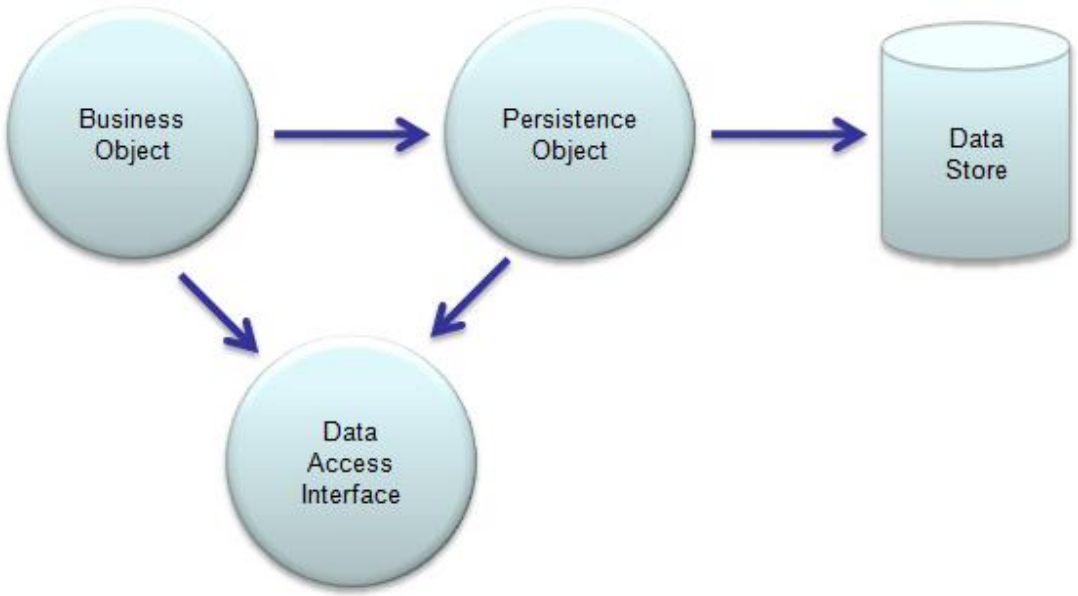

Figure 31. Business object interacting with a Data Access layer

This is a nice compromise, because it allows the business object to completely manage its own fields, because it contains the code to copy its data into and out of the DTO. At the same time, the data access logic is totally separated into a persistence object, which knows nothing about the business class or its private members.

There are several ways to implement such a model, including the use of ADO.NET Entity Framework or raw ADO.NET.

The ADO.NET Entity Framework provides abstraction and ORM capabilities. Using the tooling built into Visual Studio, it is relatively easy to create an entity model by dragging and dropping database tables onto a designer. You can then write LINQ queries against the entity model to retrieve or update data.

The raw ADO.NET approach has the benefit of providing optimal performance, because all other data access technologies, such as ADO.NET Entity Framework, are built on top of ADO.NET. Although raw ADO.NET may offer better performance, it does require that you more directly interact with the database to set up connections, commands and work with `IDataReader` components to read data.

The examples in this book series will use Data Access layers created using either the ADO.NET Entity Framework or raw ADO.NET, along with mock databases creating using LINQ to Objects.

Supporting Physical N-Tier Models

I've portrayed the business domain object as encapsulating behavior, along with the data necessary to implement that behavior. Of course, some behaviors should run on the client to efficiently interact with the user, whereas other behaviors should run on an application server to efficiently

interact with server-side resources. The question that remains, then, is how to support physical n-tier models if the client-oriented and server-oriented behaviors are encapsulated in *one* object?

Client-oriented behaviors almost always focus on interaction with the UI in order to set, retrieve, and manipulate the values of an object. This means those behaviors require many properties, methods and events—a very fine-grained interface with which the UI can interact. Almost by definition, this type of object *must* run in the same process as the UI code itself, either on the client device with WebAssembly, Xamarin, WPF, UWP, or Windows Forms, or on the web server with Razor Pages, ASP.NET MVC, or Web Forms.

Conversely, server-oriented behaviors typically involve few methods: create, fetch, update, delete. They must run on a machine where they can establish a physical connection to the data store. Sometimes, this is the client device or web server, but often it means running on a physically separate application server.

This point of apparent conflict is where the concept of *mobile objects* enters the picture. It's possible to pass a business object from an application server to the client device, work with the object, and then pass the object back to the application server so that it can store its data in the database. To do this, there needs to be some black-box component running as a service on the application server with which the client can interact. This black-box component does little more than accept the object from the client, and then call methods on the object to retrieve or update data as required. But the object itself does all the real work. Figure 32 illustrates this concept, showing how the *same physical business object* can be passed from application server to client, and vice versa, via a generic router object that's running on the application server.

In Chapter 1, I discussed anchored and mobile objects. In this model, the business object is mobile, meaning that it can be passed around the network by value. The router object is anchored, meaning that it will always run on the machine where it's created.

In CSLA .NET this router object is called the *data portal*. It acts as a portal for all data access for all the objects. Business objects interact with this portal in order to retrieve default values (create), fetch data (read), update or insert data (update), remove data (delete), or execute server code (execute). This means that the data portal will provide a standardized mechanism by which objects can perform all server interactions.

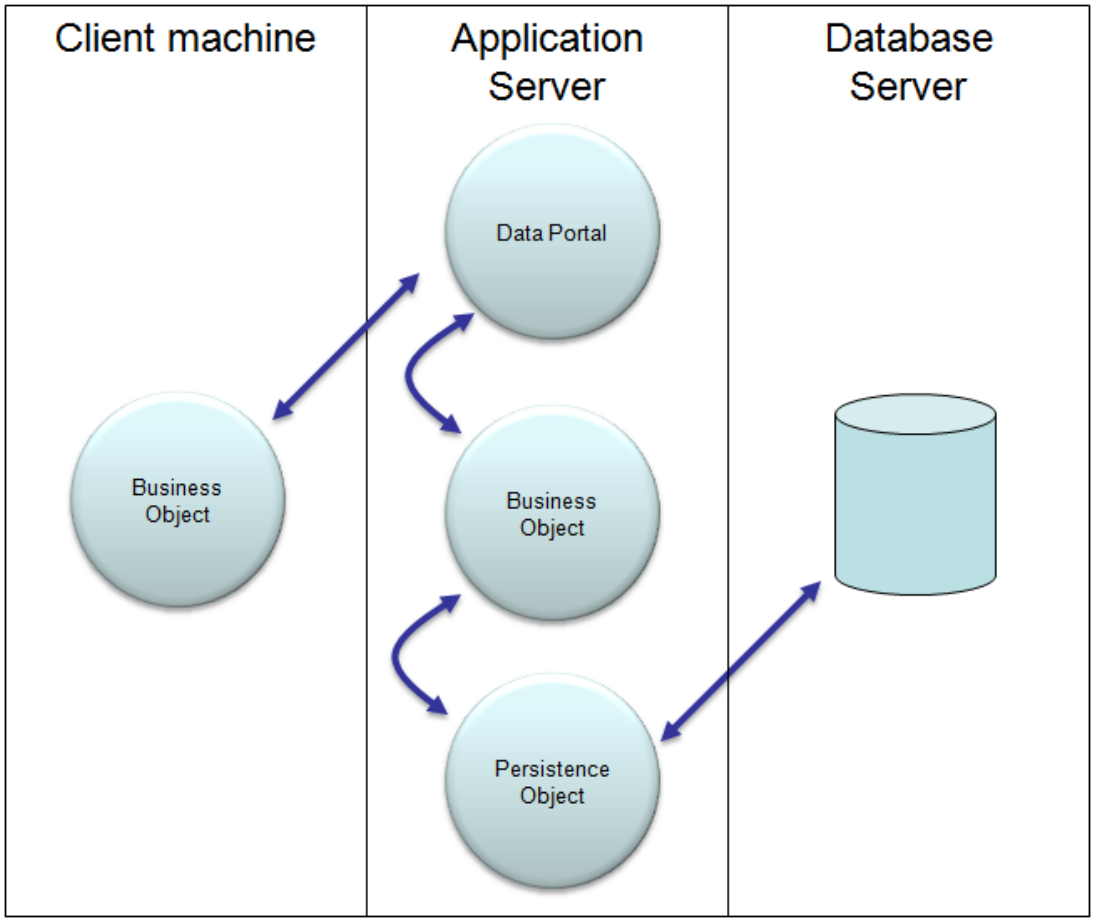

Figure 32. Passing a business object to and from the application server

Notice how the data portal concept abstracts the use of WCF, and so this code is far simpler than the WCF code used earlier in the chapter.

On the server, the data portal does one of two things, depending on your configuration. You can have the data portal invoke methods implemented in the business class itself, allowing the business object to interact with the Data Access layer. Or you can have the data portal create a factory object and invoke methods on that factory object, in which case the factory object is responsible for interacting with both the business object and Data Access layer.

In the former case, where the data portal invokes methods directly on the business class, those methods will look something like this:

```
private void DataPortal_Fetch(string customerId)
{
  // Code to invoke the data access layer and to
  // load the object's fields with data goes here
}
```

Or, if you've decided to use a factory object, the method in that object will look something like this:

```
public CustomerEdit Fetch(string customerId)
{
  var result = new CustomerEdit();
  // Code to invoke the data access layer and to
  // load the object's fields with data goes here
  return result;
}
```

The Interface Control layer won't know (or need to know) how any of this works, so in order to create a `CustomerEdit` object, the interface logic will write code along these lines:

```
var cust = await Csla.DataPortal.CreateAsync<CustomerEdit>("ABC");
```

CSLA .NET, and specifically the data portal, will take care of all the rest of the work, including figuring out whether the data access code should run on the client device or on an application server.

By implementing your business classes correctly and using the data portal, you can switch between having the data access code running on the client device and placing it on a separate application server by changing a configuration file setting. The ability to change between 2-, 3-, and 4-tier physical configurations with no changes to code is a powerful, valuable feature.

Supporting N-Tier via MobileFormatter

As I mentioned earlier in this book, CSLA provides its own serializer: `MobileFormatter`. Originally this serializer was created to overcome the fact that the "standard" .NET serializers (`BinaryFormatter` and `NetDataContractSerializer`) weren't available on some of the newer .NET implementations. Subsequently, `MobileFormatter` has been enhanced to be faster, create a smaller payload over the network, and to be more secure than the .NET serializers. As a result it is the default for all .NET implementations when using CSLA.

To avoid the reflection and security concerns, `MobileFormatter` requires that each object participate in its serialization and deserialization, and this is handled through an `IMobileObject` interface. In other words, `MobileFormatter` can serialize any object that implements `IMobileObject`.

> If you implement a custom type without subclassing one of the standard CSLA .NET base classes, and you want those objects to flow through the data portal, you will need to implement `IMobileObject` in those classes.

Rather than forcing every class to directly implement `IMobileObject`, CSLA .NET includes a set of base classes that implement the interface, including `MobileObject`, `MobileList`, and `MobileDictionary`. All the serializable types in CSLA .NET directly or indirectly implement `IMobileObject`, and so can be serialized using the .NET serializers, or `MobileFormatter`.

Custom Authentication

Application security is often a challenging issue. Applications need to be able to authenticate the user, which means that they need to verify the user's identity. The result of authentication is not only that the application knows the identity of the user, but that the application has access to the user's role membership and possibly other information about the user. Collectively, I'll refer to this

as the user's profile data. This profile data can be used by the application for various purposes; most notably, authorization.

CSLA .NET directly supports integrated security. This means that you can use objects within the framework to determine the user's Windows identity and any domain or Active Directory (AD) groups to which they belong. In some organizations, this is enough: all the users of the organization's applications are in Active Directory, and by having them log in to a device or a website using integrated security, the applications can determine the user's identity and roles (groups).

In other organizations, applications are used by at least some users who are *not* part of the organization's NT domain or AD. They may not even be members of the organization in question. This is often the case with web and mobile applications, but it's surprisingly common with Windows applications as well. In these cases, you *can't* rely on Windows integrated security for authentication and authorization.

One common solution is to use claims-based authorization, often with OAUTH-based authentication. And some applications create their own authentication model, often storing user credentials and information in a database, an LDAP server or in other locations.

Regardless of the authentication model, the ideal security solution provides user profile and role or claims information not only to server-side code, but also to the code on the client. Rather than allowing the user to attempt to perform operations that will generate errors due to security at some later time, the UI should gray out the options, or not display them at all. This requires that the developer have consistent access to the user's identity and profile at all layers of the application, including the Interface Control, Business, and Data Access layers.

Remember that the layers of an application may be deployed across multiple physical tiers. Due to this fact, there must be a way of transferring the user's identity information across tier boundaries. This is often called *impersonation*.

Impersonation works differently depending on your authentication model. Windows integrated security enables impersonation within limits defined by the configuration of your network and AD domain. ASP.NET membership and custom authentication don't automatically provide any sort of impersonation, but it is possible to implement your own, by passing user information between application layers as necessary.

The CSLA .NET framework provides support for both Windows integrated security, claims-based, and custom authentication. CSLA .NET relies on Windows itself to handle impersonation when using Windows integrated or AD security and handles impersonation itself when using custom authentication.

Extensibility

Rarely does a framework directly meet every requirement of every application. Therefore, it is important that frameworks enable extensibility and customization. Some forms of common customization are relatively easy, whereas other types of customization can be complex and require extensive knowledge about how CSLA .NET functions behind the scenes.

One of my primary goals with CSLA .NET is to simplify the creation of a business layer that encapsulates the business logic for an application. To this end, the framework uses an inheritance-

based model where a developer inherits from a base class to create different types of business class. This is not unlike inheriting from Page, Form or Window to create web pages, Windows Forms or WPF windows; the simple act of subclassing a base class grants your new type an amazing set of functionality.

The base classes have numerous methods and properties that are `virtual`, with the intent that they can be overridden to extend or customize specific areas of behavior. For example, you might override the `MarkNew` method to change whether a new object is considered changed or not by default.

At the same time, inheritance can be quite limiting, because there's a tightly coupled relationship between the base class and any subclasses. Like many frameworks, CSLA .NET often provides an alternative where you can choose to implement interfaces instead of subclassing a base class. This is *much* more complex, because you'll often need to reimplement some or all the functionality that would have been provided by the base class, but it is extremely flexible. For example, you might use this ability to create entire new base classes for use by your application.

Some parts of the framework use indirection and configuration to enable extensibility. For example, you can create your own proxy and host so that the data portal to communicate between client and server. Your proxy and host will implement specific interfaces, and you'll use configuration to tell the data portal to load your types instead of the standard proxy/host types.

Extensibility is important to the success of any framework and is particularly important for a framework with the breadth of CSLA .NET. Each area of extensibility is focused on specific business and technology requirements, and throughout this book series I'll discuss extensibility points as they relate to the topic being discussed, whether that be business class creation, the data portal, data access, or interacting with various presentation technologies such as Xamarin or ASP.NET.

Framework Design

A comprehensive framework can be a large and complex entity. There are usually many classes that go into the construction of a framework, even though the end users of the framework—the business developers—only use a few of those classes directly. CSLA .NET accomplishes the goals I've discussed, along with enabling the basic creation of object-oriented n-tier business applications.

The CSLA .NET framework contains a lot of classes and types, which can be overwhelming if taken as a whole. Fortunately, it can be broken down into smaller units of functionality to better understand how each part works. Specifically, the framework can be divided into the following functional groups:

- Business object creation
- Data binding support
- N-level undo functionality
- Validation and business rules
- A data portal enabling various physical configurations
- Transactional and non-transactional data access
- Authentication and authorization
- Helper types and classes

For each functional group, I'll focus on a subset of the overall class diagram, breaking it down into more digestible pieces.

Business Object Creation

First, it's important to recognize that the key classes in the framework are those that business developers will use as they create business objects, but that these are a small subset of what's available. In fact, many of the framework classes are never used *directly* by business developers.

Obviously, the business developer may periodically interact with other classes as well, but these are the ones that will be at the center of most activity. Classes or methods that the business developer shouldn't have access to are scoped to prevent accidental use.

As I mentioned earlier in this chapter, the most typical way you'll use CSLA .NET is to inherit from a framework base class to create your business classes. To this end, CSLA .NET has a set of base classes designed to help you create different types of business class. Table 4 summarizes each base class and its intended purpose.

Class	Purpose
`BusinessBase<T>`	Inherit from this class to create a single editable business object such as `CustomerEdit`, `OrderEdit`, or `OrderLineItemEdit`.
`BusinessListBase<T,C>`	Inherit from this class to create an editable collection of business objects such as `PaymentTerms` or `OrderLineItems`.
`BusinessBindingListBase<T,C>`	Inherit from this class to create an editable collection of business objects for use in Windows Forms applications.
`DynamicListBase<C>`	Inherit from this class to implement a collection of business objects, where changes to each object are committed automatically as the user moves from object to object (typically in a data bound grid control).
`DynamicBindingListBase<C>`	Inherit from this class to implement a collection of business objects, for use in Windows Forms applications, where changes to each object are committed automatically as the

	user moves from object to object (typically in a data bound grid control).
`CommandBase<T>`	Inherit from this class to implement a command that should run on the application server, such as implementation of a `Customer.Exists` or an `Order.ShipOrder` command.
`ReadOnlyBase<T>`	Inherit from this class to create a single read-only business object such as `OrderInfo` or `ProductStatus`.
`ReadOnlyListBase<T,C>`	Inherit from this class to create a read-only collection of objects such as `CustomerList` or `OrderList`.
`ReadOnlyBindingListBase<T,C>`	Inherit from this class to create a read-only collection of objects for use in Windows Forms applications.
`NameValueListBase<K,V>`	Inherit from this class to create a read-only collection of key/value pairs (typically for populating drop-down list controls) such as `PaymentTermsCodes` or `CustomerCategories`.
`CriteriaBase<T>`	Inherit from this class to create an object that contains criteria values passed from the client to the server to create, retrieve or delete another object.
`CslaPrincipal`	Directly use, or inherit from this class to create a custom authentication principal object.
`CslaIdentity` and `CslaIdentity<T>`	Directly use, or inherit from this class to create a custom authentication identity object.

Table 4. Business Framework Base classes

These base classes support a set of object *stereotypes*. A stereotype is a broad grouping of objects with similar behaviors or roles. The supported stereotypes are listed in Table 5.

Stereotype	Description	Base Class
Editable root	Object containing read-write properties; object can be retrieved/stored directly to database.	`BusinessBase<T>`
Editable child	Object containing read-write properties; object is contained within another object and can *not* be retrieved/stored directly to database.	`BusinessBase<T>`
Editable root list	List object containing editable child objects; list can be retrieved/stored directly to database.	`BusinessListBase<T,C>` `BusinessBindingListBase<T,C>`
Editable child list	List object containing editable child objects; list is contained within another	`BusinessListBase<T,C>` `BusinessBindingListBase<T,C>`

	object and can *not* be retrieved/stored directly to database.	
Dynamic root list	List object containing editable root objects; list is retrieved directly from database.	`DynamicListBase<C>` `DynamicBindingListBase<C>`
Command	Object that executes a command on the application server and reports back with the results.	`CommandBase<T>`
Read-only root	Object containing read-only properties; object can be retrieved directly from database.	`ReadOnlyBase<T>`
Read-only child	Object containing read-only properties; object is contained within another object and can *not* be retrieved directly from database.	`ReadOnlyBase<T>`
Read-only root list	List containing read-only child objects; list can be retrieved directly from database.	`ReadOnlyListBase<T,C>` `ReadOnlyBindingListBase<T,C>`
Read-only child list	List containing read-only child objects; list is contained within another object and can *not* be retrieved directly from database.	`ReadOnlyListBase<T,C>` `ReadOnlyBindingListBase<T,C>`
Name/value list	List object containing read-only name/value objects.	`NameValueListBase<K,V>`

Table 5. Supported object stereotypes

I discuss each stereotype in detail in the *Using CSLA 2019: Creating Business Objects* book.

Data Binding Support

As I discussed earlier in the chapter, the .NET data binding infrastructure directly supports the concept of data binding to objects and collections. Additionally, an object can provide more complete behaviors by implementing a few interfaces in the framework base classes. Table 6 lists the interfaces and their purpose.

Interface	Purpose
`IBindingList`	Defines data binding behaviors for collections, including change notification, sorting, and filtering (implemented by `BindingList<T>`); used by Windows Forms
`ICancelAddNew`	Defines data binding behaviors for collections to allow data binding to cancel addition of a new child object (implemented by `BindingList<T>`); used by Windows Forms

IRaiseItemChangedEvents	Indicates that a collection object will raise a ListChanged event to indicate that one of its child objects has raised a PropertyChanged event (implemented by BindingList<T>)
INotifyCollectionChanged	Defines a CollectionChanged event to be raised by a list when the list or its items have changed (implemented by ObservableCollection<T>)
IEditableObject	Defines single-level undo behavior for a business object, allowing the object to behave properly with in-place editing in a DataGridView
INotifyPropertyChanged	Defines an event allowing an object to notify data binding when a property has been changed
INotifyPropertyChanging	Defines an event allowing an object to notify listeners when a property is about to be changed
IDataErrorInfo	Defines properties used by the DataGridView and ErrorProvider controls to automatically show descriptions of broken validation rules within the object
INotifyDataErrorInfo	Defines properties and events used by data binding to automatically show descriptions of broken validation rules within the object; not supported by Windows Forms or Web Forms

Table 6. Data Binding Interfaces

These data binding interfaces are most important for editable objects, objects with public read-write properties. This means the focus of data binding support is in the BusinessBase, BusinessListBase, BusinessBindingListBase, DynamicListBase and DynamicBindingListBase classes.

To fully interact with data binding across all presentation technologies, BusinessBase implements INotifyPropertyChanged, IEditableObject and IDataErrorInfo. On supported .NET platforms, it also implements INotifyDataErrorInfo. The result is that editable business objects support any validation display provided by the presentation technology, as well as in-place editing in datagrid controls, and of course basic property change notification behavior.

BusinessListBase is a subclass of ObservableCollection<T> from the System.ComponentModel. ObjectModel namespace, so that means it implements INotifyCollectionChanged. This means that BusinessListBase should be used to create editable collections for most presentation technologies other than Windows Forms. It is also the case that some WPF datagrid controls won't work properly with a subclass of ObservableCollection.

BusinessBindingListBase is a subclass of the older BindingList<T> from the System.ComponentModel namespace. The BindingList base class was introduced in .NET 2.0 to simplify the creation of collections for use with Windows Forms. Any editable collections you create that will be bound to a Windows Forms interface should inherit from BusinessBindingListBase.

> You need to choose the appropriate collection base class depending on whether you are creating a Windows Forms interface or using any other presentation technology.

> This is the result of Microsoft switching from `BindingList<T>` to `ObservableCollection<T>` in .NET Framework 3.5 and WPF.

`DynamicListBase` also a subclass of `ObservableCollection<T>`, as are `ReadOnlyListBase` and `NameValueListBase`.

`DynamicBindingListBase` and `ReadOnlyBindingListBase` inherit from `BindingList<T>`, and so provide support for Windows Forms.

All the "BindingListBase" types, such as `BusinessBindingListBase`, do exist in all versions of CSLA .NET, but the `BindingList` base type does not exist on those platforms. So, on all platforms other than full .NET, these "BindingListBase" types are the same as their `ObservableCollection` counterparts. Outside of full .NET there's no difference between the `BusinessListBase` and `BusinessBindingListBase` classes. As a rule you should use `BusinessListBase` for all development unless you are supporting a Windows Forms UI with the same business library.

The key thing to understand is that the CSLA .NET base classes implement the data binding interfaces defined by each .NET UI technology, so your business types automatically gain all those behaviors with no effort on your part. Not only do your objects gain basic property change notification (via `INotifyPropertyChanged`), but they gain support for in-place datagrid editing, and validation results display as supported by each presentation technology.

N-Level Undo Functionality

N-level undo is the ability of a business object to take a snapshot of its state, and to revert to that saved state if requested. At a basic level, this functionality is required to support in-place editing in a datagrid control, where the user can edit values in a row and then press the ESC key to undo those changes. In more complex scenarios, the user may edit many properties in an object, and child objects as well, and then click a Cancel button. In that case all the changed properties in all the objects (the object graph) can be undone, as each object reverts to its previous state.

At first glance, it might appear that you could use .NET serialization to implement undo functionality: what easier way to take a snapshot of an object's state than to serialize it into a byte stream? Unfortunately, this isn't as easy as it might sound, at least when it comes to restoring the object's state.

Taking a snapshot of a `Serializable` object is easy, and can be done with code like this:

```
[Serializable]
public class Customer
{
  public byte[] Snapshot()
  {
    using (var buffer = new MemoryStream())
    {
      var formatter = new BinaryFormatter();

      formatter.Serialize(buffer, this);
      buffer.Position = 0;
      return buffer.ToArray();
    }
  }
}
```

This converts the object into a byte stream, returning that byte stream as an array of type `byte`. That part is easy—it's the restoration that's tricky. Suppose that the user now wants to undo the changes, requiring that the byte stream be restored back into the object. The code that deserializes a byte stream looks like this:

```
[Serializable]
public class Customer
{
  public Customer Deserialize(byte[] state)
  {
    using (var buffer = new MemoryStream(state))
    {
      var formatter = new BinaryFormatter();

      return (Customer)formatter.Deserialize(buffer);
    }
  }
}
```

Notice that this function returns a *new customer object*. It doesn't restore the existing object's state; it creates a new object. Somehow, you would have to tell any and all code that has a reference to the existing object to use this new object. In some cases, that might be easy to do, but it isn't always trivial. In complex applications, it's hard to guarantee that other code elsewhere in the application doesn't have a reference to the original object—and if you don't somehow get that code to update its reference to this new object, it will continue to use the old one.

What's needed is some way to restore the object's state *in place*, so that all references to the current object remain valid, but the object's state is restored. CSLA .NET provides this functionality through a set of types implemented in the framework. Table 7 lists the primary types involved in this implementation.

Type	Purpose
UndoableBase	Base class for BusinessBase that implements the ability to take and restore a snapshot of a single business object
IUndoableObject	Interface implemented by objects that wish to participate in the undo behaviors. This interface is used by UndoableBase, allowing a parent object to interact with its child objects
ISupportUndo	Interface implemented by objects that support undo. This interface is designed for use by UI framework authors, or other code that needs to tell objects to take or restore snapshots of their state
UndoException	Exception thrown when undo is unable to perform the requested action

Table 7. CSLA .NET types supporting undo functionality

The BusinessBase class inherits from UndoableBase, and thereby gains n-level undo capabilities. Because all business objects inherit from BusinessBase, they too gain n-level undo. Ultimately, the n-level undo capabilities are exposed to the business object and to UI developers via three methods:

- BeginEdit tells the object to take a snapshot of its current state, in preparation for being edited. Each time BeginEdit is called, a new snapshot is taken, allowing the state of the object to be trapped at various points during its life. The snapshot will be kept in memory so the data can be easily restored to the object if CancelEdit is called.

- `CancelEdit` tells the object to restore the object to the most recent snapshot. This effectively performs an undo operation, reversing one level of changes. If `CancelEdit` is called the same number of times as `BeginEdit`, the object will be restored to its original state.
- `ApplyEdit` tells the object to discard the most recent snapshot, leaving the object's current state untouched. It accepts the most recent changes to the object. If `ApplyEdit` is called the same number of times as `BeginEdit`, all the snapshots will be discarded, making any changes to the object's state permanent.

These methods are exposed as `public` members of `BusinessBase`, `BusinessListBase` and `BusinessBindingListBase`. They are also exposed through the `ISupportUndo` interface.

You should understand that each `BeginEdit` call takes a new snapshot of the state of the object, along with the states of all child objects. Every `BeginEdit` call *must* have a corresponding `CancelEdit` or `ApplyEdit` call.

Every object has the concept of an *edit level*, which is the number of unresolved `BeginEdit` calls that have been made on the object. Another way to look at it, is that the edit level represents the number of snapshots of the object's state that are being held in memory.

You should also realize that n-level undo is used by CSLA .NET to implement the `IEditableObject` data binding interface. That interface defines three similar methods:

- `BeginEdit` tells the object to take a snapshot of its current state, but it does not take a snapshot of any child object state, and all calls after the first call are ignored
- `CancelEdit` tells the object to restore the object to the most recent snapshot, but it does not restore any child object state, and all calls after the first call are ignored
- `EndEdit` tells the object to discard the most recent snapshot, leaving the object's current state unchanged, but it does not discard any child object snapshots, and all calls after the first call are ignored

The slightly different behavior of the `IEditableObject` methods is required, because data binding imposes specific requirements on how `IEditableObject` is to be implemented. Deviation from those requirements results in data binding working incorrectly, typically when doing in-place editing of rows in datagrid controls.

The most important thing to understand, is that `IEditableObject` will only ever raise or lower the object's edit level by 1. After `IEditableObject.BeginEdit` has been called, all subsequent `BeginEdit` calls are ignored until either `CancelEdit` or `EndEdit` have been called.

NotUndoableAttribute

The final concept to discuss regarding n-level undo is the idea that some data might not be subject to being in a snapshot. Taking a snapshot of an object's data takes time and consumes memory—if the object includes read-only values, there's no reason to take a snapshot of them. Because the values can't be changed, there's no benefit in restoring them to the same value in the course of an undo operation.

To accommodate this scenario, the framework includes a custom attribute named NotUndoableAttribute, which you can apply to fields within your business classes, as follows:

```
[NotUndoable]
private string _readonlyData;
```

The code in UndoableBase ignores any fields marked with this attribute as the snapshot is created or restored, so the field will always retain its value regardless of any calls to BeginEdit, CancelEdit, or ApplyEdit on the object.

You should be aware that the n-level undo implementation doesn't handle circular references, so if you have a field that references another object in a way that would cause a circular reference you must mark the field as NotUndoable to break the circle.

Business, Validation and Authorization Rules

Recall that one of the goals of CSLA .NET is to simplify and standardize the creation of business rules, including validation and authorization rules. The term "business rules" can mean different things to people. I think you can generally think of business rules as including several types of rule, including:

- Validation of one or more property values on an object
- Validation of values across multiple objects in an object graph
- Authorizing read or write access to a property
- Authorizing execution of a method
- Calculating one or more values
- Performing algorithmic processing
- Invoking remote processing (executing code on a server)

As you can see, this definition of business rules covers most of the business logic you'd expect to find in an application. This fits naturally with the overall philosophy of CSLA .NET, which is that the business layer should encapsulate the business logic, including calculations, algorithmic processing, validation, and authorization.

In this chapter I'll provide a high-level overview of the rules engine in CSLA .NET. I'll go into much more detail about the subsystem in the *Using CSLA 2019: Creating Business Objects* book.

CSLA .NET provides a consistent approach to implement business rules; and to then attach those rules to properties of your business objects. Keeping the rule implementations clearly separate from any property implementations means that all your properties can be implemented in a consistent manner:

```
public static PropertyInfo<string> CustomerNameProperty =
  RegisterProperty<string>(p => p.CustomerName);
public string CustomerName
{
  get { return GetProperty(CustomerNameProperty); }
  set { SetProperty(CustomerNameProperty, value); }
}
```

You'll have to trust me when I say that this property fully implements authorization, validation and will automatically run any associated business processing that should occur when the property value is changed. With only a couple possible variations, all business object properties look exactly like this one.

So how does this property implement all those business rules? Again, the business rules are implemented separately, and are then attached to properties. The secret lies in the `GetProperty` and `SetProperty` methods. They automatically invoke any rules attached to this property.

You can attach rules in a couple different ways, depending on how you implement the rules. CSLA .NET allows you to register rules through code, and if you are using the `System.ComponentModel.DataAnnotations` namespace you can define rules using its attributed-based approach. When using CSLA .NET rules, the rules are attached to properties by overriding the `AddBusinessRules` method in an editable business object. When using `DataAnnotations` attributes, the attributes are applied directly to the property. The following example shows both techniques:

```
public static PropertyInfo<string> CustomerNameProperty =
  RegisterProperty<string>(p => p.CustomerName);
[Required]
public string CustomerName
{
  get { return GetProperty(CustomerNameProperty); }
  set { SetProperty(CustomerNameProperty, value); }
}

protected override void AddBusinessRules()
{
  base.AddBusinessRules();
  BusinessRules.AddRule(new Csla.Rules.CommonRules.MaxLength(CustomerNameProperty, 30));
  BusinessRules.AddRule(new Csla.Rules.CommonRules.IsInRole(
    Csla.Rules.AuthorizationActions.WriteProperty, CustomerNameProperty, "Admin"));
}
```

The `Required` attribute is a `DataAnnotations` attribute that indicates this is a required value. Also, inside the `AddBusinessRules` override you can see where `AddRule` is called to add a `MaxLength` rule to the `CustomerName` property.

For simple validation rules these techniques are essentially interchangeable. However, CSLA .NET rules are much more powerful than the `DataAnnotations` attributes, enabling not onlysimple validation, but also:

- Rules that modify property values
- Rule severity (`Error`, `Warning`, `Information`)
- Rule priority (control order of rule execution)
- Asynchronous rules (rules that invoke asynchronous methods or actions)
- Invocation of external rule engines or workflows
- Authorization rules
- Rules that invoke other rules

Additionally, you can see that an `IsInRole` rule is attached to the `CustomerName` property, specifying that only people in the `Admin` role can write to this property.

So not only does this property have validation rules, it also has authorization rules. And the same technique is used to attach rules that perform calculations or algorithmic processing.

Common Rules

The CSLA .NET framework includes some pre-built common rules in the `Csla.Rules.CommonRules` type. These common rules are listed in Table 8.

Rule	Description
`Required`	Indicates that a `string` property is required
`MaxLength`	Indicates that a `string` property has a maximum length
`MinLength`	Indicates that a `string` property has a minimum length
`MinValue`	Specifies a minimum value for any `IComparable` property value
`MaxValue`	Specifies a maximum value for any `IComparable` property value
`RegExMatch`	Requires that a property value match a specific regular expression
`InfoMessage`	Display an informational message about a property
`Lambda`	Execute a lambda expression provided by the developer
`Dependency`	Indicates that when rules for one property are run, rules for a dependent property should also be run
`HasPermission`	Determines if the current user's claims meet the specified authorization requirements
`IsInRole`	Requires that the current user is in one or more of the specified roles
`IsNotInRole`	Requires that the current user is not in any of the specified roles

Table 8. Common rules in Csla.Rules.CommonRules

The `System.ComponentModel.DataAnnotations` namespace also includes some pre-built validation attributes as listed in Table 9.

Attribute	Description
`CustomValidation`	Executes a specified method that implements the validation rule
`Range`	Limits a value to a specified range
`RegularExpression`	Requires that a property value match a specific regular expression
`Required`	Indicates that a `string` property is required
`StringLength`	Specifies the minimum and maximum length for a `string` property

Table 9. Standard DataAnnotations validation attributes

As you can see, there's some overlap between the `DataAnnotations` attributes and the CSLA .NET `CommonRules` types. In general, you can use either approach interchangeably, but I recommend using the `DataAnnotations` attributes where possible, because these standard attributes are sometimes used by some UI technologies (most notably ASP.NET MVC) to provide extra value.

You should also recognize that these are just the pre-built rules. You can create your own business, validation and authorization rules to meet your specific application requirements.

Using Validation Results

The results of validation rules are maintained by each editable business object as a list of broken rules. This enables CSLA .NET to efficiently implement the `IDataErrorInfo` and `INotifyDataErrorInfo` data binding interfaces, but also allows you to write code that leverages this information.

Every editable object exposes an `IsValid` property. If this property is `true` then you know there are no broken validation rules for the object or any of its child objects. And if it is `false` you know that one or more validation rules are broken in this object or a child object. There's also an `IsSavable` property, which only returns `true` if `IsValid` is true, and the object has been changed, and the user is authorized to save changes to the object.

Objects also expose an `IsSelfValid` property which returns `true` or `false` by looking only at *this specific object's broken rules*, ignoring the state of any child objects.

Business objects that subclass BusinessBase also have a `BrokenRulesCollection` property that returns a collection of all broken rules for all properties of the object. You can use this information in many ways. For example, this query gets a list of all `Warning` severity rules:

```
var warnings = from r in this.BrokenRulesCollection
               where r.Severity == Csla.Rules.RuleSeverity.Warning
               select r;
```

CSLA .NET also allows you to get a consolidated list of broken rules for an object and all of its child objects through the `BrokenRules.GetAllBrokenRules` method.

Having access to the full set of broken validation rules allows you to display or use this information as required by your application.

Using Authorization Results

The results of authorization rules are listed in Table 10.

Rule	Result
Property read	Attempt to read the property results in the default property value being returned if user isn't authorized to read the property
Property write	Attempt to write to the property throws a `SecurityException` if user isn't authorized to write to the property
Execute method	Attempt to invoke the method throws a `SecurityException` if user isn't authorized to execute the method

Create object	Data portal `Create` method throws a `SecurityException` if user isn't authorized to perform the action
Get object	Data portal `Fetch` method throws a `SecurityException` if user isn't authorized to perform the action
Update object	Data portal `Update` method throws a `SecurityException` if user isn't authorized to perform the action
Delete object	Data portal `Delete` method throws a `SecurityException` if user isn't authorized to perform the action

Table 10. Results of authorization rules

The only odd result in the list is for property read, where a default value is returned instead of an exception being thrown. You can override this default behavior, but the behavior exists to support normal data binding behaviors in UWP, Xamarin.Forms, WPF and Windows Forms.

Data binding has no concept of authorization rules; unfortunately this isn't something Microsoft has formalized in the .NET Framework. Although you may write UI code to hide or disable controls the user isn't authorized to see, data binding will usually still attempt to read the value from the business object, and there's no reasonable way to prevent that from happening. So if read operations throw an exception when an unauthorized user attempts to read a property value, that would crash the app, and there's often no workaround.

So in this case, instead of throwing an exception, a default value for the property is returned so the real value can't be displayed to the user. You should still implement UI code to hide or disable the UI control so the user understands they can't see the value.

Editable business objects expose methods you can use to determine whether a user is authorized to perform various actions. These methods are listed in Table 11.

Source	Method	Description
Business object	`CanReadProperty`	Gets a Boolean indicating whether the user is authorized to read the specified property
Business object	`CanWriteProperty`	Gets a Boolean indicating whether the user is authorized to change the specified property
Business object	`CanExecuteMethod`	Gets a Boolean indicating whether the user is authorized to execute the specified method
`Csla.Rules.BusinessRules`	`HasPermission`	Gets a Boolean indicating whether the user is authorized to create, get, update or delete instances of a business object type

Table 11. Methods exposing authorization results

These methods are all public, so you can call them from your UI code or anywhere else in your application where you need to determine the results of authorization rules. Usually you'll use these methods in the UI to enable, disable, or hide various UI controls based on whether the user has permission to do various things.

Creating Business Rules

Business rules are classes that implement the `IBusinessRule` interface in the `Csla.Rules` namespace. That interface requires that you implement several properties and an `Execute` method, which is where the rule is implemented. To simplify creation of typical rules, there's a `BusinessRule` base class you should subclass when creating a rule:

```
public class MyRule : Csla.Rules.BusinessRule
{
  protected override void Execute(Csla.Rules.RuleContext context)
  {
    // implement rule here
  }
}
```

This rule could be a validation rule, or it could perform calculations or algorithmic processing. The `Execute` method is passed a `RuleContext` parameter that contains rich information about the business object and the property to which this rule is attached.

If you are creating a simple validation rule, you can also choose to implement the rule as a subclass of the `ValidationAttribute` type in `System.ComponentModel.DataAnnotations`:

```
[AttributeUsage(AttributeTargets.Property)]
public class MyAnnotationRule : System.ComponentModel.DataAnnotations.ValidationAttribute
{
  protected override ValidationResult IsValid(object value, ValidationContext validationContext)
  {
    if (true) // implement real rule here
      return null;
    else
      return new ValidationResult("Property is invalid");
  }
}
```

In this case, your rule implementation goes in an override of the `IsValid` method. The `value` parameter provides the value to be validated, and the `ValidationContext` parameter provides other information, such as a reference to the business object.

I'll provide much more detail about creating business rules in the *Using CSLA 2019: Creating Business Objects* book.

Creating Authorization Rules

Authorization rules are slightly different from other business or validation rules, in that they return a Boolean result. The user either does or doesn't have permission to perform each action. Authorization rules are classes that implement the `IAuthorizationRule` interface from the `Csla.Rules` namespace. There's an `AuthorizationRule` base class that helps simplify the creation of rules:

```
public class MyAuthzRule : Csla.Rules.AuthorizationRule
{
  protected override void Execute(Csla.Rules.AuthorizationContext context)
  {
```

```
      context.HasPermission = true; // return appropriate result
    }
  }
```

Earlier, in Table 8, I listed the `CommonRules` types provided by CSLA .NET, including the `IsInRole` and `IsNotInRole` authorization rules. Those rules give you the functionality necessary to leverage the standard .NET and Windows role-based authorization model. If you have a more sophisticated model based around permissions or claims, you can create your own authorization rule types to use instead of the pre-built rules.

Also, it is important to remember that authorization rules are business logic, they aren't just security. You can use authorization rules to control when a user can or can't read or write to a property, among other actions. Sometimes those actions are restricted simply by roles or permissions, but sometimes they are also restricted by the state of the business object, other business objects, the application, the user, or other factors. You can create custom authorization rules to meet those needs, as well as the more basic security-related authorization concepts.

The business rules subsystem provided by CSLA .NET is flexible and powerful. What I've discussed here is all at a high level, and I'll dig much deeper into how you can implement and leverage business rules through the rest of this book series.

Data Portal

Supporting object persistence—the ability to store and retrieve an object from a database—can be quite complex. I discussed this earlier in the chapter when talking about basic persistence and the concept of ORM.

CSLA .NET is not an ORM, nor does it dictate how your application stores its data, or even how you interact with your database. What CSLA .NET does is abstract the concept of communication between the client device and any application server, allowing you to switch between 1-, 2-, 3-, and 4-tier physical deployments without changing your code.

This abstraction of communication is handled by the data portal. Not only does the data portal abstract interaction with any application server, but it enables the concept of mobile objects I discussed in Chapter 2, ensuring that your business objects flow back and forth between the client and server as necessary.

Of course the point of having an application server is typically to move any data access logic off the client and onto a server. Recognizing this, the data portal is designed to formalize the way in which business objects interact with your Data Access layer. For the most part, CSLA .NET doesn't specify how you create that Data Access layer, but it does impose certain requirements around how you invoke the Data Access layer.

Managing Server-Side Types

When using an application server, not every business object in the application should be directly exposed by the server. This would be a maintenance and configuration nightmare, because it would require updating configuration information on all client devices any time a business object is added or changed.

Instead, it would be ideal if there were one consistent entry point to the application server, so that every client could be configured to know about that single entry point and never have to worry about it again. This is exactly what the data portal concept provides, as shown in Figure 33.

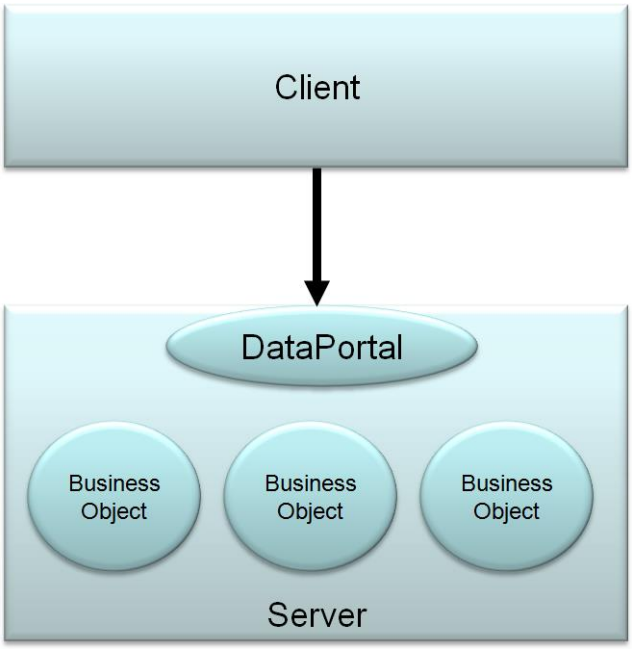

Figure 33. The data portal provides a consistent entry point to the application server.

The data portal provides a single point of entry and configuration for the server. It manages communication with the business objects while they're on the server running their data access code. Additionally, the data portal concept provides the following other key benefits:

- Centralized security when calling the application server
- A consistent object-persistence mechanism (all objects persist the same way)
- Abstraction of the network transport between client and server (enabling support for WCF, remoting, web services, Enterprise Services, and custom protocols)
- One point of control to toggle between running the data access code locally, or on a remote application server

The data portal functionality is designed in several parts, as shown in Table 12.

Area	Functionality
Client-side `DataPortal`	Functions as the primary entry point to the data portal infrastructure, for use by code in business objects
Client-side proxy classes	Implement the channel adapter pattern to abstract the underlying network protocol from the application
Message objects	Transfer data to and from the server, including security information, application context, the business object's data, the results of the call, or any server-side exception data

Server-side host classes	Expose single points of entry for different server hosts, such as WCF, remoting, asmx web services and Enterprise Services
Server-side data portal	Implements transactional and non-transactional data access behaviors, delegating all data access to appropriate business objects
Server-side child data portal	Implements data access behaviors for objects that are contained within other objects
Object factory	Provides an alternate model for the data portal, where the data portal creates and invokes a factory object, instead of interacting directly with the business object

Table 12. Parts of the Data Portal Concept

I'll discuss each area of functionality in turn.

Client-Side DataPortal

The client-side `DataPortal` can be used in two different ways: via a a `static` class or by creating an instance of the class each time.

When using the `static` class any `public` methods exposed by the `DataPortal` are available to business object code without the need to create a `DataPortal` object. The methods it provides are `CreateAsync`, `FetchAsync`, `UpdateAsync`, `DeleteAsync`, and `ExecuteAsync` (and for legacy code there are synchronous versions of each method). Business objects and collections use these methods to retrieve and update data, or in the case of a `CommandBase`-derived object, to execute server code on the server.

The `static` methods are often convenient but prohibit mocking and dependency injection. Fortunately, it is also possible to create an instance of the client-side `DataPortal` and reference it via the `IDataPortal` interface. This enables mocking and dependency injection provide a better experience for unit testing.

The client-side `DataPortal` has a great deal of responsibility because it contains the code to act on the client's configuration settings. These settings control whether the "server-side" data portal components will run remotely on a physical server or locally on the client. It also looks at the business object itself, since a `RunLocal` attribute can be used to force persistence code to run on the client, even if the configuration says to run it on the server. And it uses an `IsOffline` property to switch between "online" and "offline" configurations to support scenarios where the client device may lose connectivity to the network.

If the client configuration indicates that the server-side data portal will run on a remote server, the configuration will also specify which network transport should be used. It is the client-side `DataPortal` that loads the appropriate client-side proxy object based on configuration. That proxy object is then responsible for handling the network communication.

In any case, client-side `DataPortal` always delegates the call to the server-side data portal, which handles the object-persistence behaviors.

Applications use the client-side `DataPortal` to retrieve and update the object's information. An automatic result is that the application code won't need to know about network transports or

whether the application is deployed into a 1-, 2-, or n-tier physical environment. The calling code always looks something like this:

```
var order = await DataPortal.FetchAsync<OrderEdit>(id);
```

This code doesn't change, regardless of whether you've configured the server-side data portal to run locally, or on a remote server via HTTP, WCF, or through a service bus or queue. All that changes is the application's configuration file.

Client-Side Proxies

Although it is the client-side `DataPortal` that uses the client configuration to determine the appropriate network transport, the client-side proxy classes take care of the details of each network technology. There is a different proxy class for each technology.

The combination of a data portal proxy and a data portal host is called a *data portal channel*. CSLA includes one data portal channel in the framework itself, communicating over the network using http/https. This client-side proxy is named `HttpProxy`.

The design also allows for a business application to provide its own proxy class to use other protocols. This means you can write your own data portal channel to communicate over WCF, Azure Service Bus, RabbitMQ, or nearly any other network transport.

Every client-side proxy has a corresponding server-side host class. This is because each transport protocol requires that both ends of the network connection use the same technology.

The client-side `DataPortal` creates an instance of the appropriate client-side proxy and then delegates the request (`CreateAsync`, `FetchAsync`, `UpdateAsync`, `DeleteAsync`, or `ExecuteAsync`) to the proxy object. The proxy object is responsible for establishing a network connection to the server-side host object and delegating the call across the network.

The proxy must also pass other message data, such as security and application context, to the server. Similarly, the proxy must receive data back from the server, including the results of the operation, application context information, and any exception data from the server.

To this last point, if an exception occurs on the server, the full exception details are returned to the client. This includes the nature of the exception, any inner exceptions, and the stack trace related to the exception. This exception information will often be used on the client to rethrow the exception, giving the illusion that the exception flowed naturally from the code on the server back to the code on the client.

Message Objects

When the client-side `DataPortal` calls the server-side data portal, several types of information are passed from client to server. Obviously, the data method call (`CreateAsync`, `UpdateAsync`, `InsertAsync`, etc.) itself is transferred from client to server. But other information is also included, as follows:

- The client device's UI culture
- Client-side context data defined by the application
- Application-wide context data defined by the application

- The user's principal and identity security objects (if using custom authentication)

Client-side culture and context data is passed one way, from the client to the server. The client UI culture is not only passed to the server, but the server thread processing the client request is set to use that culture. This can be important for localization of an application when a server may be used by devices in different nations. Any application-defined client context data can be used by your server-side code as you see fit.

Application-wide context data is passed both from client to server and from server back to client. You may use this context data to pass arbitrary application-specific data between client and server on each data portal operation. This can be useful for debugging, as it allows you to build up a trace log of the call as it goes from client to server and back again.

CSLA .NET also includes the concept of local context, which is *not* passed from client to server or server to client. Local context exists on the client and the server, but each has its own separate context.

If the application is using custom authentication, then the custom principal and identity objects representing the user are passed from client to server. This means the code on the server will run under the same security context as the client. If you are using Windows integrated or AD security, then you must configure your network transport technology to handle the impersonation.

When the server-side data portal has completed its work, the results are returned to the client. Other information is also included, as follows:

- Application-wide context data (as defined by the application)
- Details about any server-side exception that may have occurred

Again, the application-wide context data is passed from client to server and from server to client.

If an exception occurs on the server, the details about that exception are returned to the client. This is important for debugging, as it means you get the full details about any issues on the server. It is also important at runtime, because it allows you to write exception handling code on the client to gracefully handle server-side exceptions—including data-oriented exceptions such as duplicate key or concurrency exceptions.

All of the preceding bulleted items are passed to and from the server on each data portal operation. Keeping in mind that the data portal supports several verbs, it is important to understand what information is passed to and from the server to support each verb. This is listed in Table 13.

Verb	To Server	From Server
`CreateAsync`	Type of object to create and (optional) criteria about new object	New object loaded with default values
`FetchAsync`	Type of object to retrieve and criteria for desired object	Object loaded with data
`UpdateAsync`	Object to be updated	Object after update (possibly containing changed data)

DeleteAsync	Type of object to delete and criteria for object to be deleted	Nothing
ExecuteAsync	Object to be executed (must derive from CommandBase)	Object after execution (possibly containing changed data)

Table 13. Data Passed to and from the Server for Data Portal Operations

Notice that the CreateAsync, FetchAsync, and DeleteAsync operations all use criteria information about the object to be created, retrieved, or removed. A criteria value contains any data you need to describe your business object. Criteria can be one of the following types of value:

- Any primitive value type (int, string, etc.)
- Any serializable type (marked with the Serializable attribute and implementing IMobileObject)
- A subclass of CriteriaBase<T>

Most criteria values are simple primitive values, often an int or string. If your criteria consists of more than one value, or it is something other than a primitive value type then you'll need to create your own criteria class. This class must be serializable, which means it must be Serializable and implement IMobileObject. The simplest way to create a custom criteria class is to subclassing CriteriaBase<T>. For example:

```
[Serializable]
public class CustomCriteria : CriteriaBase<CustomCriteria>
{
  public static readonly PropertyInfo<int> IdProperty = RegisterProperty<int>(c => c.Id);
  public int Id
  {
    get { return ReadProperty(IdProperty); }
    set { LoadProperty(IdProperty, value); }
  }

  public static readonly PropertyInfo<int> PositionProperty =
    RegisterProperty<int>(c => c.Position);
  public int Position
  {
    get { return ReadProperty(PositionProperty); }
    set { LoadProperty(PositionProperty, value); }
  }
}
```

This criteria class contains two values: Id and Position.

It is also possible to use other base classes, such as BusinessBase or ReadOnlyBase to define a criteria type. BusinessBase is often used, for example, when there is the need for the criteria to be edited directly by the user before sending it to the data portal. Because a BusinessBase derived class supports data binding and business rules this can simplify your implementation.

Server-Side Host Objects

I've already discussed the client-side proxy objects and how each one has a corresponding server-side host object. Server-side host objects are responsible for two things: first, they must accept

inbound requests over the appropriate network protocol from the client, and those requests must be passed along to the server-side data portal components; second, the host object is responsible for running inside the appropriate server-side host technology.

Microsoft provides server-side host technologies for hosting application server code, including ASP.NET Core and Internet Information Services (IIS). Basically, any hosting environment that allows you to expose a service endpoint can be used to create a data portal host.

The most common endpoint in current technologies is an ASP.NET controller. This is implemented in the CSLA ASP.NET NuGet packages (for .NET Framework and .NET Core) as `Csla.Server.Hosts.HttpPortalController`.

An older endpoint is implemented using WCF. This endpoint is only available on some .NET implementations and is generally used for legacy applications that haven't yet upgraded to the more modern technique.

Server-Side Data Portal

The server-side data portal components provide an implementation of the message router design pattern. The server-side data portal accepts requests from the client and routes those requests to an appropriate handler—either a business object or a factory object.

> I say "server-side" here, but keep in mind that the server-side data portal components may run either on the client device or on a remote server. Refer to the client-side `DataPortal` discussion regarding how this selection is made. The data portal is implemented to minimize overhead as much as possible when configured to run locally or remotely, so it is appropriate for use in either scenario.

For `CreateAsync`, `FetchAsync`, and `DeleteAsync` operations, the server-side data portal requires type information about your business object. For update and execute operations, the business object itself is passed to the server-side data portal.

But the server-side data portal is more than a simple message router. It also provides optional access to the transactional technologies available within .NET, namely the `System.Transactions` namespace and the legacy Enterprise Services feature of the .NET Framework.

The business framework defines a custom attribute named `TransactionalAttribute` that can be applied to methods within business objects. Specifically, you can apply it to any of the data access methods that your business object might implement to create, fetch, update, or delete data, or to execute server-side code. This allows you to use one of three models for transactions, as listed in Table 14.

Option	Description	Transactional Attribute
`Manual`	You are responsible for implementing your own transactions using ADO.NET, stored procedures, etc.	None or `[Transactional(TransactionalTypes.Manual)]`

Enterprise Services	Your data access code will run within a COM+ distributed transactional context, providing distributed transactional support.	[Transactional(TransactionalTypes. EnterpriseServices)]
System.Transactions	Your data access code will run within a TransactionScope from System.Transactions, automatically providing basic or distributed transactional support as required.	[Transactional(TransactionalTypes. TransactionScope)]

Table 14. Transaction Options Supported by Data Portal

This means that in the business object, there may be an update method (overriding the one in BusinessBase) marked to be transactional:

```
[Transactional(TransactionalTypes.TransactionScope)]
protected override void DataPortal_Update()
{
  // Data update code goes here
}
```

At the same time, the object might have a fetch method in the same class that's *not* transactional:

```
private void DataPortal_Fetch(Criteria criteria)
{
  // Data retrieval code goes here
}
```

Or if you are using an object factory (discussed in the next section), the Transactional attribute is applied to the Update method in the factory class:

```
public class MyFactory : Csla.Server.ObjectFactory
{
  [Transactional(TransactionalTypes.TransactionScope)]
  public object Update()
  {
    // Data update code goes here
  }
}
```

This facility means that you can control transactional behavior at the method level, rather than at the class level. This is a powerful feature, because it means that you can do your data retrieval outside of a transaction to get optimal performance, and still do updates within the context of a transaction to ensure data integrity.

The server-side data portal examines the appropriate method on the business object before it routes the call to the business object itself. If the method is marked with [Transactional(TransactionalTypes.TransactionScope)],the call is routed to a TransactionalDataPortal object that is configured to run within a System.Transactions.TransactionScope. A TransactionScope is powerful because it provides a lightweight transactional wrapper in the case that you are updating a single database; but it automatically upgrades to a distributed transaction if you are updating multiple databases. In short,

you get the benefits of COM+ distributed transactions if you need them, but you don't pay the performance penalty if you don't need them.

If the method is marked as `[Transactional(TransactionalTypes.EnterpriseServices)]`, then the call is routed to a `ServicedDataPortal` object that is configured to require a COM+ distributed transaction. The `ServicedDataPortal` then calls the `SimpleDataPortal`, which delegates the call to your business object, but only after it is running within a distributed transaction.

> Enterprise Services is only available in the .NET Framework, not in .NET Core. As such, this should be viewed as a legacy technology, and you should prefer the use of more modern techniques that are supported in .NET Core.

Either way, your code is transactionally protected.

If the method doesn't have the attribute, or is marked as `[Transactional(TransactionalTypes.Manual)]`, the call is routed directly to the `SimpleDataPortal`, as illustrated in Figure 34.

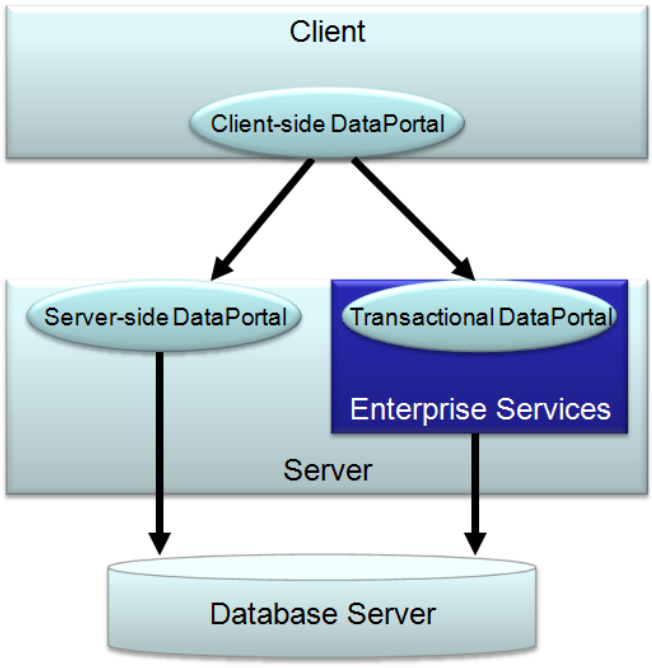

Figure 34. Routing calls through transactional wrappers

Object Factory Model

By default, the server-side data portal components route calls to methods of an instance of the business object itself. The business object becomes responsible for invoking the Data Access layer to get or save any data. An alternative is for the data portal to create an instance of a factory object, and invoke methods on that object instead.

The result is that there are four basic locations where you can implement data access code as listed in Table 15.

Data portal invokes	Data access location	Pros and Cons
Business object methods	Directly in business object methods Known as *Encapsulated Implementation*	Pros: Simplest model, and business object encapsulation is preserved Cons: Lacks separation between business layer and data access layer
Business object methods	In separate Data Access layer, typically loaded using a layer of indirection **Recommended technique** Known as *Encapsulated Invocation*	Pros: Business object encapsulation is preserved, and provides clean separation between business layer and Data Access layer Cons: Requires careful design of Data Access layer interface to avoid coupling between business and Data Access layers
Factory object methods	Directly in factory object methods Known as *Factory Implementation*	Pros: Provides clean separation between business layer and Data Access layer Cons: Breaks business object encapsulation
Factory object methods	In separate data access layer, typically loaded using a layer of indirection Known as *Factory Invocation*	Pros: Provides clean separation between business layer and Data Access layer Cons: Most complex model, breaks business object encapsulation, and involves an extra layer of indirection that is typically not needed

Table 15. Data access models supported by data portal

My default recommendation is to use the Encapsulated Invocation model: have the data portal invoke the business object methods, and to use a separate Data Access layer loaded using a layer of indirection. I typically use a simple provider model, or an inversion of control framework to dynamically load the Data Access layer, but I invoke it from within the business object. This preserves encapsulation within the business object, and at the same time provides a clean and elegant way to completely separate the data access code into its own assembly.

I'll discuss these data access options in more detail through the rest of the book series.

Data Portal Behaviors

Now that you have a grasp of the areas of functionality required to implement the data portal concept, let's discuss the specific data behaviors the data portal will support. The behaviors were listed earlier, in Table 13.

Create

The "create" operation is intended to allow the business objects to load themselves with values that must come from the database. Business objects don't need to support or use this capability, but if they do need to initialize default values, then this is the mechanism to use.

There are many types of applications for which this is important. For instance, order entry applications typically have extensive defaulting of values based on the customer. Inventory management applications often have many default values for specific parts, based on the product family to which the part belongs. Medical records also often have defaults based on the patient and physician involved.

Although you can use the `new` keyword to directly create an instance of a business class, I recommend always using the data portal to create your business objects. Not only does this provide consistency to your code, but it also allows you to add initialization logic later if necessary.

You may choose to abstract calls to the data portal within `static` methods implemented in your business classes. This has the advantage of hiding the use of the data portal from any UI code but has the drawback of making unit testing difficult.

A `static` factory method abstracts the creation of the new object, and generally looks like this:

```
public static async Task<EmployeeEdit> NewEmployeeAsync()
{
  return await DataPortal.CreateAsync<EmployeeEdit>();
}
```

Although this simplifies the UI code, it is typically better to have the UI code use the data portal directly to enable unit testing scenarios. For example, the UI code could look like this:

```
var obj = await DataPortal.CreateAsync<EmployeeEdit>();
```

Or, to further enable unit testing you can explicitly create an instance of the client-side `DataPortal`:

```
var dp = new DataPortal<EmployeeEdit>();
var obj = await dp.CreateAsync();
```

The use of the `new` keyword to create the `DataPortal` instance can be replaced by your dependency injection framework of choice, enabling mocking and other unit testing techniques.

Notice that in no case is a `EmployeeEdit` object instance created on the client here. Instead, the factory method asks the client-side `DataPortal` for the `EmployeeEdit` object. The client-side `DataPortal` passes the call to the server-side data portal. If the data portal is configured to run remotely, the business object is created on the server; otherwise, the business object is created locally on the client.

The server-side data portal looks at your business class to see if it has an `ObjectFactory` attribute. That attribute tells the data portal to create an instance of a factory object, and to invoke a `Create` method on that factory object. You are responsible for writing the factory class and its `Create` method. This method must be `public` in scope. Here's what a factory class might look like:

```
public class EmployeeFactory : ObjectFactory
{
  public EmployeeEdit Create()
  {
    var result = new EmployeeEdit();
```

```
      using (BypassPropertyChecks(result))
      {
        // initialize properties here
      }
      MarkNew(result);
      return result;
    }
  }
```

If there is no `ObjectFactory` attribute, the data portal directly creates an instance of the business class, and invokes a `DataPortal_Create` method on that object. You are responsible for writing the `DataPortal_Create` method. Here's what the method might look like:

```
    protected override void DataPortal_Create()
    {
      using (BypassPropertyChecks)
      {
        // initialize properties here
      }
      base.DataPortal_Create();
    }
```

This method may also be async, for example:

```
    private async Task DataPortal_Create()
    {
      using (BypassPropertyChecks)
      {
        // initialize properties here
      }
      base.DataPortal_Create();
    }
```

Regardless of whether the data portal invoked a method on an object factory or the business object itself, the result is a fully initialized new business object. At this point the server-side data portal returns the business object back to the client-side `DataPortal`. If the two are running on the same device, this is a simple object reference; but if they're configured to run on separate devices, then the business object is serialized across the network to the client (that is, it's passed by value), so the client device ends up with a local copy of the business object.

Fetch

Retrieving a preexisting object is similar to the creation process just discussed. Again, when you call the data portal you can pass an optional criteria value as a parameter. The UI interacts with the factory method, which in turn creates a criteria object and passes it to the client-side `DataPortal` code. The client-side `DataPortal` determines whether the server-side data portal should run locally or remotely, and then delegates the call to the server-side data portal components.

The calling code typically looks like this:

```
    var obj = await DataPortal.FetchAsync<EmployeeEdit>(id);
```

Notice how a criteria value is passed as a parameter to the `FetchAsync` method. Remember that the `CreateAsync`, `FetchAsync`, and `DeleteAsync` methods can optionally accept *one* criteria parameter. If you need to pass multiple criteria values, you need to create a serializable custom class that contains those values, typically by subclassing `CriteriaBase<T>`.

If your business class has the `ObjectFactory` attribute, you'll create a factory object something like this:

```
public class EmployeeFactory : ObjectFactory
{
  public EmployeeEdit Fetch(int id)
  {
    var result = new EmployeeEdit();
    using (BypassPropertyChecks(result))
    {
      // invoke the data access layer and
      // initialize properties here
    }
    MarkOld(result);
    return result;
  }
}
```

Otherwise you'll implement a `DataPortal_Fetch` method directly in the business class.

```
  private void DataPortal_Fetch(int id)
  {
    using (BypassPropertyChecks)
    {
      // invoke the data access layer and
      // initialize properties here
    }
  }
```

Or as an async method:

```
  private async Task DataPortal_Fetch(int id)
  {
    using (BypassPropertyChecks)
    {
      // invoke the data access layer and
      // initialize properties here
    }
  }
```

As with the create process, in an n-tier physical configuration, the criteria value (if any) and business object move by value across the network, as required. The resulting business object is returned to the calling code from the factory method.

Update

The update process is a bit different from the previous operations. In this case, the UI already has a business object with which the user has been interacting, and this object needs to save its data into the database.

To achieve this, all editable business objects have `Save` and `SaveAsync` methods (as part of the `BusinessBase` class from which all business objects inherit). The save methods call the `DataPortal` to do the update, passing the business object itself, `this`, as a parameter.

The thing to remember when doing updates is that the object's data will likely change as a result of the update process. Any changed data must be placed back into the object. So the UI code will look like this:

```
    employee = employee.Save();
```

Or like this:

```
    employee = await employee.SaveAsync();
```

Notice how in both examples the original object reference is replaced to use the *new object instance returned from the save operation*. In practice this also means updating all data binding or other references to the object throughout your client-side code. To avoid having to find and update all those references (and data binding), you can choose to merge the results of the save operation back into the original object graph:

```
    await employee.SaveAndMergeAsync();
```

In most cases this approach is the simplest and should be your choice.

There are two common scenarios illustrating how data changes during an update. The first is when the database assigns the primary key value for a new object. That new key value needs to be put into the object and returned to the client. The second scenario is when a timestamp is used to implement optimistic first-write-wins concurrency. In this case, every time the object's data is inserted or updated, the timestamp value must be refreshed in the object with the new value from the database. Again, the updated object must be returned to the client.

This means that the update process is *bidirectional*. It isn't just a matter of sending the data to the server to be stored, but also a matter of returning the object *from* the server after the update has completed, so that the UI has a current, valid version of the object.

Due to the way the data portal passes objects by value, it may introduce a bit of a wrinkle into the overall process. When passing the object to be saved over to the server, .NET makes a copy of the object from the client onto the server, which is exactly what is desired. The wrinkle arises after the update is complete, and the object must be returned to the client. When an object is returned from the server to the client, a new copy of the object is made on the client, which isn't the desired behavior.

Figure 35 illustrates the initial part of the update process.

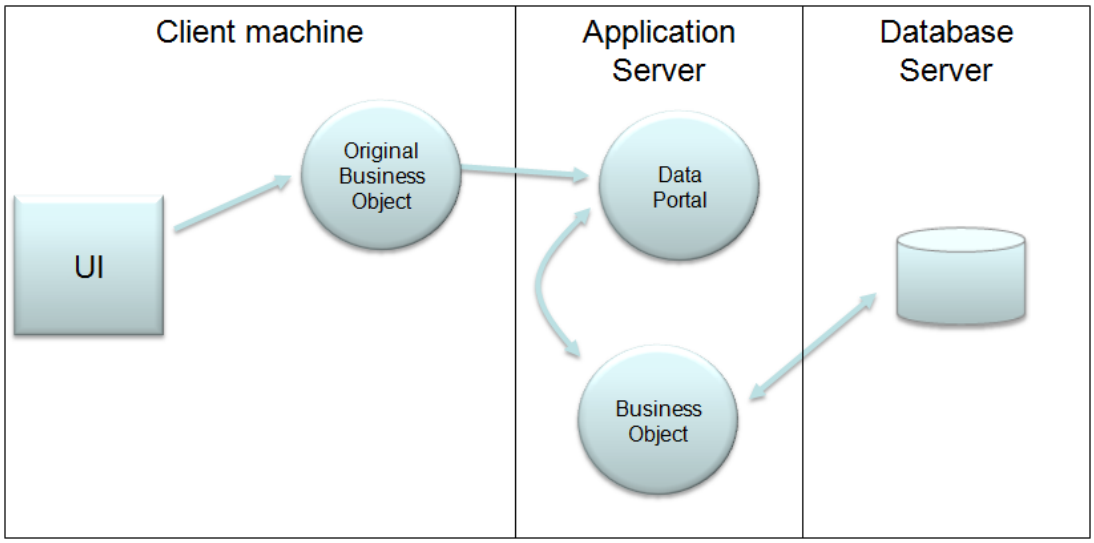

Figure 35. Sending a business object to the data portal to be inserted or updated

The UI has a reference to the business object and calls its `SaveAsync` method. This causes the business object to ask the data portal to save the object. The result is that a copy of the business

object is made on the server, where it can save itself to the database. So far, this is pretty straightforward.

The business object has a `SaveAsync` method, but the data portal infrastructure has methods named `Update`. Although this is a bit inconsistent, remember that the business object is being called by UI developers, and I've found that it's more intuitive for the typical UI developer to call `Save` than `Update`, especially because the `Save` call can trigger an `Insert`, `Update`, or even `Delete` operation.

Once this part is done, the updated business object is returned to the client, and the UI must update its references to use the *newly updated* object instead, as shown in Figure 36.

This is fine, too—but it's important to keep in mind that you can't continue to use the old business object; you must update all object references to use the newly updated object or merge the new object into the original.

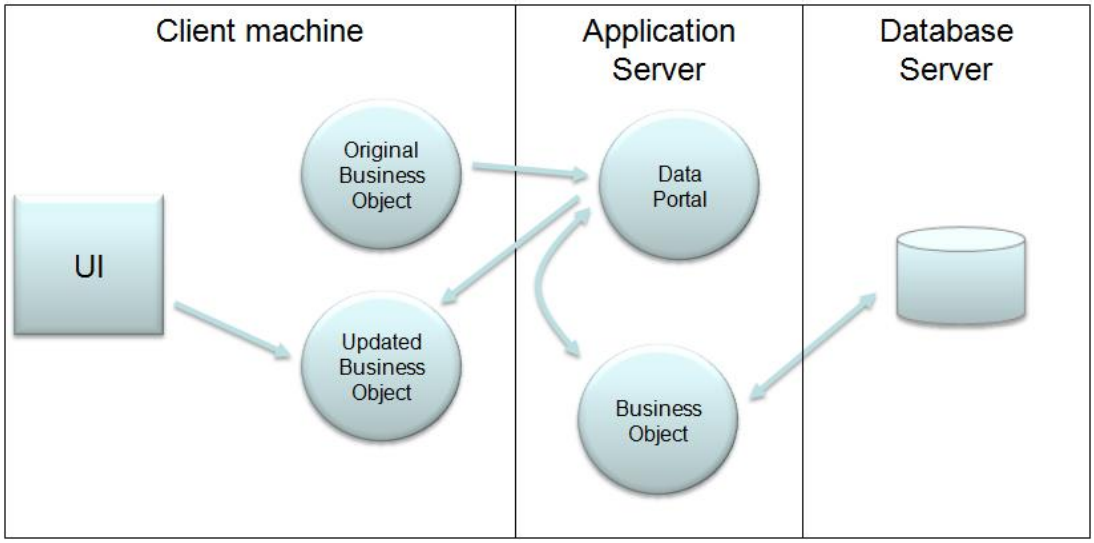

Figure 36. Data portal returning the inserted or updated business object to the UI

When the UI calls the `SaveAsync` method on the business object that results in a call to the client-side `DataPortal`'s `Update` method, passing the business object as a parameter. As usual, the client-side `DataPortal` determines whether the server-side data portal is running locally or remotely, and then delegates the call to the server-side data portal.

The server-side data portal looks at the business class to see if it has an `ObjectFactory` attribute. If it does, the data portal creates an instance of the factory class and invokes an `Update` method. The method might look like this:

```
public EmployeeEdit Update(EmployeeEdit obj)
{
  if (obj.IsDeleted)
  {
    if (!obj.IsNew)
    {
      // invoke data access layer to delete data
    }
    MarkNew(obj);
  }
  else
  {
```

```
      if (obj.IsNew)
      {
        // invoke data access layer to insert data
      }
      else
      {
        // invoke data access layer to update data
      }
      using (BypassPropertyChecks(obj))
      {
        // update any changed properties here
      }
      MarkOld(obj);
    }
    return obj;
  }
```

Notice that the `Update` method is responsible for using the business object's `IsNew` and `IsDeleted` properties to determine whether to have the Data Access layer insert, update, or delete data.

If the business class doesn't have the `ObjectFactory` attribute, the data portal looks at the object's `IsNew` and `IsDeleted` properties and automatically calls `DataPortal_Insert`, `DataPortal_Update`, or `DataPortal_DeleteSelf` methods as appropriate. These methods might look like this:

```
  protected override void DataPortal_Insert()
  {
    // invoke the data access layer to insert the data
    using (BypassPropertyChecks)
    {
      // update any changed property values
    }
  }

  protected override void DataPortal_Update()
  {
    // invoke the data access layer to update the data
    using (BypassPropertyChecks)
    {
      // update any changed property values
    }
  }

  protected override void DataPortal_DeleteSelf()
  {
    // invoke the data access layer to delete the data
  }
```

At this point, two versions of the business object exist: the original version on the client and the newly updated version on the application server. The best way to view this is to think of the original object as being obsolete and invalid at this point. Only the newly updated version of the object is valid.

Once the update is done, the new version of the business object is returned to the UI; the UI can then interact with the *new* business object as needed.

> The UI must merge the new object into the original or update any references from the old business object to the newly updated business object as soon as the new object is returned from the data portal.

In a physical n-tier configuration, the business object is automatically passed by value to the server, and the updated version is returned by value to the client.

If the server-side data portal is running locally, the object is cloned and the clone is updated and returned to the calling code. This is necessary because it is possible for the update process to fail half-way through. If your business object contains other business objects, some might have been changed during the update process, whereas others are unchanged. The database transaction will ensure that the database is in a consistent state, but your object model can be left in an inconsistent state. By saving a clone, if the update fails the UI is left referencing the *original unchanged object*, which is still in a consistent state.

Execute

CSLA .NET supports the concept of a command object, which is a business object that inherits from `CommandBase<T>`. A command object represents some server-side command or action, and it is used to encapsulate processing other than simple create, read, update, and delete operations.

A command object isn't saved, it is executed. To support this, the data portal has `Execute` and `ExecuteAsync` methods. Here's an example of calling a command via the `DataPortal`:

```
var cmd = new OrderProcessor { OrderId = id };
cmd = await DataPortal.ExecuteAsync<OrderProcessor>(cmd);
```

Notice that the data portal's `ExecuteAsync` method is invoked, passing the command object to the server where it can execute. The command object is returned as a result, though it is a new and updated instance of the original object. This works much the same as the Update operation I discussed.

The server-side data portal looks for an `ObjectFactory` attribute on the business class, and if that attribute exists a factory object is created, and its `Execute` method is invoked. That method might look like this:

```
public OrderProcessor Execute(OrderProcessor cmd)
{
  // implement server-side processing here, possibly
  // including use of the data access layer to
  // read and update data
  return cmd;
}
```

If there is no `ObjectFactory` attribute, a `DataPortal_Execute` method on the business object itself is invoked. That method might look like this:

```
protected override void DataPortal_Execute()
{
  // implement server-side processing here, possibly
  // including use of the data access layer to
  // read and update data
}
```

Or async:

```
private async Task DataPortal_Execute()
{
  // implement server-side processing here, possibly
  // including use of the data access layer to
  // read and update data
}
```

Once the command is done executing on the server, the object is returned to the client so the calling code can use any results of the operation.

Delete

The final operation, and probably the simplest, is to delete an object from the database. The framework supports two approaches to deleting objects.

The first approach is called *deferred deletion*. In this model, the object is retrieved from the database and is marked for deletion by calling a `Delete` method on the business object. Then the `Save` or `SaveAsync` method is called to cause the object to update itself to the database. In other words, deferred deletion uses the Update process I discussed.

The second approach, called *immediate deletion*, consists of passing a criteria value to the server, where the corresponding data is deleted immediately by calling the `Delete` or `DeleteAsync` method of the client-side `DataPortal`. This results in a server-side call to a factory object, or a `DataPortal_Delete()` method in the business object.

This second approach provides superior performance because you don't need to load the object's data and return it to the client. Instead, you pass the criteria fields to the server, where the object deletes its data.

The framework supports both immediate and deferred models, providing you with the flexibility to allow either or both in your object models, as you see fit.

When using immediate deletion the UI code looks like this:

```
await DataPortal.DeleteAsync<EmployeeEdit>(42);
```

As you can probably guess, the client-side data portal's `DeleteAsync` method looks at the client configuration to determine whether to invoke the server-side data portal locally or remotely. Either way, the call is handed off to the server-side data portal.

The server-side data portal looks at the business class to see if there's an `ObjectFactory` attribute. If there is, the data portal creates an instance of the factory class and calls a `Delete` method that looks like this:

```
public void Delete(int id)
{
  // invoke data access layer to delete data
}
```

Otherwise the data portal creates an instance of the business object and invokes a `DataPortal_Delete` method on that object:

```
private void DataPortal_Delete(int id)
{
  // invoke the data access layer to delete the data
}
```

Or async:

```
private async Task DataPortal_Delete(int id)
{
  // invoke the data access layer to delete the data
}
```

As you can see, the data portal abstracts communication between the client and any application server, and also provides a high level framework within which you will invoke your Data Access layer.

Custom Authentication

As discussed earlier in the chapter, many environments include users who aren't part of a Windows domain or AD. In such a case, relying on Windows integrated security for the application is problematic at best, and you're left to implement your own security scheme.

The full .NET Framework includes several security concepts, along with the ability to customize them to implement your own security as needed. Some other .NET implementations don't include all these concepts, so CSLA .NET implements them on your behalf. This means that CSLA .NET provides a relatively consistent authentication model across all implementations of .NET.

Specifically, across all application types and .NET implementations CSLA .NET exposes a consistent view of the current user's identity (principal) via the `ApplicationContext`:

```
var userPrincipal = Csla.ApplicationContext.User;
```

This user principal object may be a `WindowsPrincipal` or some other type that implements the `IPrincipal` interface from the `System.Security.Principal` namespace. Ideally the principal is provided to the application by the runtime environment (such as Windows), but that is not always the case. If the runtime environment doesn't manage the user's identity on your behalf, then you are responsible for managing the user's identity.

Windows and .NET may manage the user's identity (principal) on your behalf when your code is running entirely on a Windows server with Windows integrated security enabled. It may also manage the user's identity on smart clients (Windows Forms, WPF, and UWP) when the client device is domain-joined and Windows integrated security is enabled on client and server.

Also, if you are running a .NET Core application on a Linux server with ASP.NET Core, then ASP.NET Core can be configured to manage the user's identity.

Outside of those scenarios you will be responsible for some or all user identity management. This is accomplished using custom principal and identity types.

Custom Principal and Identity Objects

.NET includes a couple of built-in *principal* and *identity* objects that support Windows integrated security or generic security. You can also create your own principal and identity objects by creating classes that implement the `IPrincipal` and `IIdentity` interfaces from the `System.Security.Principal` namespace.

Implementations of principal and identity objects will be specific to your environment and security requirements. CSLA .NET includes `CslaPrincipal` and `CslaClaimsPrincipal` classes to streamline the process.

When you create a custom principal object, it must implement the `IPrincipal` interface from `System.Security.Principal`, and it must be serializable. The easiest way is to create a customer type is to inherit from `CslaPrincipal` or `CslaClaimsPrincipal`. In many cases, your custom principal object will require very little code. The base classes already implement the `IPrincipal`

interface and provide default methods that work automatically if your custom identity is a subclass of `CslaIdentity` or `CslaClaimsIdentity` respectively.

You will also need to implement a custom identity object that is serializable and implements `IIdentity`. Typically, this object will populate itself with user profile information and a list of user roles from a database. CSLA .NET provides `CslaIdentity` and `CslaClaimsIdentity` classes you can use to easily create custom identity types.

You'll also need to implement a `Login` method that the UI code can call to initiate the process of authenticating the user's credentials (username and password) and loading data into the custom identity object. In many cases, this method will look something like this:

```
public async Task LoginAsync(string username, string password)
{
  var criteria = new UserCriteria(username, password);
  IIdentity identity = DataPortal.FetchAsync(criteria);
  if (identity.IsAuthenticated)
  {
    IPrincipal principal = new CustomPrincipal(identity);
    Csla.ApplicationContext.User = principal;
  }
}
```

The primary operation here is fetching the custom identity object via the data portal. This allows the server-side code to verify the user's credentials and retrieve any user-specific roles or claims from your security database, LDAP server, or other identity server. The way .NET defines principal types is that they are basically just containers for the user's identity object.

A corresponding `Logout` method may look like this:

```
public void Logout()
{
  Csla.ApplicationContext.User = new Csla.Security.UnauthenticatedPrincipal();
}
```

CSLA .NET includes an `UnauthenticatedPrincipal` type that can be used to indicate that no user is logged into the application at this time.

> Most smart client platforms (Xamarin, Windows Forms, WPF, UWP, Blazor) require that your application explicitly set the current principal to an unauthenticated value as the application is first loaded into memory. It is recommended that you call your `Logout` method in the startup code of any smart client app.

I go into much more detail about authentication and authorization in the *Using CSLA 2019: Security* book.

Helper Types and Classes

Most business applications require a set of common behaviors not covered by the concepts discussed thus far. These behaviors are a grab bag of capabilities that can be used to simplify common tasks that would otherwise be complex. These include the items listed in Table 16.

Type or Class	Description
ConnectionManager	Enables easy reuse of an open database connection, making the use of TransactionScope transactions more practical
ObjectContextManager	Enables easy reuse of an Entity Framework object context, making the use of TransactionScope transactions more practical
ContextManager	Enables easy reuse of a LINQ to SQL data context, making the use of TransactionScope transactions more practical
TransactionManager	Enables easy reuse of an ADO.NET transaction object
SafeDataReader	Wraps any IDataReader (such as SqlDataReader) and converts all null values from the database into non-null empty or default values
DataMapper	Maps data from an IDictionary to an object's properties, or from one object's properties to another object's properties
SmartDate	Implements a DateTime data type that understands both how to translate values transparently between DateTime and string representations and the concept of an empty date

Table 16. Helper Types and Classes

Let's discuss each of these in turn.

ConnectionManager

The TransactionScope class from System.Transactions is typically the preferred technology for implementing data update transactions, because it results in simpler code and good performance. Unfortunately, TransactionScope will automatically invoke the Distributed Transaction Coordinator (DTC) if your code opens more than one database connection, and that results in a substantial performance penalty (often around 15%). If you avoid opening multiple database connections then TransactionScope uses a light-weight transaction scheme that is just as safe, but is much faster.

The result is that you should reuse one open database connection across all your objects when using a TransactionScope object for transactional support. This means you must write code to open the connection object and then make it available to all objects that will be interacting with the database within the transaction. That can unnecessarily complicate what should be simple data access code.

The Csla.Data.ConnectionManager class is intended to simplify this process by managing and automatically reusing a single database connection object. The result is that all data access code that uses a database connection object has the following structure:

```
using (var ctx = ConnectionManager<SqlConnection>.GetManager("DatabaseName"))
{
  // ctx.Connection is now an open connection to the database
  // save your data here
  // call any child objects to save themselves here
}
```

If the connection isn't already open a connection object is created and opened. If the connection is already open it is reused. When the last nested using block completes the connection object is automatically disposed.

DbContextManager

When using Entity Framework Core your code won't typically interact with the underlying database connection object directly. To share an open database connection you must share the EF Core context object. `Csla.Data.DbContextManager` is intended to simplify this process by managing and automatically reusing a single object context. The result is that all data access code that uses a data context object has the following structure:

```
using (var ctx = DbContextManager<EntityContextType>.GetManager("DatabaseName"))
{
  // ctx.DbContext is now an open object context
  // interact with the entity model here
}
```

If the connection isn't already open, a connection object is created and opened. If the object context is already open, it is reused. When the last `using` block completes, the context object is automatically disposed.

ObjectContextManager

When using ADO.NET Entity Framework your code won't typically interact with the underlying database connection object directly. To share an open database connection you must share the EF object context object. `Csla.Data.ObjectContextManager` is intended to simplify this process by managing and automatically reusing a single object context. The result is that all data access code that uses a data context object has the following structure:

```
using (var ctx = ObjectContextManager<EntityContextType>.GetManager("DatabaseName"))
{
  // ctx.ObjectContext is now an open object context
  // interact with the entity model here
}
```

If the connection isn't already open, a connection object is created and opened. If the object context is already open, it is reused. When the last `using` block completes, the context object is automatically disposed.

TransactionManager

If you choose to directly use ADO.NET transactions, instead of using `TransactionScope` or Enterprise Services, you need some way to make the transaction object (and its associated connection object) available to all objects participating in the transaction. Fortunately the transaction object maintains a reference to the database connection, so all that's required is to managed and automatically reuse the ADO.NET transaction object. The `Csla.Data.TransactionManager` type helps you do this. The result is that all data access code that uses a transaction object has the following structure:

```
using (var ctx = TransactionManager<SqlConnection, SqlTransaction>.GetManager("DatabaseName"))
{
  // ctx.Context is now an open data context
  // interact with the entity model here
  ctx.Commit();
}
```

If the transaction doesn't already exist, a transaction object is created and a transaction started. If the transaction is already active, it is reused. When the last `using` block completes, the data

context object is automatically disposed. At that point, if all code blocks called `Commit` the transaction is committed, otherwise the transaction is rolled back.

SafeDataReader

Most of the time, applications don't care about the difference between a `null` value and an empty value (such as an empty string or a zero)—but databases often do. When retrieving data from a database, an application needs to handle the occurrence of unexpected `null` values with code such as the following:

```
if (dr.IsDBNull(idx))
  myValue = string.Empty;
else
  myValue = dr.GetString(idx);
```

Clearly, doing this over and over again throughout the application can get tiresome. One solution is to fix the database so that it doesn't allow nulls when they provide no value, but this is often impractical for various reasons.

> Here's one of my pet peeves: allowing nulls in a column in which you care about the difference between a value that was never entered and the empty value ("", or 0, or whatever) is fine. Allowing nulls in a column where you *don't* care about the difference merely complicates your code for no good purpose, thereby decreasing developer productivity and increasing maintenance costs.

As a more general solution, CSLA .NET includes a utility class that uses `SqlDataReader` (or any `IDataReader` implementation) in such a way that you never have to worry about `null` values again. Unfortunately, the `SqlDataReader` class isn't inheritable—it can't be subclassed directly. Instead, it is wrapped using containment and delegation. The result is that your data access code works the same as always, except that you never need to write checks for `null` values. If a `null` value shows up, `SafeDataReader` will automatically convert it to an appropriate empty value.

Obviously, if you *do* care about the difference between a `null` and an empty value, you can use a regular `SqlDataReader` to retrieve the data. Starting in .NET 2.0 you can use the `Nullable<T>` generic type that helps manage null database values. This new type is valuable when you do care about `null` values: when business rules dictate that an "empty" value like `0` is different from `null`.

DataMapper

When building ASP.NET Web Forms applications, and some types of services, it is necessary to copy values from the postback or service request into your business objects. You end up writing code much like this:

```
cust.Name = e.Values["Name"].ToString();
cust.Address1 = e.Values["Address1"].ToString();
cust.City = e.Values["City"].ToString();
```

This is repetitive, boring code to write. One alternative, though it does incur a performance hit, is to use a data copy utility, or data mapper, to automate the copy process. This is the purpose of the `DataMapper` class: to automate the copying of data to reduce all those lines of code to one simple line. It is up to you whether to use `DataMapper` in your applications.

SmartDate

Dates are a perennial development problem. Of course, there are the `DateTime` and `DateTimeOffset` data types, which provide powerful support for manipulating dates, but they have no concept of an "empty" date. The trouble is that many applications allow the user to leave date fields empty, so you need to deal with the concept of an empty date within the application.

On top of this, date formatting is problematic—rather, formatting an ordinary date value is easy, but again you're faced with the special case whereby an "empty" date must be represented by an empty string value for display purposes. In fact, for the purposes of data binding, we often want any date properties on the objects to be of type `string` so that the user has full access to the various data formats as well as the ability to enter a blank date into the field.

Dates are also a challenge when it comes to the database: the date data types in the database don't understand the concept of an empty date any more than .NET does. To resolve this, date columns in a database typically *do* allow `null` values, so a `null` can indicate an empty date.

> Technically, this is a misuse of the `null` value, which is intended to differentiate between a value that was never entered, and one that's empty. Unfortunately, we're typically left with no choice, because there's no way to put an empty date value into a date data type.

You may be able to use `DateTime?` (`Nullable<DateTime>`) as a workable data type for your date values. But even that isn't always perfect, because `DateTime?` doesn't offer specialized formatting and parsing capabilities for working with dates. Nor does it understand the concept of an empty date: it isn't possible to compare actual dates with empty dates, yet that is often a business requirement.

The `SmartDate` type is an attempt to resolve this issue. Repeating the problem with `SqlDataReader`, the `DateTime` data type isn't inheritable, so `SmartDate` can't just subclass `DateTime` to create a more powerful data type. Instead, it uses containment and delegation to create a new type that provides the capabilities of the `DateTime` data type while also supporting the concept of an empty date.

Applications often need to compare an empty date to a real date, but an empty date might be considered very small or very large. The `SmartDate` class is designed to support these concepts, and to integrate with the `SafeDataReader` so that it can properly interpret a `null` database value as an empty date.

Additionally, `SmartDate` is a robust data type, supporting numerous operator overloads, casting and type conversion. Better still, it works with both `DateTime` and the new `DateTimeOffset` type.

Package, Assembly, and Namespace Organization

At this point, I've discussed some of the classes that make up CSLA .NET. Given that there are quite a few classes and types required to implement the framework, there's a need to organize them for easier discovery and use. The solution for this is to organize the types into a set of *namespaces*.

Although assemblies and namespaces aren't tightly linked, by convention it is common for assembly names to reflect the high level namespace contained within the assembly. For example,

System.Runtime.Serialization.dll contains elements in the System.Runtime.Serialization namespace.

Namespaces allow you to group classes together in meaningful ways so that you can program against them more easily. Additionally, namespaces allow different classes to have the same name as long as they're in different namespaces. From a business perspective, you might use a scheme like the following:

```
MyCompany.MyApplication.FunctionalArea.Class
```

A convention like this immediately indicates that the class belongs to a specific functional area within an application and organization. It also means that the application could have multiple classes with the same names:

```
MyCompany.MyApplication.Sales.Product
MyCompany.MyApplication.Manufacturing.Product
```

It's quite likely that the concept of a "product" in sales is different from that in manufacturing, and this approach allows reuse of class names to make each part of the application as clear and self-documenting as possible.

The same is true when you're building a framework. Classes should be grouped in meaningful ways so that they're comprehensible to the end developer. Additionally, use of the framework can be simplified for the end developer by putting little-used or obscure classes in separate namespaces. This way, the business developer doesn't typically see them via IntelliSense.

Consider the UndoableBase class, which isn't intended for use by a business developer: it exists for use within the framework only. Ideally, when business developers are working with the framework, they won't see UndoableBase via IntelliSense unless they go looking for it by specifically navigating to a specialized namespace. The framework has some namespaces that are to be used by end developers, and others that are intended for internal use.

All the namespaces in CSLA .NET are prefixed with Csla, and all assembly names start with Csla.

Table 17 lists the assemblies that make up the CSLA .NET framework along with the NuGet package that contains each assembly.

Assembly	NuGet Package	Platforms	Description
Csla.dll	CSLA-Core	.NET Standard 2.0 .NET Core 2.0 .NET Framework 4+	Core framework functionality common to all platforms and presentation technologies
Csla.Web.dll	CSLA-ASP.NET	.NET Framework 4+	Controls and other types useful in creating an ASP.NET Web Forms application
Csla.Web.Mvc.dll	CSLA-ASP.NET-MVC4	.NET Framework 4+	Controls and other types useful in creating an

			ASP.NET MVC 4 application
Csla.Web.Mvc.dll	CSLA-ASP.NET-MVC5	.NET Framework 4+	Controls and other types useful in creating an ASP.NET MVC 5 application
Csla.Web.Mvc.dll	CSLA-ASPNETCORE-MVC	.NET Core 2.0 .NET Framework 4+	Controls and other types useful in creating an ASP.NET Core application
Csla.Xaml.dll	CSLA-XamarinForms	Xamarin	Controls and other types useful in creating a Xamarin.Forms application
Csla.Axml.dll	CSLA-Android	Xamarin	Controls and other types useful in creating a Xamarin Android application
Csla.Iosui.dll	CSLA-iOS	Xamarin	Controls and other types useful in creating a Xamarin iOS application
Csla.Data.EF4.dll	CSLA-EF4	.NET Framework 4+	Types useful for using Entity Framework 4
Csla.Data.EF5.dll	CSLA-EF5	.NET Framework 4+	Types useful for using Entity Framework 5
Csla.Data.EF6.dll	CSLA-EF6	.NET Framework 4+	Types useful for using Entity Framework 6
Csla.Data.EntityFrameworkCore.dll	CSLA-EntityFrameworkCore	.NET Standard 2.0 .NET Core 2.0	Types useful for using Entity Framework Core
Csla.Xaml.dll	CSLA-WPF	.NET Framework 4+	Controls and other types useful in creating a WPF application
Csla.Xaml.dll	CSLA-UWP	UWP	Controls and other types useful in creating a UWP application
Csla.Windows.dll	CSLA-WindowsForms	.NET Framework 4+	Controls and other types useful in creating a Windows Forms application

Table 17. Assemblies and NuGet packages making up the CSLA .NET framework

Any application using CSLA .NET will need to reference the `CSLA-Core` NuGet package. Applications may also reference other packages that target other technologies as needed. For example, an ASP.NET Core application will typically reference the `CSLA-ASPNETCORE-MVC` package. This will automatically also bring in the `CSLA-Core` package.

Within the assemblies, the framework types are further organized into namespaces. Table 18 lists the namespaces used in the CSLA .NET framework.

Namespace	Description
`Csla`	Types most commonly used by business developers
`Csla.Axml`	Types that support Xamarin Android development; used by Android UI developers
`Csla.Configuration`	Types that abstract differences between configuration in .NET Framework and .NET Standard; should be used by business developers instead of using either platform-specific implementation
`Csla.Core`	Core functionality for the framework; not intended for regular use by business developers
`Csla.Core.FieldManager`	Field manager types used to implement properties in business objects
`Csla.Core.LoadManager`	Types that implement asynchronous loading of property values
`Csla.Core.TypeConverters`	Type converter for SmartDate
`Csla.Data`	Optional types used to support data access operations; often used by business developers, web UI developers, and Web Service developers
`Csla.DataPortalClient`	Types that support the client-side `DataPortal` behaviors; used when creating a custom data portal proxy
`Csla.Iosui`	Types that support Xamarin iOS development; used by iOS UI developers
`Csla.Properties`	Code generated by Visual Studio for the `Csla` project; not intended for use by business developers
`Csla.Reflection`	Types that abstract and enhance the use of reflection and dynamic method invocation
`Csla.Rules`	Types supporting business rules; often used when creating business, validation, and authorization rules
`Csla.Security`	Types supporting authentication; used when creating custom principal and identity types
`Csla.Serialization`	Abstracts the use of the `MobileFormatter` and the .NET `BinaryFormatter` or `NetDataContractSerializer` serialization technologies
`Csla.Serialization.Mobile`	Implementation of the `MobileFormatter`

Csla.Server	Contains the types supporting the server-side data portal behaviors; not intended for use by business developers
Csla.Server.Dashboard	Implementation of the optional data portal dashboard used to expose data portal activity and diagnostic information
Csla.Server.Hosts	Types supporting server-side data portal hosts; used to access default host implementations or when creating a custom data portal host
Csla.Threading	Types that simplify some threading scenarios
Csla.Web	Types that support ASP.NET Web Forms and MVC 2 development; used by web UI developers
Csla.Web.Design	Contains the supporting types for the CslaDataSource control; not intended for use by business developers
Csla.Web.Security	Contains an IdentityFactory type, used to support the loading of membership identity objects for use by smart client applications
Csla.Web.Mvc	Types that support ASP.NET MVC development; used by web UI developers
Csla.Windows	Contains controls to assist with Windows Forms data binding; used by Windows Forms UI developers
Csla.Xaml	Types that support XAML development (Xamarin.Forms, WPF, and UWP); used by UI developers

Table 18. Namespaces Used in the CSLA .NET Framework

The primary base classes intended for use by business developers are found in the Csla namespace itself. They are named as follows:

- Csla.BusinessBase<T>
- Csla.BusinessListBase<T,C>
- Csla.BusinessBindingListBase<T,C>
- Csla.DynamicListBase<T,C>
- Csla.DynamicBindingListBase<T,C>
- Csla.ReadOnlyBase<T>
- Csla.ReadOnlyListBase<T,C>
- Csla.ReadOnlyBindingListBase<T,C>
- Csla.NameValueListBase<K,V>
- Csla.CommandBase<T>

The rest of the classes and types in the framework are organized into the remaining namespaces based on their purpose.

The end result is that a typical business developer can use the `Csla` namespace as follows:

```
using Csla;
```

And all they'll see are the classes intended for use during business development. All the other classes and concepts within the framework are located in other namespaces, and therefore won't appear in IntelliSense by default, unless the developer specifically imports those namespaces.

When using custom authentication, you'll likely import the `Csla.Security` namespace. But if you're not using that feature, you can ignore those classes and they won't clutter up the development experience. Similarly, `Csla.Configuration`, `Csla.Data` and `Csla.Rules` may be used in some cases. If the types those namespaces contain are useful, they can be brought into a class with a `using` statement; otherwise, they are safely out of the way.

Conclusion

This book has examined some of the key design goals for the CSLA .NET framework. The key design goals include the following:

- Validation and maintaining a list of broken business rules
- Standard implementation of business rules
- Integrated authorization rules at the object and property levels
- Tracking whether an object's data has changed
- Strongly typed collections of child objects (parent-child relationships)
- N-level undo capability
- A simple and abstract model for the UI developer
- Full support for data binding in all .NET interface technologies
- Saving objects to a database and getting them back again
- Custom authentication
- Enable appropriate extensibility

I've also provided a high level walk through the design of the framework itself, providing a glimpse into the purpose and rationale behind each of the classes that make up the framework. With each class, I discussed how it relates back to the key goals to provide the features and capabilities of the framework.

The chapter closed by defining the assemblies and namespaces that contain the framework classes. This way, they're organized so that they're easily understood and used.

This is the first of a series of books. Subsequent books will dive deeper into many aspects of the framework, taking the high level concepts from this book and showing you how to apply them in your application development projects.