

# **Expert VB 2005 Business Objects**

**Second Edition**



**Rockford Lhotka**

## **Expert VB 2005 Business Objects, Second Edition**

**Copyright © 2006 by Rockford Lhotka**

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 and the publisher.

ISBN-13 (pbk): 978-1-59059-631-9

ISBN-10 (pbk): 1-59059-631-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

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.

Lead Editor: Jonathan Hassell

Technical Reviewer: Petar Kozul

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick,  
Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft,  
Jim Sumser, Keir Thomas, Matt Wade

Project Manager: Kylie Johnston

Copy Edit Manager: Nicole LeClerc

Copy Editor: Damon Larson

Assistant Production Director: Kari Brooks-Copony

Production Editor: Laura Cheu

Compositor: Linda Weidemann, Wolf Creek Press

Proofreader: April Eddy

Indexer: John Collin

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail [orders-ny@springer-sbm.com](mailto:orders-ny@springer-sbm.com), or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail [info@apress.com](mailto:info@apress.com), or visit <http://www.apress.com>.

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, neither the author(s) nor Apress shall 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 is available to readers at <http://www.apress.com> in the Source Code section.



# Object-Oriented Application Design

**C**hapters 1 and 2 discussed the concepts behind distributed, object-oriented systems, and the .NET technologies that make them practical to implement with reasonable effort. Then, Chapters 3 through 5 covered the design and implementation of CSLA .NET, a framework upon which you can build distributed, object-oriented applications; thereby avoiding the complexities of the underlying technologies while creating each business class or user interface.

Chapter 7 will discuss the basic structure of business objects based on CSLA .NET. Chapter 8 will put that knowledge to use to implement a set of sample business objects for an application to track projects and resources assigned to projects. Chapter 9 will walk through the implementation of a Windows Forms UI, and in Chapter 10, a Web Forms UI will be implemented based on these objects. Chapter 11 will discuss the creation of a Web Services interface so the business objects can be used by other applications through the standard SOAP protocol.

This chapter will focus on the object-oriented application design process, using a sample scenario and application that will be implemented through the rest of the book. The design process in this chapter will result in a design for the business objects, and for an underlying database.

Obviously, the challenge faced in designing and building a sample application in a book like this is that the application must be small enough to fit into the space available, and yet be complex enough to illustrate the key features I want to cover. To start with, here's a list of the key features that I want to focus on:

- Creation of a business object
- Implementation of business validation rules
- Implementation of business authorization rules
- Transactional and nontransactional data access
- Parent-child relationships between objects
- Many-to-many relationships between objects
- Use of name/value lists
- Use of custom CSLA .NET authentication

In this chapter, I'll focus on the design of the application by using some example user scenarios, which are generally referred to as *use cases*. Based on those use cases, I'll develop a list of potential business objects and relationships. This information will be refined to develop a class design for the application. Based on the scenarios and object model, a relational database will be designed to store the data.

As I mentioned in Chapter 2, object-oriented design and relational design aren't the same process, and you'll see in this case how they result in two different models. To resolve these models, the business objects will include object-relational mapping (ORM) when they are implemented in

Chapter 8. This ORM code will reside in the `DataPortal_XYZ` methods of the business objects, and will translate the data between the relational and object-oriented models as each object is retrieved or updated.

## Application Requirements

There are many ways to gather application requirements, but in general there are three main areas of focus from which you can choose:

- Data analysis and data flow
- UI design and storyboarding
- Business concept and process analysis

The oldest of the three is the idea that an application can be designed by understanding the data it requires, and how that data must flow through the system. While this approach can work, it isn't ideal when trying to work with object-oriented concepts, because it focuses less on business ideas and more on raw data. It's often a very good analysis approach when building applications that follow a data-centric architecture.

---

**Note** The data-focused analysis approach often makes it hard to relate to users well. Very few users understand database diagrams and database concepts, so there's a constant struggle as the business language and concepts are translated into and out of relational, data-oriented language and concepts.

---

The idea of basing application analysis around the UI came into vogue in the early-to-mid 1990s with the rise of rapid application development (RAD) tools such as Visual Basic, PowerBuilder, and Delphi. It was subsequently picked up by the web development world, though in that environment, the term “storyboarding” was often used to describe the process. UI-focused analysis has the benefit of being very accessible to the end user—users find it very easy to relate to the UI and how it will flow.

The drawback to this approach is that there's a tendency for business validation and processing to end up being written directly into the UI. Not that this *always* happens, but it's a very real problem—primarily because UI-focused analysis frequently revolves around a UI prototype, which includes more and more business logic as the process progresses, until developers decide just to use the prototype as the base for the application, since so much work has already been done.

---

**Tip** Obviously, people can resist this trend and make UI-focused design work, but it takes a great deal of discipline. The reality is that a lot of great applications end up crippled because this technique is used.

---

Another drawback to starting with the UI is that users often see the mocked-up UI in a demonstration and assume that the application is virtually complete. They don't realize that the bulk of the work comes from the business and data access logic that must still be created and tested *behind* the UI. The result is that developers are faced with tremendous and unrealistic time pressure to deliver on the application, since from the user's perspective, it's virtually complete already.

The third option is to focus on business concepts and process flow. This is the middle road in many ways, since it requires an understanding of how the users will interact with the system, the processes that the system must support, and (by extension) the data that must flow through the system to make it all happen. The benefit of this approach is that it's very business focused, allowing both the analyst and the end users to talk the language of business, thereby avoiding computer

concepts and terminology. It also lends itself to the creation of object-oriented designs, because the entities and concepts developed during analysis typically turn into objects within the application.

The drawback to this approach is that it doesn't provide users with the look and feel of the UI, or the graphical reinforcement of how the system will actually work from their perspective. Nor does it produce a clear database design, thereby leaving the database analyst to do more work in order to design the database.

Personally, I use a blend of the business concept and UI approaches. I place the strongest emphasis on the business concept and process flow, while providing key portions of the UI via a prototype, so that the user can get the feel of the system. Since end users have such a hard time relating to database diagrams, I almost never use data-focused analysis techniques, instead leaving the database design process to flow from the other analysis techniques.

In this chapter, I'll make use of the business concept and process-flow techniques. It's difficult to storyboard the application at this stage, because we'll be developing both Windows Forms and Web Forms user interfaces, along with a web service application interface. The starting point, then, is to create a set of use case descriptions based on how the users (or other applications) will interact with the system.

## Use Cases

Let's create a set of imaginary use cases for the project-tracking system. In a real application, these would be developed by interviewing key users and other interested parties. The use cases here are for illustration purposes.

---

**Tip** This application is relatively simple. A real project-tracking system would undoubtedly be more complex, but it is necessary to have something small enough to implement within the context of this book. Remember that my focus is on illustrating how to use CSLA .NET to create business objects, child objects, and so forth.

---

Though not mentioned specifically in the following use cases, this system will be designed to accommodate large numbers of users. In Chapter 9, for instance, the Windows Forms UI will use the mobile object features of CSLA .NET to run the application in a physical n-tier deployment with an application server. This physical architecture will provide for optimum scalability. In Chapter 10, the Web Forms UI will make use of the CSLA .NET framework's ability to run the application's UI, business logic, and data access all on the web server. Again, this provides the highest-scaling and best-performing configuration, because you can easily add more web servers as needed to support more users.

## Project Maintenance

Since this is a project-tracking system, there's no surprise that the application must work with projects. Here are some use cases describing the users' expectations.

### Adding a Project

A project manager can add projects to the system. Project data must include key information, including the project's name, description, start date, and end date. A project can have a unique project number, but this isn't required, and the project manager shouldn't have to deal with it. The project's name is the field by which projects are identified by users, so every project must have a name.

The start and end dates are optional. Many projects are added to the system so that a list of them can be kept, even though they haven't started yet. Once a project has been started, it should

have a start date, but no end date. When the project is complete, the project manager can enter an end date. These dates will be used to report on the average lengths of the projects, so obviously the end date can't be earlier than the start date.

Every project also has a list of the resources assigned to it (see the “Assigning a Resource” section later in this chapter).

### **Editing a Project**

Project managers can edit any existing projects. The manager chooses from a list of projects, and can then edit that project. They need the ability to change the project's start and end dates, as well as its description. They also need to be able to change the resources assigned to the project (see the “Assigning a Resource” section later in this chapter).

### **Removing a Project**

Project managers or administrators must be able to remove projects. There is no need to keep historical data about deleted projects, so such data should be completely removed from the system. The user should just choose from a list of projects, confirm his choice, and the project should be removed.

### **Resource Maintenance**

At this point, the system not only tracks projects, but also tracks the resources assigned to each project. For the purposes of this simple example, the only project resources tracked are the people assigned to the projects. With further questioning of the users, a set of use cases revolving around the resources can be developed, without reference (yet) to the projects in which they may be involved.

### **Adding a Resource**

We don't want to replicate the Human Resources (HR) database, but we can't make use of the HR database because the HR staff won't give us access. We just want to be able to keep track of the people we can assign to our projects. All we care about is the person's name and employee ID. Obviously, each person must have an employee ID and a valid name.

Resources can be added by project managers or supervisors. It would be really nice to be able to assign a person to a project at the same time as the person is being added to the application (see the “Assigning a Resource” section later in this chapter).

### **Editing a Resource**

Sometimes, a name is entered incorrectly and needs to be fixed, so project managers and supervisors need to be able to change the name.

### **Removing a Resource**

When an employee is let go or moves to another division, we want to be able to remove him from the system. Project managers, supervisors, and administrators should be able to do this. Once they're gone, we don't need any historical information, so they should be totally removed.

### **Assigning a Resource**

As we were talking to the users to gather information about the previous use cases, the users walked through the requirements for assigning resources to projects. Since this process is common across several other processes, we can centralize it into a use case that's referenced from the others.

The project managers and supervisors need to be able to assign a resource to a project. When we do this, we need to indicate the role that the resource is playing in the project. We have a list of the roles, but we might need to change the list in the future. We also want to know when the resource was assigned to the project.

Sometimes, a resource will switch from one role to another, so we need to be able to change the role at any time. Equally, a resource can be assigned to several projects at one time. (We often have people working part-time on several projects at once.)

Lastly, we need to be able to remove an assignment. This happens when an employee is let go or moves to another division (see the “Removing a Resource” section earlier in this chapter); but we also often move people around from project to project. There’s no need to keep track of who used to be on a project, because we only use this system for tracking current projects and the resources assigned to them right now.

## Maintaining a List of Roles

Resources are assigned to projects to fill a specific role. The list of possible roles needs to be maintainable by end users: specifically administrators.

## External Access

During conversations with users, we discovered that a number of them are highly technical, and are already skeptical of our ability to create all the UI options they desire. They indicated high interest in having programmatic access to the database, or to our business objects. In other words, we have some power users who are used to programming in Access and know a bit of VBA, and they want to write their own reports, and maybe their own data entry routines.

---

**Tip** This same scenario would play out if there’s a requirement to provide access to the application to business partners, customers, vendors, or any external application outside our immediate control.

---

Obviously, there are serious issues with giving other people access to the application’s database—especially read-write access. Unless *all* the business logic is put into stored procedures, this sort of access can’t be safely provided.

Likewise, there are issues with providing direct access to the business objects. This is safer in some ways, because the objects implement the business logic and validation; but it’s problematic from a maintenance perspective. If other people are writing code to interact directly with the business objects, then the objects can’t be changed without breaking their code. Since the other people are outside of our control, it means that the project tracker application can never change its object model.

Of course, this is totally unrealistic. It is a virtual guarantee that there will be future enhancements and requests for changes to the system, which will undoubtedly require changes to the business objects. Fortunately, Web Services offers a clean solution. If web services are treated just like any another interface (albeit a programmatic one) to the application, they can be used to easily provide access to the application without allowing external programs to directly interact with the application’s database or business objects.

In Chapter 11, I’ll revisit these ideas, showing how to implement a set of web services so that external applications can safely interact with the application in a loosely coupled manner.

# Object Design

At this point, the key requirements for the application have been gathered from the use cases. Based on these use cases, it is possible to create an object-oriented design. There are a variety of techniques used in object-oriented design (you may have heard of CRC cards and decomposition, in addition to others), and in this chapter, I'll use ideas from both decomposition and CRC cards. A form of decomposition will be used to identify the “nouns” in the use cases, and then narrow down which of these are actual business objects. These objects will be described in terms of their class, responsibility, and collaborators (CRC).

## Initial Design

The first step in the process, then, is to assemble a list of the nouns in the use case write-ups. By using a bit of judgment, you can eliminate a few nouns that are obviously not objects, but still end up with a good-sized list of potential business objects or entities, as shown in Table 6-1.

**Table 6-1.** *Potential Entities Discovered in the Initial Design*

Project manager	Project	Project number
Project name	Start date	End date
Administrator	List of projects	Employee
Resource	Employee name	Employee ID
Supervisor	List of assignments	Role
List of roles	Assignment	Date assigned
List of resources	List of assigned resources	

Using your understanding of the business domain (and probably through further discussion with business users and fellow designers), the options can be narrowed. Some of these aren't objects, but rather data elements or security roles. These include the following:

- Project manager
- Administrators
- Supervisor

**Tip** I am assuming there's already an object to deal with a user's role. Such an object will be created by subclassing the `Csla.Security.BusinessPrincipalBase` class later in the chapter. But these security roles should not be confused with the role a resource (person) plays on a project—they're two very different concepts.

Pulling out these nouns, along with those that are likely to be just data fields (such as project name and employee ID), you can come up with a smaller list of likely business objects, allowing you to start creating a basic class diagram or organizing the classes using CRC cards. Table 6-2 lists the high-level CRC data for each potential object.



**Table 6-2.** *Potential Objects and Their Associated Class Names*

Potential Class	Responsibility	Collaborators
Project	Adds and edits a valid project	ProjectResources
Resource	Adds and edits a valid resource	ResourceAssignments, Employee
Employee	Adds and edits a valid employee	None
ProjectList	Gets a read-only list of projects	Project
ResourceList	Gets a read-only list of resources	Resource
ProjectResources	Maintains a list of resources assigned to a project	Resource, RoleList
ResourceAssignments	Maintains a list of projects to which a resource is assigned	Project, RoleList
RoleList	Gets a read-only list of roles	Role
Role	Provides read-only role data	None
RoleEditList	Maintains a list of roles in the system	RoleEdit
RoleEdit	Adds and edits a valid role	None

One key aspect of CRC-based design is that an object's responsibility should be short and to the point. Long, complex responsibility descriptions are an indication that the object model is flawed, and that the complicated object should probably be represented by a set of simpler objects that collaborate to achieve the goal.

The diagram should also include relationships between the entities in the diagram. For the most part, these relationships can be inferred from the use case descriptions—for instance, we can infer that a “list of projects” will likely contain `Project` objects; and that a `Project` object will likely contain a “list of assigned resources,” which in turn will likely contain `Resource` objects.

Note that I use the word *likely* here, rather than *will*. We're still very much in a fluid design stage here, so nothing is yet certain. We have a list of potential objects, and we're inferring a list of potential relationships.

Figure 6-1 is an illustration of how these objects relate to each other.

Looking at the CRC list and this diagram, there is some indication that there's more work to do. There are several issues that you should look for and address, including duplicate objects, trivial objects, objects that have overly complex relationships in the diagram, and places that can be optimized for performance.

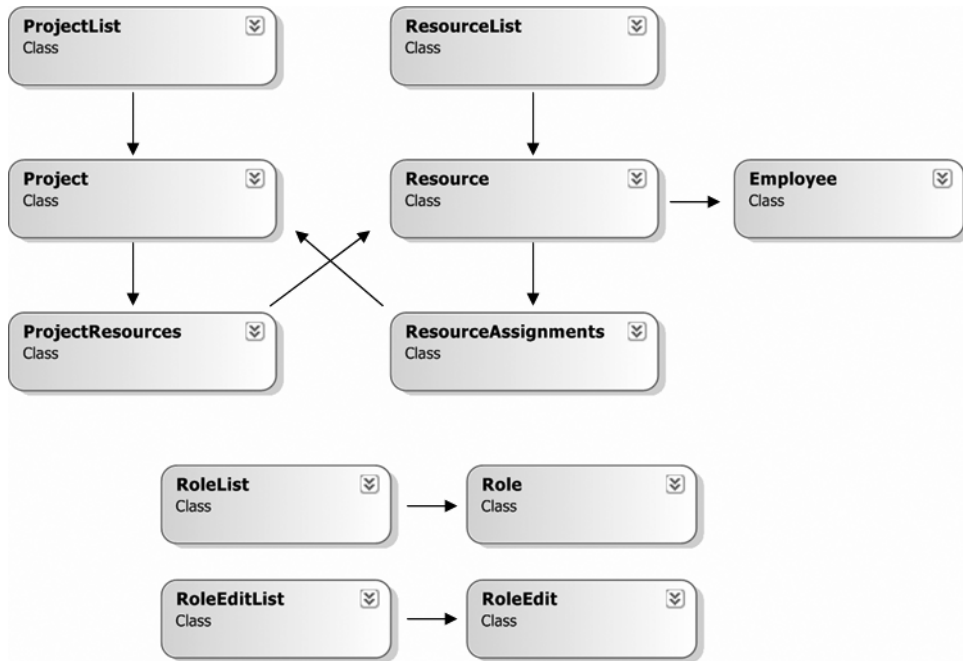


Figure 6-1. Possible class diagram for the project tracker application

## Revising the Design

The following list indicates some of the things to address:

- Resource and Employee could be duplicates. It isn't clear that Resource adds anything to Employee, so the two can probably be merged into one class.
- Based on the use case description, we know that RoleList is a name/value list, which directly implies the Role is just a name/value placeholder. Given `Csla.NameValueListBase`, this can be simplified.
- The relationship between Project, ProjectResources, Resource, and ResourceAssignments is very complex. In fact, it forms a loop of references, which is always a danger sign.
- The RoleList object isn't used by any other objects in the model. Given that the use cases indicate that resources are assigned to projects based on a specific role, this is suspicious.
- The use cases for ProjectList and ResourceList indicate that they're primarily used for selection of objects, not for editing all the projects or resources in the system. Actually loading all the Project or Resource objects just so that the user can make a simple selection is expensive, performance-wise, so this design should be reviewed.
- It is clear that when the list of roles is edited, any RoleList objects need to know about the changes so that they can read the new data. This is not explicitly stated in a use case, but is an inferred requirement.

In the early stages of *any* object-design process, there will be duplicate objects, or potential objects that end up being mere data fields in other objects. Usually, a great deal of debate will ensue during the design phase as all the people involved in the design process thrash out which objects

are real, which are duplicates, and which should be just data fields. This is healthy and important, though obviously some judgment must be exercised to avoid *analysis paralysis*, whereby the design stalls entirely due to the debate.

Let's discuss this in a bit more detail.

## Duplicate Objects

First, you should identify duplicate objects that have basically the same data and relationships (like Resource and Employee). In this case, Employee can be eliminated in favor of Resource, since that's the term used most often in the use case descriptions (and thus, presumably, most used by the end users).

In most scenarios, the end users will have numerous terms for some of their concepts. It's your job, as part of the analysis process, to identify when multiple terms really refer to the same concepts (objects) and to clarify and abstract the appropriate meaning.

## Trivial Objects

The Role object may not be required either. Fundamentally, a Role is just a string value, presumably with an associated key value. This is the specific scenario for which the NameValueListBase class in the CSLA .NET framework is designed. That base class makes it easy to implement name/value lists.

---

**Tip** My characterization of the Role value is based on the use cases assembled earlier. If you intuitively feel that this is overly simplistic or unrealistic, then you should revisit the use cases and your users to make sure that you haven't missed something. For the purposes of this book, I'll assume that the use cases are accurate, and that the Role field really is a simple name/value pair.

---

Note that I'm not suggesting elimination of the RoleEdit class. While NameValueListBase can be used to create read-only name/value lists, RoleEdit and RoleEditList are used to *edit* the role data. They can't be automated away like a simple name/value pair.

Like the process of removing duplicates, the process of finding and removing trivial objects is as much an art as it is a science. It can be the cause of plenty of healthy debate!

## Overly Complex Relationships

Although it's certainly true that large and complex applications often have complex relationships between classes and objects, those complex relationships should always be carefully reviewed.

As a general rule, if relationship lines are crossing each other or wrapping around each other in a diagram like Figure 6-1, you should review those relationships to see if they need to be so complex. Sometimes, it's just the way things have to be, but more often, this is a sign that the object model needs some work. Though relying on the aesthetics of a diagram may sound a bit odd, it is a good rule of thumb.

In this case, there's a pretty complex relationship between Project, ProjectResources, Resource, and ResourceAssignments. It is, in fact, a circular relationship, in which all these objects refer to the other objects in an endless chain. In a situation like this, you should always be looking for a way to simplify the relationships. What you'll often find is that the object model is missing a class: one that doesn't necessarily flow directly from the use cases, but is required to make the object model workable.

The specific problem caused by the circular relationship in Figure 6-1 becomes very apparent when an object is to be loaded from the database. At that time it will typically also load any child objects it contains. With an endless loop of relationships, that poses a rather obvious problem!

There must be some way to short-circuit the process, and the best way to do this is to introduce another object into the mix.

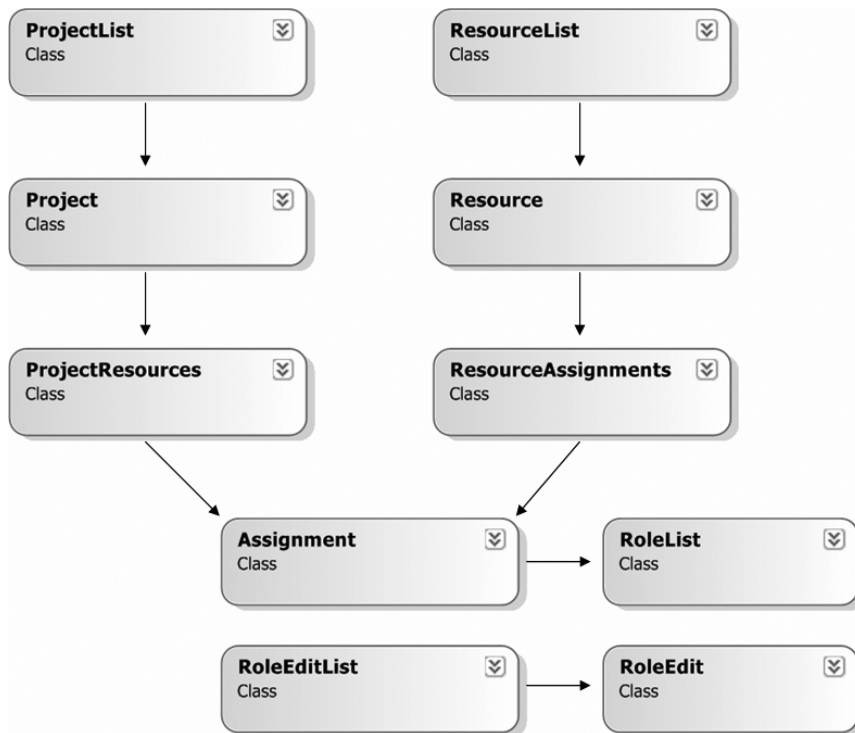
In the object model thus far, what's missing is a class that actually represents the assignment of a resource to a project. At this point, there's no object responsible for assigning a resource to a project, so there's an entire behavior from the use cases that's missing in the object model.

Additionally, there's data described in the use cases that isn't yet reflected in the object model, such as the role of a resource on a particular project, or the date that the resource was assigned to a project. These data fields can't be kept in the `Project` object, because a project will have many resources filling many different roles at different times. Similarly, they can't be kept in the `Resource` object, because a resource may be assigned to many projects at different times and in different roles.

### Adding an Assignment Class

The need for another object—an `Assignment` object—is clear. This object's responsibility is to *assign a resource to a project*.

Figure 6-2 shows an updated diagram, including the changes thus far.



**Figure 6-2.** Revised class diagram for the project tracker application

However, we're still not done. The `Assignment` class itself just became overly complex, because it's used within two different contexts: from the list of resources assigned to a project, and from the list of projects to which a resource is assigned. This is typically problematic. Having a single object as a child of two different collections makes for very complicated implementation and testing, and should be avoided when possible.

Beyond that, think about its responsibility in the diagram in Figure 6-2. Assignment is now responsible for *assigning a resource to a project AND for associating a project with a resource*. When used from ProjectResources, it has the first responsibility, and when used from ResourceAssignments, it has the second responsibility. Sure, the responsibilities are similar, but they are different enough that it matters.

There's also an issue with data. A Project object uses the ProjectResources collection to get a list of resources assigned to the project. This implies that the Assignment object contains information about the resource assigned to the project.

Yet a Resource object uses the ResourceAssignments collection to get a list of projects to which the resource is assigned. This implies that the Assignment object contains information about the project to which the resource is assigned.

The fact that both behavioral and data conflicts exist means that the object model remains flawed.

There are two possible solutions. The list objects (ProjectResources and ResourceAssignments) could be combined into a single list of Assignment objects, or there could be two different objects representing assignments. To resolve this, we need to think about the different behaviors that are required when approaching the concept of assignments from Project and from Resource.

### Assigning a Resource to a Project

Based on the use cases, resources can be assigned to projects. This implies that the user has identified the project and wishes to assign a resource to it. It also implies that a project has a collection of assigned resources: hence the ProjectResources collection in the object model.

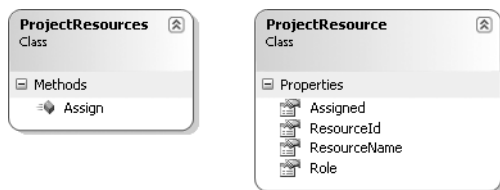
But what behavior and information would a user expect from the items in the ProjectResources collection?

Certainly, one behavior is to return the list of resources assigned to the project. Another behavior is to allow a new resource to be assigned to the project, implying something like an Assign() method that accepts the Id property from a Resource.

It is also worth considering what information should be provided to the user. When viewing or editing a Project, the list of assigned resources should probably show something like this:

- Resource ID
- Resource name
- Date assigned to the project
- Role of the resource on the project

This means that ProjectResources, and the items returned by ProjectResources, might look something like Figure 6-3.



**Figure 6-3.** The ProjectResources collection and the ProjectResource child object

Though not visible in Figure 6-3, the `Assign()` method accepts a `resourceId` parameter to identify the resource being assigned to the project.

Given this analysis, let's consider the behaviors and information required to assign a project to a resource—basically the same process, but starting with a `Resource` instead of a `Project`.

### Assigning a Project to a Resource

The use cases provide for the idea that a user could start by identifying a resource rather than a project. In this case, the user can still associate a project with the resource by selecting a project. This implies that the `Resource` object has a collection of projects to which the resource is assigned. The object model thus far represents this collection as `ResourceAssignments`.

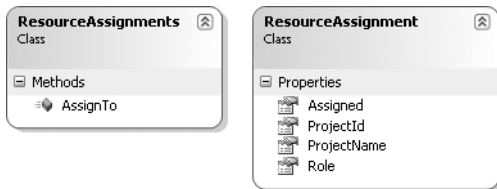
Let's consider the behaviors and information for the `ResourceAssignments` collection and the items it would contain.

In this case, the user starts with a `Resource` and wishes to assign the resource to a project. So the `ResourceAssignments` object will have a couple behaviors: listing the projects to which the resource is assigned, and assigning the resource to a new project. This can probably be handled by an `AssignTo()` method that accepts the `Id` property of a `Project`.

The items in `ResourceAssignments` have the behavior of returning information about the project assigned to the resource. The information of value to a user is likely the following:

- Project ID
- Project name
- Date assigned to the project
- Role of the resource on the project

Figure 6-4 shows the potential `ResourceAssignments` object and what its items might look like.



**Figure 6-4.** *The ResourceAssignments collection and the ResourceAssignment child object*

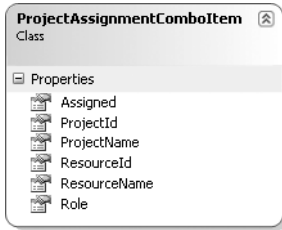
The `AssignTo()` method accepts a `projectId` parameter to identify the project to which the resource should be assigned.

### Can the Classes Be Merged?

It is important to notice that the objects described by Figure 6-3 and Figure 6-4 are *similar*, but they are not the same. Yet they do share at least some common information, if not behavior. Both child classes contain `Assigned` and `Role` properties, implying that there's commonality between them.

Such commonality is *not* justification for combining the two classes into one, because their behaviors are distinctly different. The items in `ProjectResources` have one responsibility: managing information about a resource assigned to a project. The items in `ResourceAssignments` have a different responsibility: managing information about a project to which a resource is assigned.

While this difference may seem subtle, it is a difference nonetheless. It is tempting to consider that the two classes could be merged into one, as shown in Figure 6-5.



**Figure 6-5.** Merged child items with assignment information

Of course, `ProjectName` isn't valid if the user got to this object from a `Project` object, but it is valid if she got here through a `Resource` object. The same is true for several other properties.

Perhaps business logic could be added to properties to throw exceptions if they were called from an inappropriate context. But the obvious complexity of this sort of logic should give you pause. The problem is that one object is trying to handle more than one responsibility. Such a scenario means that the object model is flawed. Going down such a path will lead to complex, hard-to-maintain code.

---

**Note** Historically, this sort of complex code was referred to as spaghetti code. It turns out that with improper object design, it is *very* possible to end up with spaghetti code in business objects. The result is terrible, and is exactly what *good* object design is intended to prevent!

---

It should be quite clear at this point that merging the two collections or their child objects into a single set of objects isn't the right answer. They have different responsibilities, and so they should be separate objects.

But this leaves one glaring issue: what about the common properties and any common business logic they might require? How can two objects use the same data without causing duplication of business logic?

### Dealing with Common Behaviors and Information

When designing relational databases, it is important to normalize the data. There are many aspects to normalization, but one of the most basic and critical is avoiding redundant data. A given data element should exist *exactly once* in the data model. And that's great for relational modeling.

Unfortunately, many people struggle with object design because they try to apply relational thinking to objects. But object design is *not the same* as relational design. Where the goal with relational design is to avoid duplication of data, the goal of object design is quite different.

There's no problem with a data field being used or exposed by different objects. I realize this may be hard to accept. We've all spent so many years being trained to think relationally that it's often very hard to break away and think in terms of objects. Yet creating a good object model *requires* changing this mode of thought.

---

**Caution** Object design isn't about normalizing data. It is about normalizing *behavior*.

---

The goal in object design is to ensure that a given *behavior* exists only once within the object model. Simple examples of behavior include the idea of a string being required, or one value being larger than another. More complex behaviors might be the calculation of a tax or discount amount.

Each behavior should exist only once in the object model, though it may be *used* from many different objects.

This is why collaboration is so critical to good object design. For example, one object—the `DiscountCalculator`—will implement the complex calculation for a discount. Many other objects may need to determine the discount, and so they collaborate with `DiscountCalculator` to find that value. In this manner, the behavior exists exactly once in the model.

### Dealing with Common Information

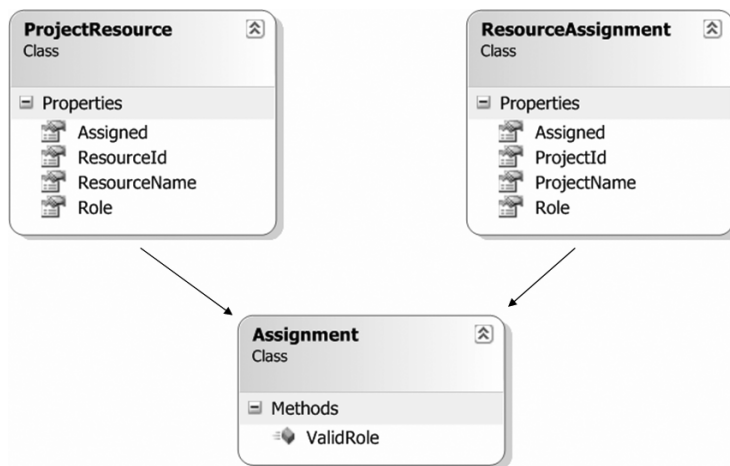
So the real question isn't whether the `Assigned` and `Role` *properties* can be put into a common object—that's relational thinking. Instead, the question is whether those properties have common *behaviors* (business rules or logic) that can be put into a common object.

As it turns out, the `Role` property must be validated to ensure that any new value is a real role. Since the `Role` property can be set in both `ProjectResource` and `ResourceAssignment`, that behavior could be duplicated.

A better answer is to normalize that behavior, putting it into a central object. Let's call this new object `Assignment`, since it will be responsible for centralizing the code common to assignments of projects to resources, and resources to projects. Then both `ProjectResource` and `ResourceAssignment` can collaborate with `Assignment` to ensure that the `Role` property is validated.

This means that `Assignment` will contain the rule method that implements the role-validation behavior. In Chapter 3, the CSLA .NET framework defined the `RuleHandler` delegate to support exactly this type of scenario.

Given a `ValidRole()` rule method in `Assignment`, both `ProjectResource` and `ResourceAssignment` merely have to associate that rule method with their `Role` properties to share the common behavior. Figure 6-6 illustrates this relationship.



**Figure 6-6.** *ProjectResource and ResourceAssignment collaborating with Assignment*

The code to do exactly this is in Chapter 8.

### Dealing with Common Behaviors

The responsibility of the `Assignment` object from Figure 6-6 is to manage the association between a project and resource.



This means that the `Assignment` object's behavior could include the idea of associating a project with a resource. This is a broader behavior than that provided by `ProjectResources`, which assigns a resource to a project; or by `ResourceAssignments`, which assigns a project to a resource. In fact, the behavior of `Assignment` is more general, and encompasses the needs of both other objects.

Of course, the real work of dealing with a resource assigned to a project, or a project associated with a resource, is handled by the `ProjectResource` and `ResourceAssignment` classes. The collection classes really just add and remove these child objects, leaving it to the child objects to handle the details.

The end result is that `ProjectResource`, to fulfill its behavior, can ask `Assignment` to do the actual work, as shown in Figure 6-7. The same is true of `ResourceAssignment`. The implication is that `Assignment` could have a method such as `AddAssignment()` that accepts a project's `Id` property and a resource's `Id` property, along with the role the resource will play on the project.

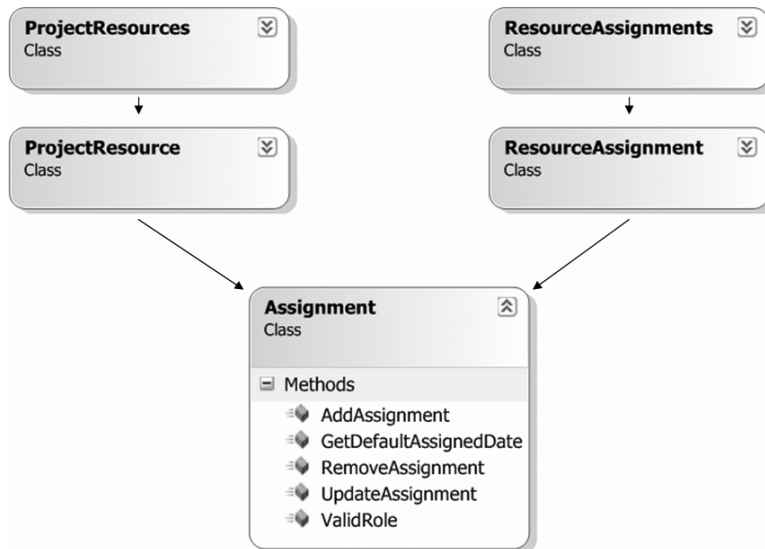
---

**Tip** Object models should be simple and intuitive, even when underlying behaviors are complex. By centralizing common behaviors using objects internal to the business layer, a simpler and more tailored public interface can be exposed to the UI developer.

---

Similarly, `ProjectResource` and `ResourceAssignment` have behaviors that involve removing a resource from a project or removing a project from a resource. `Assignment`, then, will have a more general behavior to remove an association between a project and a resource.

Figure 6-7 shows the full extent of `Assignment`, including all the methods that implement behaviors common to both `ProjectResource` and `ResourceAssignment`.



**Figure 6-7.** *Objects collaborating with Assignment*

At this point, all the common behaviors from `ProjectResource` and `ResourceAssignment` have been normalized into a single location in the object model.

## Optimizing for Performance

Part of object design includes reviewing things to ensure that the model won't lead to poor performance. This isn't really a single step in the process, as much as something that should be done on a continual basis during the whole process. However, once you think the object model is complete, you should always pause to review it for performance issues.

One primary performance issue with many object models deals with the use of relational thinking when designing the objects. Normalizing data within the object model is perhaps the most common flaw causing performance issues. Due to the design of `ProjectResource`, `ResourceAssignment`, and `Assignment`, the object model has already eliminated this issue by normalizing behavior instead of data. This helps avoid loading entire business objects just to display a couple of common data elements.

There is, however, another performance issue in the model. The `ProjectList` and `ResourceList` collection objects, as modeled, retrieve collections of `Project` and `Resource` business objects so that some of their data can be displayed in a list. Based on the use cases, the user then selects one of the objects and chooses to view, edit, or remove that object.

From a purely object-oriented perspective, it's attractive to think that you could just load a collection of `Project` objects and allow the user to pick the one he wants to edit. However, this could be very expensive, because it means loading all the data for *every* `Project` object, including each project's list of assigned resources, and so forth. As the user adds, edits, and removes `Project` objects, you would potentially have to maintain your collection in memory too.

Practical performance issues dictate that you're better off creating a read-only collection that contains only the information needed to create the user interface. (This is one of the primary reasons why CSLA .NET includes the `ReadOnlyListBase` class, which makes it very easy to create such objects.)

This stems from behavioral design. The responsibility of a `Resource` object is to add and edit a valid resource. The responsibility of a `ResourceList` object is to get a read-only list of resources. It is clear that these responsibilities are in conflict. To use a `Resource` object as a child of `ResourceList`, it would need to be read-only—yet its whole purpose is to add and edit data!

Obviously `ResourceList` and `ProjectList` must contain child objects other than `Resource` and `Project`. Instead, the `ProjectList` and `ResourceList` objects should contain child objects that contain only the data to be displayed, in read-only format. These new child objects will have responsibilities appropriate to their purpose. `ResourceInfo`, for instance, will be responsible for returning read-only information about a resource.

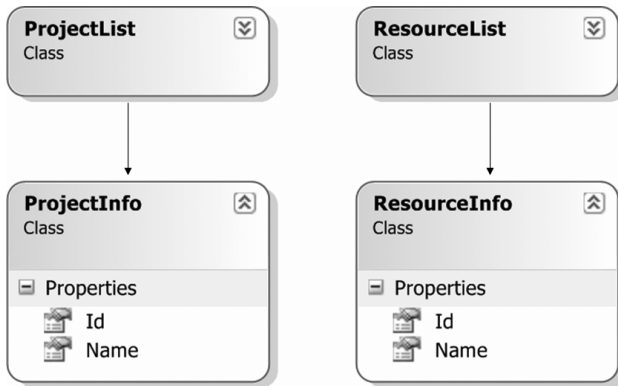
---

**Tip** As discussed earlier, if there are common business rules or logic for properties exposed in such read-only objects, the common behaviors should be normalized into another object.

---

Figure 6-8 shows the two collection objects with their corresponding read-only child objects.

The `ProjectInfo` object is responsible for providing read-only information about a project, while the `ResourceInfo` object provides read-only information about a resource. By loading the minimum amount of data required to meet these responsibilities, these objects provide a high-performance solution and follow good behavioral object design.



**Figure 6-8.** The read-only collection objects, *ProjectList* and *ResourceList*

## Inter-Object Collaboration

The object model has a *RoleList* object, responsible for providing a read-only list of role data. It also has a *Roles* object, responsible for editing the list of roles in the application. While these two objects have very distinct responsibilities, they do have a point of interaction that should be addressed.

Though not required by any use case from a user, the *RoleList* object can, and probably should, be cached. The list of roles won't change terribly often, and yet the *RoleList* object will be used frequently to populate UI controls and to validate data from the user. There's no sense hitting the database every time to get the same data over and over.

You'll see how to easily implement the caching in Chapter 8, but first, there's a design issue to consider: what happens when the user edits the list of roles using the *Roles* object? In such a case, the *RoleList* object will be inaccurate.

---

**Note** There's a related issue too, which is when *another user* edits the list of roles. That issue is harder to solve, and requires either periodic cache expiration or some mechanism by which the database can notify the client that the roles have changed. Solving this problem is outside the scope of this discussion, however.

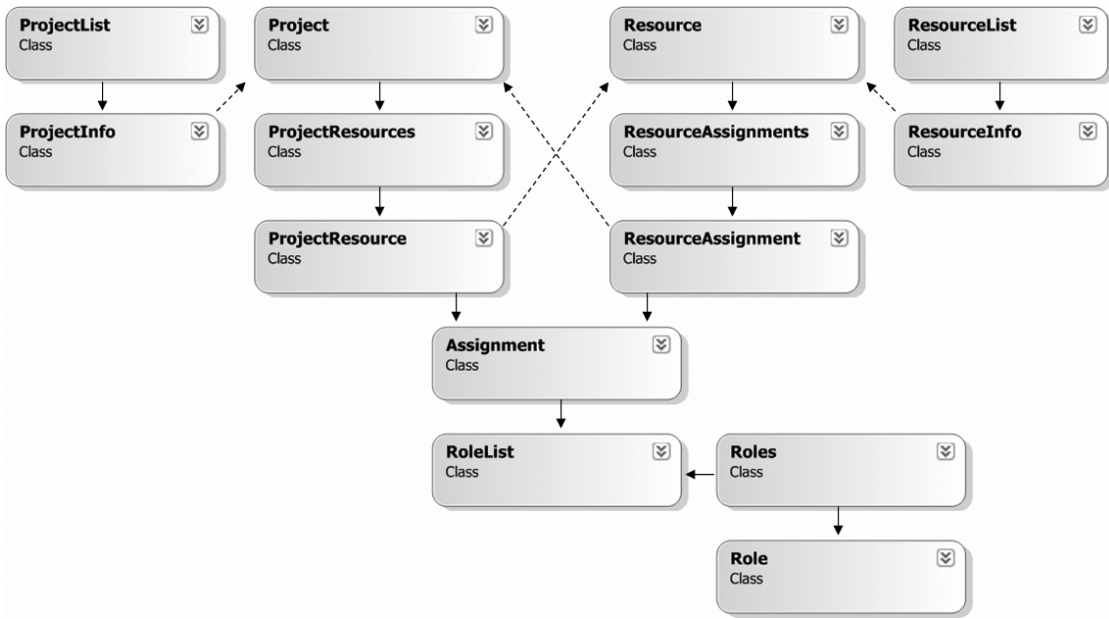
---

It is relatively trivial to have the *Roles* object notify *RoleList* to tell it that the data has changed. In such a case, *RoleList* can simply invalidate its cache so the data is reloaded on the next request. Again, the implementation of this behavior is shown in Chapter 8.

From an object model perspective, however, this means that there is interaction between *Roles* and *RoleList*. From a CRC perspective, this means that *Roles* collaborates with *RoleList* to expire the cache when appropriate.

## Reviewing the Design

The final step in the object design process is to compare the new class diagram with the original use case descriptions in order to ensure that everything described in each use case can be accomplished through the use of these objects. Doing so helps to ensure that the object model covers all the user requirements. The complete object model is shown in Figure 6-9, with the updated CRC information shown in Table 6-3.



**Figure 6-9.** Final project tracker object model

The solid-lined arrows in Figure 6-9 indicate collaboration between objects, illustrating how many of them work together to provide the required functionality. The dashed lines show *navigation* between objects. For instance, if you have a `ProjectInfo` object, it is possible to navigate from there to a `Project`, typically by calling a `GetProject()` method.

While navigation between objects isn't strictly necessary, it is often of great benefit to UI developers. Consider that a UI developer will get access to a `ProjectInfo` object when the user selects a project from a control in the UI. In most cases, the next step is to load the associated `Project` so that the user can view or edit the data. Providing navigational support directly in the object model makes this trivial to implement within the UI.

**Table 6-3.** Final List of Objects and Their Responsibilities

Potential Class	Responsibility	Collaborators
Project	Adds and edits a valid project	ProjectResources, CommonRules
ProjectResources	Maintains a list of resources assigned to a project	ProjectResource
ProjectResource	Manages assignment of a resource to a project	Assignment, CommonRules, Resource
Resource	Adds and edits a valid resource	ResourceAssignments, CommonRules
ResourceAssignments	Maintains a list of projects to which a resource is assigned	ResourceAssignment
ResourceAssignment	Manages a project to which a resource is assigned	Assignment, CommonRules, Project
Assignment	Manages association of a project and a resource	RoleList

Potential Class	Responsibility	Collaborators
ProjectList	Gets a read-only list of projects	ProjectInfo
ProjectInfo	Provides read-only information for a project	Project
ResourceList	Gets a read-only list of resources	ResourceInfo
ResourceInfo	Provides read-only information for a resource	Resource
RoleList	Gets a read-only list of roles	None
Roles	Maintains a list of roles in the system	Role, RoleList
Role	Adds and edits a valid role	None

If you review the use cases, you should find that the objects can be used to accomplish all of the tasks and processes described in the following list:

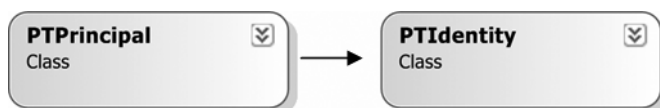
- Users can get a list of projects.
- Users can add a project.
- Users can edit a project.
- Users can remove a project.
- Users can get a list of resources.
- Users can add a resource.
- Users can edit a resource.
- Users can remove a resource.
- Users can assign a resource to a project (and vice versa).
- When a resource is assigned to a project, users can specify the role that the resource will play on the project.

## Custom Authentication

Though the objects required to service the business problem have been designed, there's one area left to address. For this application, I want to show how to use custom authentication. Perhaps this requirement became clear due to a user requirement to support users external to our organization; users that aren't in our corporate domain or Active Directory (AD).

The topic of authentication has been discussed several times in the book thus far, and you should remember that CSLA .NET supports Windows integrated (AD) authentication—in fact, that's the default. But it also supports custom authentication, allowing the business developer to create custom .NET principal and identity objects that authenticate the user using credentials stored in a database, LDAP server, or other location.

To this end, the object model will include two objects: *PTPrincipal* and *PTIdentity*. They are shown in Figure 6-10.



**Figure 6-10.** Business objects subclassing *BusinessListBase*

PTPrincipal is a .NET principal object, and acts as the primary entry point for custom authentication and role-based authorization. PTIdentity is a .NET identity object and is responsible for representing the user's identity.

At this point, the object model can be considered complete.

## Using CSLA .NET

The class diagrams created so far have focused entirely on the business domain—which is a good thing. Ideally, you should always start by focusing on business issues, and deferring much of the technical design to a later stage in the process. Users typically don't understand (or care about) the technical issues behind the scenes, such as how you are going to implement the Cancel buttons, or how to retrieve data from the database.

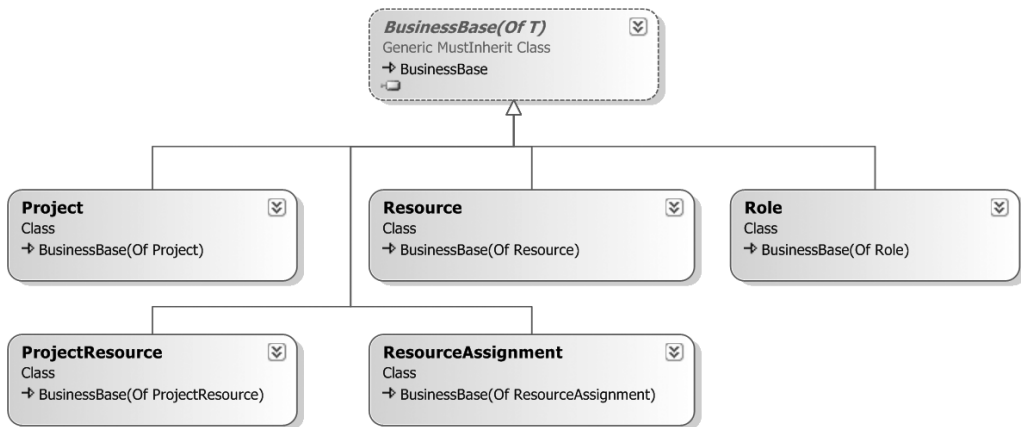
Of course, the business developer cares about these issues—but these issues can be dealt with after the basic object modeling is complete, once you have a good understanding of the business issues and confidence that your model can meet the requirements laid out in the use cases.

At this point in the book, we also have the significant advantage of having designed and built a business framework. This means spending less time figuring out how to design or implement the features included in the framework. By relying on CSLA .NET, developers gain the benefits listed in Table 6-4.

**Table 6-4.** *Benefits Gained by Using CSLA .NET*

Feature	Description
Smart data	Business data is encapsulated in objects along with its associated business logic, so developers are never working with raw, unprotected data, and all business logic is centralized for easy maintenance.
Easy object creation	Developers use standard .NET object-oriented programming techniques to create business objects.
Flexible physical configuration	Data access runs locally or on an application server, without changing business code.
Object persistence	Clearly defined methods contain all data access code.
Optimized data access	Objects only persist themselves if their data has been changed. It's easy to select between various transaction technologies to balance between performance and features.
Optional n-level undo capabilities	Support for complex Windows Forms interfaces is easy, while also supporting high-performance web interfaces.
Business rule management	Reduces the code required to implement business rules.
Authorization rule management	Reduces the code required to implement per-property authorization.
Simple UI creation	With full support for both Windows Forms and Web Forms data binding, minimal code is required to create sophisticated user interfaces (see Chapters 9 and 10).
Web service support	Developers can readily create a web service interface for the application, so that other applications can directly tap into the application's functionality (see Chapter 11).
Custom authentication	Makes it easy to select between Windows integrated security and CSLA .NET custom security. It's also easy to customize CSLA .NET custom security to use preexisting security databases. In either case, standard .NET security objects are used, providing a standard way to access user security information.

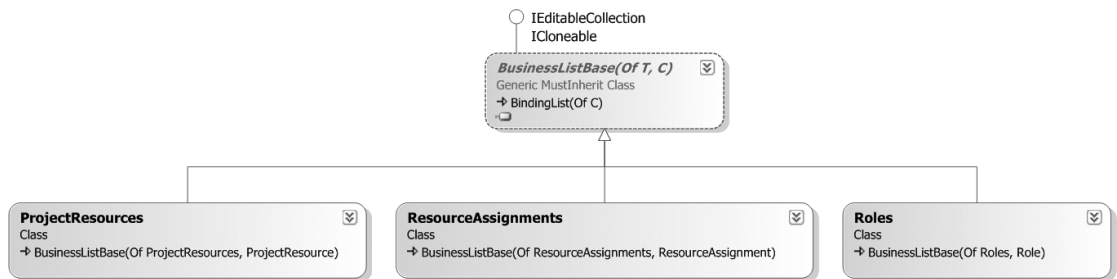
To use CSLA .NET, developers merely need to determine which base classes to inherit from when creating each business class. For example, some business objects will be editable objects that can be loaded directly by the user. These need to inherit from *BusinessBase*, as shown in Figure 6-11.



**Figure 6-11.** Business objects subclassing *BusinessBase*

By subclassing *BusinessBase*, all these objects gain the full set of business object capabilities implemented in Chapters 3 through 5.

The model also includes objects that are *collections* of business objects, and they should inherit from *BusinessListBase*, as shown in Figure 6-12.



**Figure 6-12.** Business objects subclassing *BusinessListBase*

*BusinessListBase* supports the undo capabilities implemented for *BusinessBase*; the two base classes work hand in hand to provide this functionality.

As shown in Figure 6-13, the two objects that list read-only data for the user inherit from *ReadOnlyListBase*.

This base class provides the support objects need for retrieving data from the database *without* the overhead of supporting undo or business rule tracking. Those features aren't required for read-only objects.

The *ProjectInfo* and *ResourceInfo* classes don't inherit from any CSLA .NET base classes. As you'll see in Chapters 7 and 8, they must be marked with the `<Serializable>` attribute, but they don't need to inherit from a special base class just to expose a set of read-only properties.

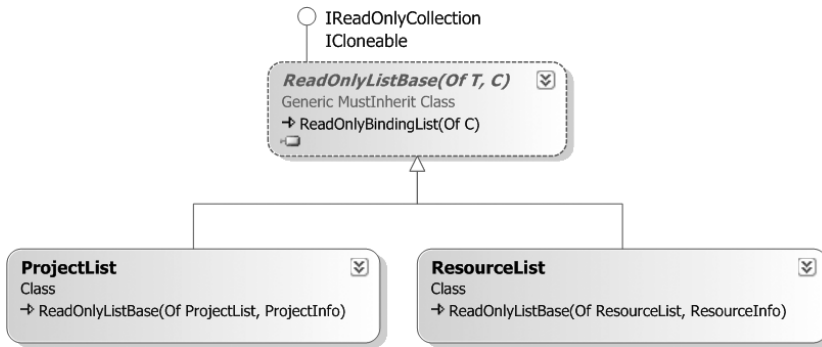


Figure 6-13. Read-only list objects subclassing *ReadOnlyListBase*

Next, there's the *RoleList* object, which is a read-only list of name/value data. Although this *could* be implemented using *ReadOnlyListBase*, Chapter 5 added a better alternative into the framework—the *NameValueListBase* class, as shown in Figure 6-14.

This base class is designed to make it as easy as possible to create read-only lists of text values, so it's ideal for building the *RoleList* class.

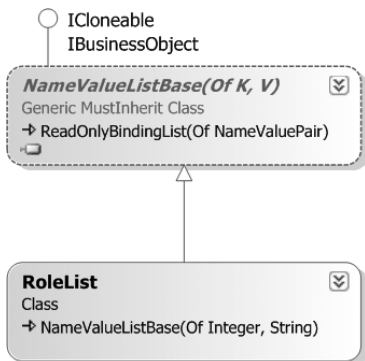


Figure 6-14. *RoleList* subclassing *NameValueListBase*

Finally, there are the two custom authentication objects: *PTPrincipal* and *PTIdentity*. Figure 6-15 shows these objects along with their CSLA .NET base classes.

*PTPrincipal* inherits from *Csla.Security.BusinessPrincipalBase*, ensuring that it implements the *System.Security.Principal.IPrincipal* interface, and also that it will work with the data portal, as implemented in Chapter 4. A required property from the *IPrincipal* interface is *Identity*, which provides a reference to a .NET identity object—in this case, *PTIdentity*.

The *PTIdentity* object inherits from *ReadOnlyBase*. It exposes only read-only data, and so this is a natural fit.

All of these classes will be implemented in Chapter 8. During that process, you'll see how to use the CSLA .NET framework to simplify the process of creating business objects.



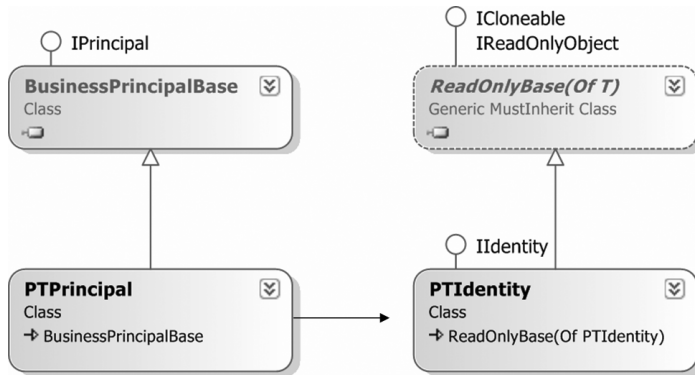


Figure 6-15. Objects supporting custom authentication

## Database Design

It's a rare thing to be able to design a database specifically for an application. More often than not, the databases already exist, and developers must deal with their existing design. At best, you might be able to add some new tables or columns.

This is one reason why ORM is a key concept for object-oriented development. The object model designed earlier in the chapter matches the business requirements without giving any consideration to the database design. An important step in the development process is to create code that translates the data from the databases into the objects, and vice versa. That code will be included in Chapter 8 as the business objects are implemented.

In *this* chapter, let's create a database for use by the project-tracking application. One thing to note is that even though the database is created specifically for this application, the data model will not match the object model exactly. A good relational model and a good object model are almost never the same thing.

---

**Tip** Speaking of good relational models, I strongly recommend that database design be done by a professional DBA, not by software developers. While many software developers are reasonably competent at database design, there are many optimizations and design choices that are better made by a DBA. The database design shown here is that of a software developer, and I'm sure a DBA would see numerous ways to improve or tweak the results to work better in a production setting.

---

To make development and testing relatively easy, this will be a SQL Server 2005 Express database. As you'll see in Chapter 8, you write the data access code for each object, so neither CSLA .NET nor your business objects are required to use SQL Server 2005 Express or any other specific database. You can use any data storage technology you choose behind your objects. In most cases, your applications will use production database servers such as SQL Server 2005 Enterprise Edition, Oracle, or DB2, rather than the more limited Express Edition used here.

The database will include tables, along with some stored procedures to enable database access from code. Additionally, there will be a second database to contain security information for use by the PTIdentity object.

**Tip** If you're using a database other than SQL Server 2005 Express, you should translate the table creation and stored procedures to fit with your environment. You can find the database, table, and stored procedure scripts in the PTData project in the code download from [www.apress.com](http://www.apress.com).

While stored procedures may or may not offer any performance benefits, I believe they are a critical part of any business application. Stored procedures provide an abstract, logical interface to the database. They provide a level of indirection between the business objects and the underlying table structures, and thus they reduce coupling between the data management and business layers in your application. In short, stored procedures help make applications more maintainable over time.

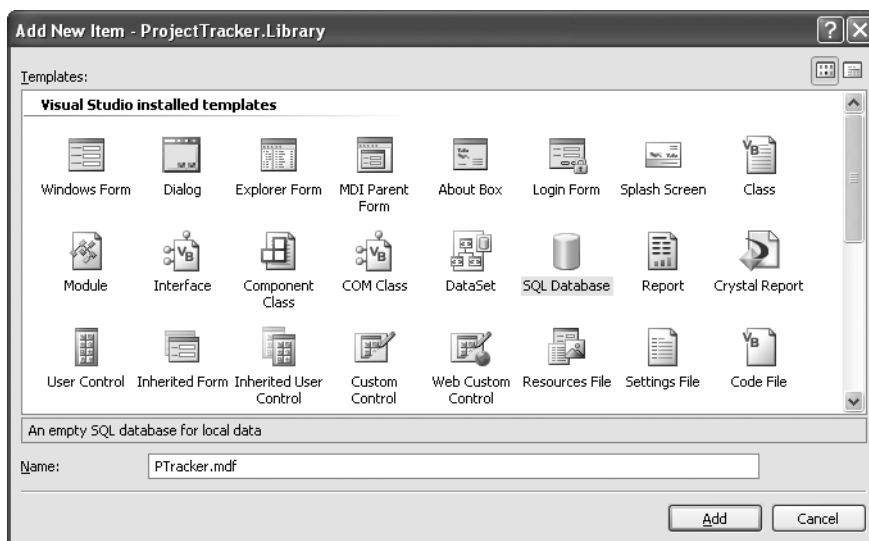
That said, you'll notice that none of these stored procedures are complex, and every effort is made to keep business logic out of the database and in the business objects. Putting the business logic in both the objects and the database is just another way to duplicate business logic, which increases maintenance costs for the application as a whole.

## Creating the Databases

The PTracker database will contain tables and stored procedures to persist the data for the business objects in the object model designed earlier in the chapter. This is a SQL Server 2005 Express database, and so you can think of it as being just another file in your project.

To create the database, open Visual Studio and create a new Class Library project named PTDB. I won't have you build this project at any point, so the project settings and `Class1.vb` file can be ignored. The purpose of this project is just so you can use Visual Studio to set up the database.

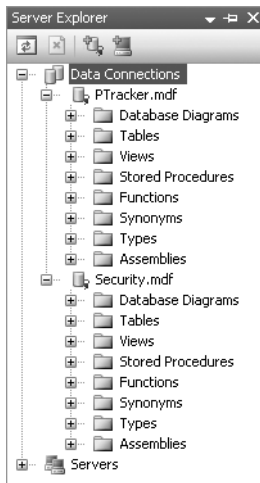
Choose Project ► Add New Item, and choose the SQL Database option. As shown in Figure 6-16, name the file and click Add.



**Figure 6-16.** Adding the PTracker database in Visual Studio

Visual Studio will force you to walk through the process of creating a DataSet for the new database. You can walk through or cancel that wizard as you choose. It is not required for anything covered in this book.

Repeat the process to add a Security.mdf database as well. The end result is that you'll have two databases in the project—and more importantly, in the Server Explorer window, as shown in Figure 6-17.



**Figure 6-17.** *The PTracker and Security databases in Server Explorer*

Table creation can also be done within Server Explorer: just right-click the Tables node under the database, and choose New Table. This will bring up a table designer in VS .NET, with which you can define the columns for the new table.

Once the columns, keys, and indexes have been set up, save the changes by closing the designer or clicking the Save button in the toolbar. At this point, you'll be prompted to provide a name for the table, and it will be added to the database.

## PTracker Database

Follow this process to add each of the following four tables to the database.

### Roles

The Roles table will store the list of possible roles a resource can fill when assigned to a project—it simply contains an Id value and the name of the role. Figure 6-18 shows the VS .NET designer with these columns added, and the Id column configured as the primary key.

dbo.Roles: Ta...PTRACKER.MDF}			
	Column Name	Data Type	Allow Nulls
PK	Id	int	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	LastChanged	timestamp	<input type="checkbox"/>
			<input type="checkbox"/>

**Figure 6-18.** *Design of the Roles table*

Notice that none of the columns allow null values. There's no business requirement to differentiate between an empty value and one that was never entered, so null values would make no sense.

The table also has a `LastChanged` column, which will be used to implement optimistic, first-write-wins concurrency in Chapter 8. It is of type `timestamp`, and so provides a unique, auto-incrementing value every time a row is inserted or updated. All the tables in the PTracker database will have this type of column.

## Projects

The `Projects` table will contain the data for each project in the system. The columns for this table are shown in Figure 6-19.

dbo.Projects... \PTRACKER.MDF			
	Column Name	Data Type	Allow Nulls
PK	Id	uniqueidentifier	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	Started	datetime	<input checked="" type="checkbox"/>
	Ended	datetime	<input checked="" type="checkbox"/>
	Description	varchar(MAX)	<input checked="" type="checkbox"/>
	LastChanged	timestamp	<input type="checkbox"/>
			<input type="checkbox"/>

**Figure 6-19.** Design of the `Projects` table

The `Id` column is set up as the primary key, and it's of type `uniqueidentifier`, which is a `Guid` type in .NET.


There are many ways to create primary key columns in tables, including using auto-incrementing numeric values or user-assigned values. However, the use of a `uniqueidentifier` is particularly powerful when working with object-oriented designs. Other techniques don't assign the identifier until the data is added to the database, or they allow the user to provide the value, which means that you can't tell if it collides with an existing key value until the data is added to the database. With a `uniqueidentifier`, however, the business developer can write code to assign the primary key value to an object as the object is created. There's no need to wait until the object is inserted into the database to get or confirm the value. If the value isn't assigned ahead of time, the database will supply the value.

Notice that the two `datetime` fields allow null values. The null value is used here to indicate an empty value for a date. The `Description` column is also allowed to be null. This isn't because of any business requirement, but rather because it is quite common for database columns to allow null values in cases in which they're meaningless. Chapter 8 will illustrate how to easily ignore any null values in this column.

The `Description` column is of type `varchar(MAX)`, so that it can hold a blob of text data. This field allows the user to enter a lengthy description of the project, if so desired.


## Resources

The `Resources` table will hold the data for the various resources that can be assigned to a project. The columns for this table are shown in Figure 6-20.

dbo.Resources...PTRACKER.MDF)		
Column Name	Data Type	Allow Nulls
 Id	int	<input type="checkbox"/>
LastName	varchar(50)	<input checked="" type="checkbox"/>
FirstName	varchar(50)	<input checked="" type="checkbox"/>
LastChanged	timestamp	<input type="checkbox"/>
		<input type="checkbox"/>

**Figure 6-20.** Design for the Resources table

Once again, the Id column is the primary key—it's an int that is configured as an identity column using the Column Properties window, as shown in Figure 6-21.

Column Properties	
Has Non-SQL Server Subscriber	No
 Identity Specification	Yes
(Is Identity)	Yes
Identity Increment	1
Identity Seed	1
Indexable	Yes

**Figure 6-21.** Making the Id column an identity column



This table has now been given an identity key; the code in Chapter 8 will demonstrate how to support this concept within your business objects.

As with the Description field in the Projects table, the LastName and FirstName columns allow null values even though they have no business meaning. Again, this is merely to illustrate how to build business objects to deal with real-world database designs and their intrinsic flaws.

## Assignments

Finally, there's the Assignments table. A many-to-many relationship exists between projects and resources—a project can have a number of resources assigned to it, and a resource can be assigned to a number of projects.

The way you can represent this relationally is to create a *link table* that contains the primary keys of both tables. In this case, it will also include information about the relationship, including the date of the assignment and the role that the resource plays in the project, as shown in Figure 6-22.

dbo.Assignme...TRACKER.MDF)		
Column Name	Data Type	Allow Nulls
 ProjectId	uniqueidentifier	<input type="checkbox"/>
 ResourceId	int	<input type="checkbox"/>
Assigned	datetime	<input type="checkbox"/>
Role	int	<input type="checkbox"/>
LastChanged	timestamp	<input type="checkbox"/>
		<input type="checkbox"/>

**Figure 6-22.** Design for the Assignments table

The first two columns here are the primary keys from the Projects and Resources tables; when combined, they make up the primary key in the link table. Though the Assigned column is of datetime type, null values are not allowed. This is because this value can't be empty—a valid date is always required. The Role column is also a foreign key, linking back to the Roles table. The data in this table will be used to populate the ProjectResource and ResourceAssignment objects discussed earlier in the chapter.

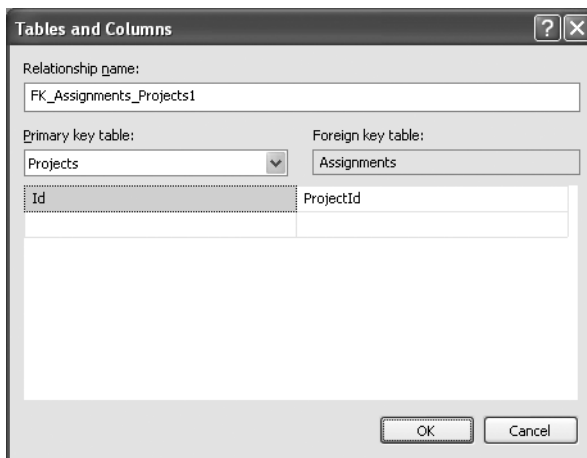
This really drives home the fact that a relational model isn't the same as an object-oriented model. The many-to-many relational design doesn't match up to the object model that represents much of the same data. The objects are designed around normalization of behavior, while the data model is designed around normalization of data.

## Database Diagrams

Server explorer in Visual Studio supports the creation of database diagrams, which are stored in the database. These diagrams not only illustrate the relationships between tables, but also tell SQL Server how to enforce and work with those relationships.

Under the PTracker.mdf node in Server Explorer, there's a node for Database Diagrams. Right-click this entry and choose New Diagram. Visual Studio will prompt you for the tables to be included in the diagram. Highlight all of them, and click Add and Close.

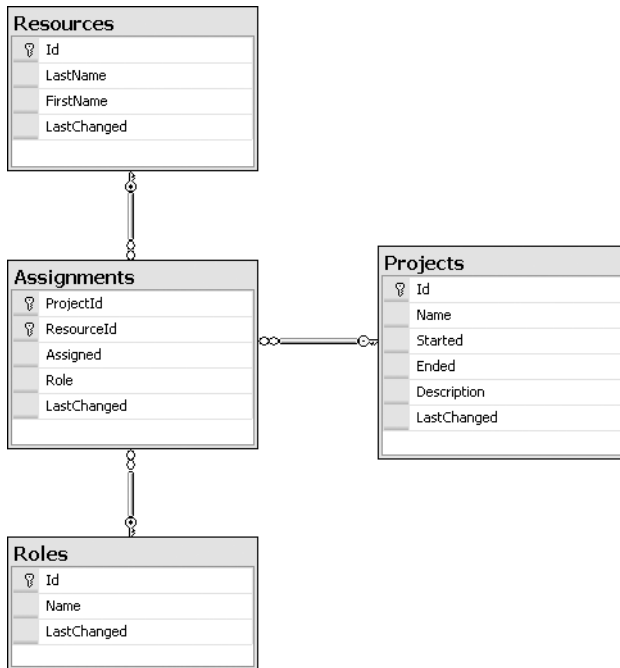
The result is a designer window in which the tables are shown as a diagram. You can drag and drop columns from tables to other tables in order to indicate relationships. For example, drag and drop the Id field from Projects to the ProjectID field in the Assignments table. This will bring up a Tables and Columns dialog box, in which you can specify the nature of this relationship, as shown in Figure 6-23. Click OK to create the relationship.



**Figure 6-23.** *Creating a relationship between Assignments and Projects*

Do the same to link the Resources table to Assignments. You can also link the Roles table's Id column to the Role column in Assignments, thereby allowing the database to ensure that only valid roles can be added to the table.

The resulting diagram should appear in a way that's similar to Figure 6-24.



**Figure 6-24.** Database diagram for the PTracker database

Save the diagram to the database, naming it PTrackerRelationships. VS .NET will then ask whether to update the tables. Remember that these relationships are reflected as formal constraints within the database itself, so this diagram directly impacts the database design.

## Stored Procedures

Whenever possible, database access should be performed through stored procedures. Stored procedures offer powerful security control over the database, and—perhaps most importantly—provide an abstraction layer between the physical structure of the database and the logical way in which it is used. The business objects created in Chapter 8 will make use of stored procedures for their database interaction.

You can use Server Explorer to add the stored procedures to the database by right-clicking the Stored Procedures node under the database, and choosing Add New Stored Procedure. This will bring up a designer window in which you can write the stored procedure code. When you close the designer, the stored procedure will be added to the database.

## getProjects

The getProjects procedure will return the project data to populate the ProjectList object as follows:

```

CREATE PROCEDURE getProjects
AS
    SELECT    Id, Name
    FROM      Projects
    RETURN
  
```

It simply returns basic data about all of the projects in the system. The use cases didn't specify details about the order in which the projects should be listed in a project list, so I haven't included an ORDER BY clause here. Developers may have to do so during the testing process, as users typically add such requirements during that phase.

## existsProject

The existsProject procedure is used to determine if a project's data is in the database:

```
CREATE PROCEDURE dbo.existsProject
(
    @id uniqueidentifier
)
AS
SELECT COUNT(*)
FROM Projects WHERE Id=@id
RETURN
```

The procedure just returns the number of projects with a matching Id value.

## getProject

The getProject procedure retrieves the information for a single project. This is a relatively complex proposition, since a Project object needs to retrieve not only the core project data, but also the list of resources assigned to the project.

This *could* be done by making two stored procedures and calling both of them to populate the business objects, but this can be reduced to a single database call by putting both SELECT statements in a single stored procedure. The stored procedure will then return two result sets, which can be read within the business object's code:

```
CREATE PROCEDURE getProject
(
    @id uniqueidentifier
)
AS
SELECT Id,Name,Started,Ended,
    Description,LastChanged
FROM Projects
WHERE Id=@id

SELECT ResourceId,LastName,
    FirstName,Assigned,Role,
    Assignments.LastChanged AS LastChanged
FROM Resources,Assignments
WHERE ProjectId=@id AND ResourceId=Id
RETURN
```

Notice how the second SELECT statement merges data from both the Assignments table and the Resources table. Remember that the ProjectResource object will expose some resource data as read-only properties, so that data must be returned here.

To some degree, I'm putting ORM logic in the stored procedures by designing them to make it easy for the data access code in each business object to populate the objects. This isn't essential—you could write more complex code in the business objects—but it *is* a good idea, when you can do it.



---

**Tip** In many cases, applications must be built without the option of altering the structure of the database, or even its stored procedures. When that happens, all of the ORM logic must be written within the business objects. The end result is the same; it's merely a matter of where the ORM logic resides.

---

## addProject

The `addProject` procedure is called to add a record to the `Projects` table, as follows:

```
CREATE PROCEDURE addProject
(
    @id uniqueidentifier,
    @name varchar(50),
    @started datetime,
    @ended datetime,
    @description text,
    @description varchar(MAX),
    @newLastChanged timestamp output
)
AS
INSERT INTO Projects
(Id,Name,Started,Ended,Description)
VALUES
(@id,@name,@started,@ended,@description)

SELECT @newLastChanged = LastChanged
FROM Projects WHERE Id=@id
RETURN
```

Note that this only adds the record to the `Projects` table; a separate stored procedure adds records to the `Assignments` table.

This stored procedure not only includes an `INSERT` statement, but also a `SELECT` statement that loads an output parameter value. This is required to support concurrency. Recall that all the tables in the database include a `timestamp` column, which is automatically incremented each time a row is inserted or updated. As you'll see in Chapter 8, the business object must keep track of this value. Since the value changes any time the row changes, the value is returned as the result of any `INSERT` or `UPDATE` operation.

## updateProject

Not only are records added to the `Projects` table, but the application must allow them to be changed. The `updateProject` procedure provides this capability, as shown here:

```
CREATE PROCEDURE updateProject
(
    @id uniqueidentifier,
    @name varchar(50),
    @started datetime,
    @ended datetime,
    @description varchar(MAX),
    @lastChanged timestamp,
    @lastChanged timestamp,
    @newLastChanged timestamp output
)
```

```

AS
UPDATE Projects
SET
    Name=@name,
    Started=@started,
    Ended=@ended,
    Description=@description
WHERE Id=@id
    AND LastChanged=@lastChanged
IF @@ROWCOUNT = 0
    RAISERROR('Row has been edited by another user', 16, 1)

SELECT @newLastChanged = LastChanged
FROM Projects WHERE Id=@id
RETURN

```

Again, this procedure only updates the record in the Projects table; the related records in the Assignments table are updated separately.

Notice the @lastChanged parameter required by the procedure. This represents the last known timestamp value for the row. In Chapter 8, you'll see how this value is maintained by the business object.

When the object attempts to update the row, it provides the last known value for the LastChanged column. If that value hasn't changed in the database, then no other user has updated the row since the object read its data. But if the value *has* changed in the database, then some other user did change the data in the row since the object read the data. First-write-wins optimistic concurrency specifies that this second update can't be allowed, because it could overwrite changes made by that other user.

The UPDATE statement itself uses this parameter in the WHERE clause to ensure that the row is only updated if the value matches. The procedure then checks to see if the row was actually updated. If no rows were updated, it raises an error, which shows up as a database exception in the data access code of the business object.

On the other hand, if the update goes through and the row is changed, then a SELECT statement is executed to return the *new* value of the LastChanged column as an output parameter, so that the object can maintain the new value to allow possible future updates.

## deleteProject

The deleteProject procedure deletes the appropriate record from the Projects table, and also removes any related records from the Assignments table. When creating the relationships between tables in the database diagram, the default is to *not* automatically cascade deletions to child tables:

```

CREATE PROCEDURE deleteProject
(
    @id uniqueidentifier
)
AS
DELETE Assignments
WHERE ProjectId=@id

DELETE Projects
WHERE Id=@id
RETURN

```

If you set up your table relationships to cascade deletes automatically, then obviously the preceding stored procedure would only delete the data in the Projects table.

Though this procedure updates multiple tables, it does *not* include transactional code. Although you *could* manage the transaction at this level, you can gain flexibility by allowing the business object to manage the transaction.

Using the CSLA .NET framework, you have the option to run the data access code within a `System.Transactions` transactional context, to run it within an Enterprise Services distributed transaction, or to manually manage the transaction. When using either `System.Transactions` or Enterprise Services, transactional statements in the stored procedures will cause exceptions to occur. If you opt to handle the transactions manually, you can choose to put the transactional statements here in the stored procedure, or use an ADO.NET `Transaction` object within the business object's data access code.

## addAssignment

When adding or editing a project or a resource, the user may also add or change the associated data in the `Assignments` table. The `addAssignment` procedure adds a new record as follows:

```
CREATE PROCEDURE addAssignment
(
    @projectID uniqueidentifier,
    @resourceID varchar(10),
    @assigned datetime,
    @role int,
    @newLastChanged timestamp output
)
AS
INSERT INTO Assignments
(ProjectId,ResourceId,Assigned,Role)
VALUES
(@projectId,@resourceId,@assigned,@role)

SELECT @newLastChanged = LastChanged
FROM Assignments
WHERE ProjectId=@projectId AND ResourceId=@resourceId
RETURN
```

This procedure may be called during the adding or editing of either a `Project` or a `Resource` object in the application.

Like `addProject`, this procedure ends with a `SELECT` statement that returns the new value of the `LastChanged` column for the row as an output parameter. This value must be maintained by the business object to allow for future updates of the row using the `updateAssignment` stored procedure.

## updateAssignment

Likewise, there's a requirement to *update* records in the `Assignments` table:

```
CREATE PROCEDURE updateAssignment
(
    @projectId uniqueidentifier,
    @resourceId int,
    @assigned datetime,
    @role int,
    @lastChanged timestamp,
    @newLastChanged timestamp output
)
```

```

AS
UPDATE Assignments
SET
    Assigned=@assigned,
    Role=@role
WHERE ProjectId=@projectId AND ResourceId=@resourceId
    AND LastChanged=@lastChanged
IF @@ROWCOUNT = 0
    RAISERROR('Row has been edited by another user', 16, 1)

SELECT @newLastChanged = LastChanged
FROM Assignments
WHERE ProjectId=@projectId AND ResourceId=@resourceId
RETURN

```

As with `addAssignment`, this may be called when updating data from either a `Project` or a `Resource` object.

Notice the `@lastChanged` parameter. It is used in the same way the parameter was used in `updateProject`: to implement first-write-wins optimistic concurrency. If the `UPDATE` statement succeeds, the new value of the `LastChanged` column is returned as a result through an output parameter so that the business object can maintain the new value.

## deleteAssignment

As part of the process of updating a project or resource, it is possible that a specific record will be deleted from the `Assignments` table. An assignment is a child entity beneath a project or resource; and a user can remove a resource from a project, or a project from a resource. In either case, that specific assignment record must be removed from the database:

```

CREATE PROCEDURE deleteAssignment
(
    @projectId uniqueidentifier,
    @resourceId int
)
AS
DELETE Assignments
WHERE ProjectId=@projectId AND ResourceId=@resourceId
RETURN

```

This completes the operations that can be performed on the `Assignments` data. Notice that there's no `getAssignments` procedure. This is because assignments are always children of a project and a resource. The business objects never retrieve just a list of assignments, except as part of retrieving a project or resource. The `getProject` procedure, for instance, also retrieves a list of assignments associated with the project.

## getResources

The `ResourceList` object needs to be able to retrieve a list of basic information about all the records in the `Resources` table, as follows:

```

CREATE PROCEDURE getResources
AS
SELECT Id, LastName, FirstName
FROM Resources
RETURN

```

This information will be used to populate the read-only `ResourceList` business object.

## existsResource

The existsResource procedure is used to determine if a resource's data is in the database:

```
CREATE PROCEDURE dbo.existsResource
(
    @id int
)
AS
SELECT COUNT(*)
FROM Resources WHERE Id=@id
RETURN
```

Like existsProject, the procedure just returns the number of resource rows with a matching Id value.

## getResource

The Resource object needs to be able to get detailed information about a specific record in the Resources table, along with its associated data from the Assignments table. This is very similar to the getProject procedure. Here, too, two result sets are returned from the stored procedure:

```
CREATE PROCEDURE getResource
(
    @id int
)
AS
SELECT Id, LastName, FirstName, LastChanged
FROM Resources
WHERE Id=@id

SELECT ProjectId, Name, Assigned, Role,
    Assignments.LastChanged AS LastChanged
FROM Projects, Assignments
WHERE ResourceId=@id AND ProjectId=Id
RETURN
```

The second SELECT statement returns data not only from the Assignments table, but also from the Projects table. This data will be provided as read-only properties in the ResourceAssignment object. By combining the two SELECT statements into a single stored procedure, the Resource object can make a single database call to retrieve all the data it requires.

## addResource

When a new Resource object is created and saved, its data needs to be inserted into the Resources table:

```
CREATE PROCEDURE addResource
(
    @lastName varchar(50),
    @firstName varchar(50),
    @newId int output,
    @newLastChanged timestamp output
)
```

AS

```
INSERT INTO Resources
  (LastName,FirstName)
VALUES
  (@lastName,@firstName)

SELECT @newId = Id, @newLastChanged = LastChanged
FROM Resources WHERE Id=SCOPE_IDENTITY()
RETURN
```

Remember that the `Id` column in the `Resources` table is an identity column. This means its value is automatically assigned by the database when a new row is inserted. The built-in `SCOPE_IDENTITY()` function is used to retrieve the generated key value, and that value is returned in an output parameter, as a result of the stored procedure. In Chapter 8, you'll see how this value is retrieved by the `Resource` object so that the object becomes aware of the new value. Also, as in `addProject`, the new value for the `LastChanged` column is returned to the object.

The associated `addAssignment` procedure, which can be used to add related records to the `Assignments` table, was created earlier.

## updateResource

Likewise, there's a need to update data in the `Resources` table, as shown here:

```
CREATE PROCEDURE updateResource
(
  @id int,
  @lastName varchar(50),
  @firstName varchar(50),
  @lastChanged timestamp,
  @newLastChanged timestamp output
)
AS
UPDATE Resources
SET
  LastName=@lastName,
  FirstName=@firstName
WHERE Id=@id
  AND LastChanged=@lastChanged
IF @@ROWCOUNT = 0
  RAISERROR('Row has been edited by another user', 16, 1)

SELECT @newLastChanged = LastChanged
FROM Resources WHERE Id=@id
RETURN
```

This procedure will be called when an existing `Resource` object is edited and saved.

## deleteResource

A `Resource` object can be removed from the system. This means removing not only the record from the `Resources` table, but also the associated records from the `Assignments` table, as shown here:

```
CREATE PROCEDURE deleteResource
(
    @id int
)
AS
    DELETE Assignments
    WHERE ResourceId=@id

    DELETE Resources
    WHERE Id=@id
    RETURN
```

This procedure works the same as deleteProject.

## getRoles

The getRoles procedure will return the list of roles to populate the RoleList and Roles objects as follows:

```
CREATE PROCEDURE [dbo].[getRoles]
AS
    SELECT Id,Name,LastChanged
    FROM Roles
    RETURN
```

All the role data is returned as a result of this procedure. Though RoleList and Roles use the data differently, they both use the same set of values.

## addRole

The addRole procedure adds a new entry to the Roles table:

```
CREATE PROCEDURE [dbo].[addRole]
(
    @id int,
    @name varchar(50),
    @newLastChanged timestamp output
)
AS
    INSERT INTO Roles
    (Id,Name)
    VALUES
    (@id,@name)

    SELECT @newLastChanged = LastChanged
    FROM Roles WHERE Id=@id
    RETURN
```

This stored procedure is called by the Role object when it needs to insert its data into the database. As with the other add procedures, this one returns the new value of the LastChanged column for use by the business object.

## updateRole

The updateRole procedure updates an existing entry in the Roles table:

```
CREATE PROCEDURE [dbo].[updateRole]
(
    @id int,
    @name varchar(50),
    @lastChanged timestamp,
    @newLastChanged timestamp output
)
AS
UPDATE Roles
SET
    Name=@name
WHERE Id=@id
    AND LastChanged=@lastChanged
IF @@ROWCOUNT = 0
    RAISERROR('Row has been edited by another user', 16, 1)

SELECT @newLastChanged = LastChanged
FROM Roles WHERE Id=@id
RETURN
```

This stored procedure is called by the Role object when it needs to update the data in the database.

## deleteRole

The deleteRole procedure removes an entry from the Roles table:

```
CREATE PROCEDURE [dbo].[deleteRole]
(
    @id int
)
AS
DELETE Roles
WHERE Id=@id
RETURN
```

This stored procedure is called by the Role object when it needs to remove a row of data from the database.

At this point, stored procedures exist to do every bit of data access. In Chapter 8, the business objects will implement data access code using ADO.NET that makes use of these stored procedures.

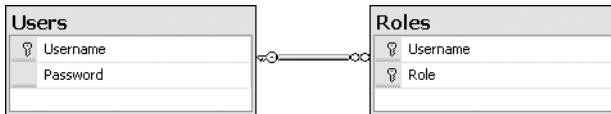
## Security Database

With the PTracker database complete, let's wrap up the chapter by creating the tables and stored procedures for the Security database. This database will be used by the PTIdentity object to perform custom authentication of a user's credentials. Assuming the user is valid, the user's roles will be loaded into the business object so they can be used for authorization as the application is used.

The PTPrincipal and PTIdentity objects will be implemented in Chapter 8. In most cases, you'll be creating similar custom security objects—but designed to use your preexisting security database tables. The database created in this chapter and the objects created in Chapter 8 exist primarily to demonstrate the basic process required for creating your own objects.

Figure 6-25 shows the two tables in the database, along with their relationship.





**Figure 6-25.** Database diagram for the Security database

In the Users table, Username and Password are both `varchar(20)` columns, as is the Role column in the Roles table. Only the Password column allows null values. All other values are required. Of course, a password should be required as well, but for this simple example, it is left as optional.

Finally, there's a Login stored procedure:

```

CREATE PROCEDURE Login
(
    @user varchar(20),
    @pw varchar(20)
)
AS
    SELECT Username
    FROM Users
    WHERE Username=@user AND Password=@pw;

    SELECT R.Role
    FROM Users AS U INNER JOIN Roles AS R ON
        R.UserName = U.UserName
    WHERE U.Username = @user and U.Password = @pw
    RETURN
  
```

This procedure is called by `PTIdentity` to authenticate the user and retrieve the user's list of roles. As you'll see in Chapter 8, `PTIdentity` determines whether the user's credentials are valid or not by finding out whether any data is returned from this stored procedure. If no data is returned, then the user's credentials are assumed to be invalid and the user is not authenticated.

On the other hand, if the stored procedure does return data, then `PTIdentity` stores that data, especially the list of roles to which the user belongs. This list of security roles (not to be confused with the project roles from the `PTracker` database) is then used for authorization throughout the application. The `CanReadProperty()` and `CanWriteProperty()` methods on each business object rely on this data.

## Conclusion

This chapter has started the process of building a sample application that will make use of the `CSLA .NET` framework. It's a simple project-tracking application that maintains a list of projects and a list of resources, and allows the resources to be assigned to the projects.

The application's design used an object-oriented analysis technique that involved creating use cases that described the various ways in which the users need to interact with the system. Based on the use cases, and by using elements of CRC-style design, a list of potential business objects was created and refined.

That object list was then used to create a preliminary class diagram that showed the classes, their key data fields, and their relationships. Based on the diagram, our understanding of the business domain, and the use cases, we were able to refine the design to arrive at a final class diagram that describes the business classes that will comprise the application.

The next step was to determine the appropriate CSLA .NET base classes from which each business object should inherit. The editable business objects inherit from `BusinessBase`, and the collections of editable child objects inherit from `BusinessListBase`. The lists of read-only data inherit from `ReadOnlyListBase`, each of which contain simple child objects that don't inherit from a CSLA .NET base class at all. The list of simple name/value role data inherits from `NameValueListBase`.

Finally, a simple relational database was created to store the data for the application. In most applications, the database already exists, but in this case, we had the luxury of creating a database from scratch. Even so, it's interesting to note the differences between the object model and the relational model, thus highlighting the fact that a good object-oriented model and a good relational model are almost never the same.

Chapter 7 will discuss the basic structure of each type of business object directly supported by CSLA .NET. The chapter will also walk through a code template for each type. Then, Chapter 8 will implement the business objects designed in this chapter, and Chapter 9 will show how to build a Windows Forms UI based on those objects. In Chapter 10, a comparable Web Forms UI will be built, and Chapter 11 will walk through the construction of a Web Services interface that reuses the exact same objects. Finally, Chapter 12 will show how to host the server-side data portal components on various application server technologies.