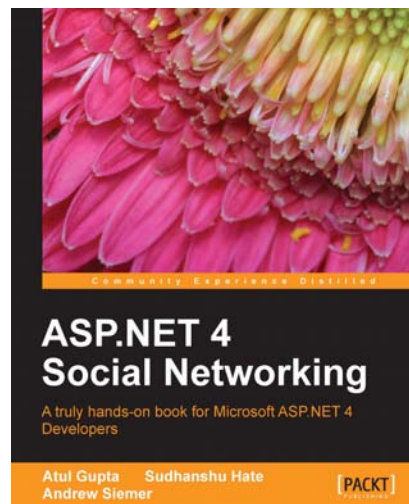




ASP.NET 4 Social Networking

Atul Gupta
Sudhanshu Hate
Andrew Siemer



Chapter No.3 "User Accounts"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.3 "User Accounts"

A synopsis of the book's content

Information on where to buy this book

About the Authors

Atul Gupta is a Principal Technology Architect at Infosys Technologies Limited. He has more than 15 years of experience working on Microsoft Technologies. The technology experience is spread across VC++, MFC, COM/DCOM, SQL, .NET, Microsoft ASP.NET, Microsoft ASP.NET AJAX, Microsoft ASP.NET MVC, C#, ADO.NET, VS (various versions), BizTalk Server, Commerce Server and he has worked across all SDLC life cycle phases like architecture definition, design, development, and testing.

His current focus is User Experience Technologies from Microsoft like Windows Presentation Foundation and Silverlight. He is also exploring Microsoft Surface, Windows Touch, Windows Phone 7, Pivot, and Augmented Reality. He blogs on latest technologies at: <http://blogs.infosys.com/microsoft>. Some of his other publications can be accessed at Infosys' Technology Showcase (<http://www.infosys.com/microsoft/resource-center/Pages/technology-showcase.aspx>). Working on latest technologies is his passion and this had helped him bag the Microsoft's Most Valuable Professional (MVP) award for six consecutive years. He has also spoken on events like Microsoft Virtual Tech Days.

For More Information:

www.PacktPub.com/asp-net-4-social-networking/book

Atul holds a Bachelor's degree in Chemical Engineering and Master's in Software Systems from Premier Universities in India.

He recently helped review the book titled *Refactoring with Microsoft Visual Studio 2010* from Packt Publishing.

For More Information:

www.PacktPub.com/asp-net-4-social-networking/book

ASP.NET 4 Social Networking

Social networking has become a driving force on the Internet. Many people are part of at least one social network, while more often people are members of many different communities. For this reason many business people are trying to capitalize on this movement and are in a rush to put up their own social network. As the growth of social networks continues, we have started to see more and more niche communities popping up all over in favor of the larger, all-encompassing networks in an attempt to capture a sliver of the market.

In this book, we will discuss the many aspects and features of what makes up the majority of today's social networks or online communities. Not only will we discuss the features, their purpose, and how to go about building them, but we will also take a look at the construction of these features from a large scale enterprise perspective. The goal is to discuss the creation of a community in a scalable fashion.

What This Book Covers

Chapter 1, Social Networking gives you an overall structure of this book, that is, what a reader can expect from this book.

Chapter 2, An Enterprise Approach to our Community Framework helps you create an enterprise framework to handle the needs of most web applications. It discusses design patterns, best practices, and certain tools to make things easier. It also covers error handling and logging.

Chapter 3, User Accounts covers registration and account creation process by means of an email verification system and a permission system to ensure security. It also touches upon password encryption/decryption techniques.

Chapter 4, User Profiles covers the creation of a user's profile and an avatar in a manner that is flexible enough for all systems to use. In this chapter, we also implement some form of privacy to allow users to hide parts of their profile that they don't want to share with others.

Chapter 5, Friends shows you how to implement friends, how to search for them, find them in the site's listings, and import your contacts into the site to find your friends.

Chapter 6, Messaging helps you create a messaging system that will resemble a webbased email application similar to Hotmail or Gmail. We will also learn how to implement the Xinha WYSIWYG editor in a way that can be re-used easily across the site for complex inputs.

Chapter 7, Media Galleries covers details on how to build a generic media management system that will allow you to host video, photos, resumes, or any number of physical fi

For More Information:

www.PacktPub.com/asp-net-4-social-networking/book

les with minimal tweaking. It also addresses the issue of multi-file uploads via RIA technologies like Flash and Silverlight.

Chapter 8, Blogs is all about Blogging. With search engines, users, and security in mind, we invest a part of this chapter to address an issue that plagues many dynamic websites—query string data being used to determine page output.

Chapter 9, Forums discusses the creation of the core features of a discussion forum—categories, forums, threads, and posts. Along with these features, the chapter also extends the friendly URLs concept to make our content more suitable for search engine optimization.

Chapter 10, Groups covers the concept of Groups. It focuses on how groups can be used to bring many different systems together in a way to start creation of sub communities.

Chapter 11, User Interactivity helps us build controls to allow our users to express their opinions about various content areas of our site—tagging, rating, commenting, voting and mark as answer. It also discusses how these in turn allow users to earn medals and hence reputation on the site.

Chapter 12, Moderation focuses on Moderation, that is, the means to manage community provided content using a very simple flagging tool. It also covers methods such as Ggging to deal with habitual rule breakers. It also takes a look at how to filter specific words from content on the site.

Chapter 13, Scaling discusses some concepts to help you support a large number of users on your social network. It starts by looking at some key concepts of tiered architecture and web farming. It also discusses ways to create and search indexed data, methods to optimize data retrieval and content creation, and some mail queuing concepts.

Appendix covers a discussion on the Microsoft ASP.NET MVP and MVC patterns and explains why we continued to use the MVP pattern for this book.

For More Information:

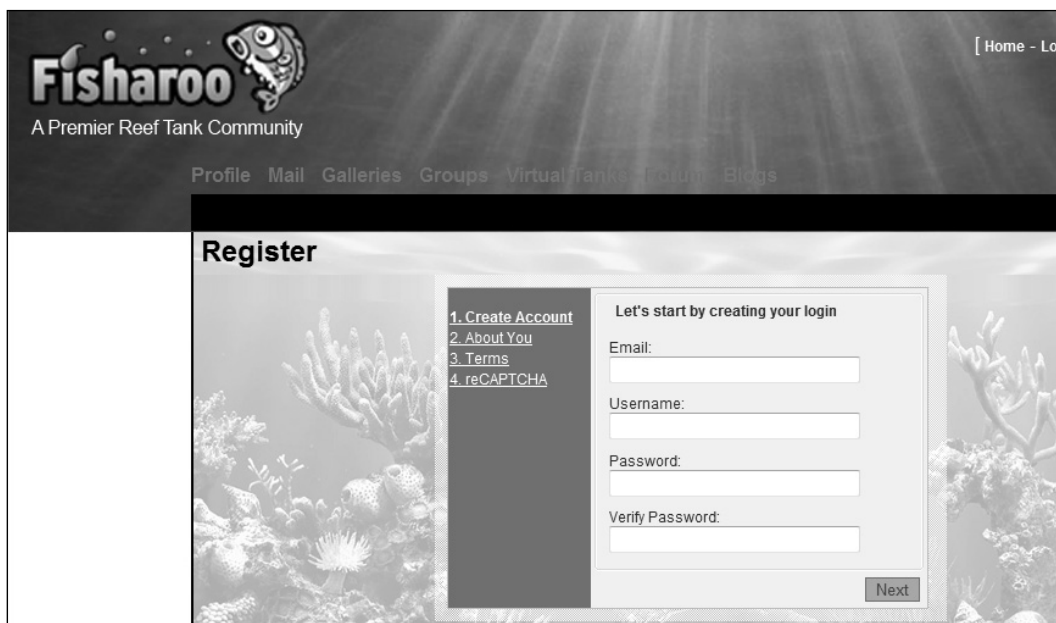
www.PacktPub.com/asp-net-4-social-networking/book

3

User Accounts

Without people, your community doesn't exist!

For any community site to be considered successful, it must first have a group of dedicated users. While there are other measures, however in most cases the larger the community's population the more successful it is considered to be. It would then make sense that we create a way for users to come to our site, create an account, and become a part of our community. The following screenshot shows how the registration page will look like:



Fisharoo
A Premier Reef Tank Community

[Home - Log

Profile Mail Galleries Groups Virtual Tanks Forum Blogs

Register

1. [Create Account](#)
2. [About You](#)
3. [Terms](#)
4. [reCAPTCHA](#)

Let's start by creating your login

Email:

Username:

Password:

Verify Password:

Next

For More Information:
www.PacktPub.com/asp-net-4-social-networking/book

In this chapter, we will discuss many of the common features that are related to user accounts. This will include handling registration, authentication, permissions, and security. We will also go over some basic tools such as password reminders, account administration, and reCAPTCHA. This chapter will provide the foundation for our users upon which we will be able to build all of our other features.

Problem

With most sites these days, regardless of their purpose, you need to know who your users are. You need to know this so that you can restrict how users interact with the site, or you might need this information so that you can provide a dynamic experience to your user. No matter what your reason is, the task of identifying and controlling the users has a few basic requirements.

In order to get to know our users, we will need a way to register them on our site. This would give us a footprint for that user, that we can use each time the user returns. The registration process is fairly straightforward most of the time so that we can easily bring onboard new users. We need to capture the data that we are interested in (such as username, password, email, and so on). We must make sure that we store their password properly so that their identification is safe not only from the other users of the site but also from the administrators and staff of the site. Also, given the amount of fraud and spam on the Internet these days, we need to equip our site with some form of intelligence to guard it from automated registrations. In another attempt to protect the site, we need to make sure that our users are providing us with valid information. We can do this by validating the email provided by them to check if it is a functioning account under their control. As part of the registration process, we also need to inform the user about our current terms and conditions so that they know the rules of our site up-front.



When we refer to automated registrations, we are really describing the act of a bot (or program) that is used to create accounts with the sole purpose of posting advertisements to public areas such as message boards, blogs, and so on.

Once a user has successfully registered, we will need to provide them with tools so that they can identify themselves to us each time they return. This is typically done using a centralized login screen. Upon successful authentication, we can track that user through the site. Knowing that users frequently forget the information that they provided us with, we will need to offer tools to remind the users how to get into our site with a password reminder feature. After the users have authenticated themselves, we would need to define where a user can go and what they can do on our site with some kind of permissions-based system.

Once the users are registered and authenticated, we will need to provide them with a way to administer their account data. In addition to the users being able to administer their own data, the staff that runs the site will also need tools to manage all the users and their data. In addition to managing user data, administrators should be able to control the users' permissions and update the terms and conditions.

Design

In this section, we will discuss the various aspects that are required to implement our new features. Once we are finished, we should have a good idea of what will be required from each area.

Registration

Registration includes the task of acquiring user information, allowing them to pick a username, password, and email verification. In addition, we will require that our users agree to our terms and provide verification that they are human and not a bot by reading our reCAPTCHA image. Once we have all this information, we will create the user account and assign appropriate permissions to the account.

Accounts

While ASP.NET provides various pre-built tools for handling your users via the membership controls, we have decided to explore a custom way to handle our users with regards to logging them in, encrypting their passwords, and so on. The reason for this is to demonstrate that it is fairly simple to build custom authentication logic, and there is sufficient literature available on regular ASP.NET authentication anyway.

To begin with, we need a way to describe our accounts. From the database point of view, it will be fairly simple. All we need is an "Accounts" table where we can hold a username, password, and a few other bits of information.

Password strength

Password strength is not only an issue for the account's security but also for the site owner. The weaker your user's password is, the more likely someone performs a brute force attack on your site. If an account with a high-level permission (such as Administrator) is compromised due to a weak password, you will look pretty silly! It doesn't make much sense on your part to create a secure site in every other way and then allow your users to bypass all your efforts!

Having said that forcing your users to have a strong password, can become an inconvenience to them. Believe it or not, there are many users who would prefer to have a password as "password". While we are not for letting your users become lazy and placing their account at risk, you do need to be aware that there is a chance that you will lose signups due to this requirement. It is up to you to decide how important a secure site is.

Terms and conditions

While terms and conditions are not a necessary requirement for a good site, this section is the place to cover the concept. We will create a simple way to manage your terms and conditions. Terms and conditions are a legal thing. Knowing that terms and conditions can change over time, it is important that you track which version of terms and conditions your users last agreed to. It is also important that you track when they agreed to them.

reCAPTCHA

CAPTCHA, or **C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part, is a form of challenge-response test to determine if the user of your site is a computer or human. reCAPTCHA is a CAPTCHA service from Google, that helps in digitizing books, newspapers, and old time radio shows.

reCAPTCHA presents two words to the user and of these two, one is a word that the **Optical Character Recognition (OCR)** process is able to convert from image to text. The other is the word that the OCR cannot translate. The system assumes that if the user provided the right answer for the word for which the translation was known, the other one for which it wasn't known is also translated correctly. The system will then reuse this new word in multiple instances to gain higher confidence in correctness of translation. reCAPTCHA provides plugins for various programming environments and we will be using the one for ASP.NET. For more details check out this website: <http://www.google.com/recaptcha>.

Here is an example of an image that will be generated by our system:



Email confirmation and verification

There are several reasons to use email communications in our community. For example, our site will require registration confirmation and email validation.

When a user signs up at our site, we need to be able to let them know that the registration was completed successfully. This email will usually welcome a user to the site. It may also provide them with some frequently asked questions, a list of benefits received upon registration, and any other pertinent information that a user may need prior to using your site.

In addition to the registration receipt, we need a way to check whether the email address that the user has provided is a valid account (that it actually exists), and one to which the user actually has access. We will validate this by embedding a link in the email that the user will have to click. Once this link is clicked, we will assume that the email address is valid!

Security

Security is obviously one of the most important aspects of building a site. Not only should you be able to provide access to certain areas of your site a specific user, but more important is the ability to deny access to various users of your site. All sites have areas that need to be locked down. For example, one of the most important areas could be the administration section of your site, or paid areas of your site. Therefore, it is very important to make sure that you have some form of security.

Permissions

There are a number of ways to handle permissions (also called as authorization). We could make something really complex by implementing a permission-based system using permissions, roles, groups, and so on. We could even make it as complex as Microsoft's Active Directory system. However, I find that keeping something only as complex as your current requirement is the best thing to do. We can always add to the system as our needs increase.

Our permissions system will simply encompass a name/role (Administrator, Editor, Restricted, and so on). This permission name will then be statically mapped to each page (using the `Sitemap` file that .NET provides us!). As long as we keep our pages to serve a single functionality, this should always suit our needs. As this is a good design practice anyway, we shouldn't have any problems here. Of course, a user can have many permissions tied to their accounts so that they can traverse various sections of our site. Once this is in place, all we need to do is have a system that checks a user's permissions upon entering each page to ensure that they have the permission that the page requires.

We will have a `Permissions` table, that relates to the `Accounts` table through the `AccountPermissions` table. This is a pretty straightforward and simple design.

Password encryption/decryption

When working with passwords, there is an important question to answer. Do we have one way hashing, or do we store encrypted passwords and provide a way to decrypt them? If we don't provide decryption facilities, then we won't be able to send reminder emails to our users who have their passwords. If we were creating a banking system, sending passwords via email would not be acceptable. However, as we are creating something slightly less confidential, the convenience for users to be able to retrieve their passwords without too much hassle is a great reason for decrypting the password and sending it to the user.

Logging in

Once a user has created an account, it is important for them to be able to re-identify themselves to us. For this reason, we will need to provide a way for them to do this. This will come in the form of a page that accepts a username and a password. Once they have authenticated themselves to us, we will need to make sure that they are still allowed to get into our site (if they are valid users).

Password reminder

A user will inevitably forget his/her password. As we require a strong password, and a fair number of users would rather not have a password at all, or would like to use the word "password" as their password, it is highly possible that they forget what they registered with. Not a problem! As we decided to use a two-way encryption, we will be able to decrypt their chosen password and email it to them. This way, our user will never be locked out for too long!

Manage account

In order to keep customer service calls to a minimum, we will need a way for our customers to manage their own accounts. Our customers will need a way to update most of the information they provide us. While we will not allow them to change their username, we will allow them to edit the rest. And when we allow them to change their email address we need to make sure that we force them to validate their new address.

Solution

Now, let's take a look at how we can go about implementing all the new features.

Implementing the database

We will start by implementing our database, and we will work our way up from there.

The Accounts table

The `Accounts` table will store all the base information for a user. Most of this is easy to figure out as they have indicative names (a sign of good design).

	Column Name	Data Type	Allow Nulls
?	AccountID	int	<input type="checkbox"/>
	FirstName	varchar(30)	<input checked="" type="checkbox"/>
	LastName	varchar(30)	<input checked="" type="checkbox"/>
	Email	varchar(150)	<input checked="" type="checkbox"/>
	EmailVerified	bit	<input type="checkbox"/>
	Zip	varchar(15)	<input checked="" type="checkbox"/>
	Username	varchar(30)	<input checked="" type="checkbox"/>
	Password	varchar(50)	<input checked="" type="checkbox"/>
	BirthDate	smalldatetime	<input checked="" type="checkbox"/>
	CreateDate	smalldatetime	<input checked="" type="checkbox"/>
	LastUpdateDate	smalldatetime	<input checked="" type="checkbox"/>
	Timestamp	timestamp	<input type="checkbox"/>
	TermID	int	<input type="checkbox"/>
	AgreedToTermsDate	smalldatetime	<input checked="" type="checkbox"/>

However, there are a few columns that may not be 100 percent clear at first glance. We will explain those here.

EmailVerified	This is a bit flag to let us know if a user's email address has been verified or not.
CreateDate	This is the date on which the record was created. It has a default value of <code>GetDate()</code> .
LastUpdateDate	This is similar to the CreateDate with the exception that we should update it every time we update the record. This could be done with a trigger, or done programmatically.
AgreedToTermsDate	This is used to track the date on which the user agreed to the terms and conditions.

The Permissions table

The `Permissions` table primarily acts as a lookup table for the various types of permissions. It holds the name of each permission with a unique ID.

The AccountPermissions table

The AccountPermissions table allows us to create a many-to-many type of relationship between our Permissions and our Accounts. It simply holds a reference to a record the end of the relationship.

The Terms table

The Terms table is a lookup for our terms and conditions. Also, it provides us with a historical view of the terms our customers have agreed to in the past.

Creating the relationships

First, while you could work in a database without any enforced relationships, we wouldn't advise it. Secondly, if you don't have all your database constraints clearly defined, you might find yourself working with corrupt data.

For this set of tables we have relationships between the following tables:

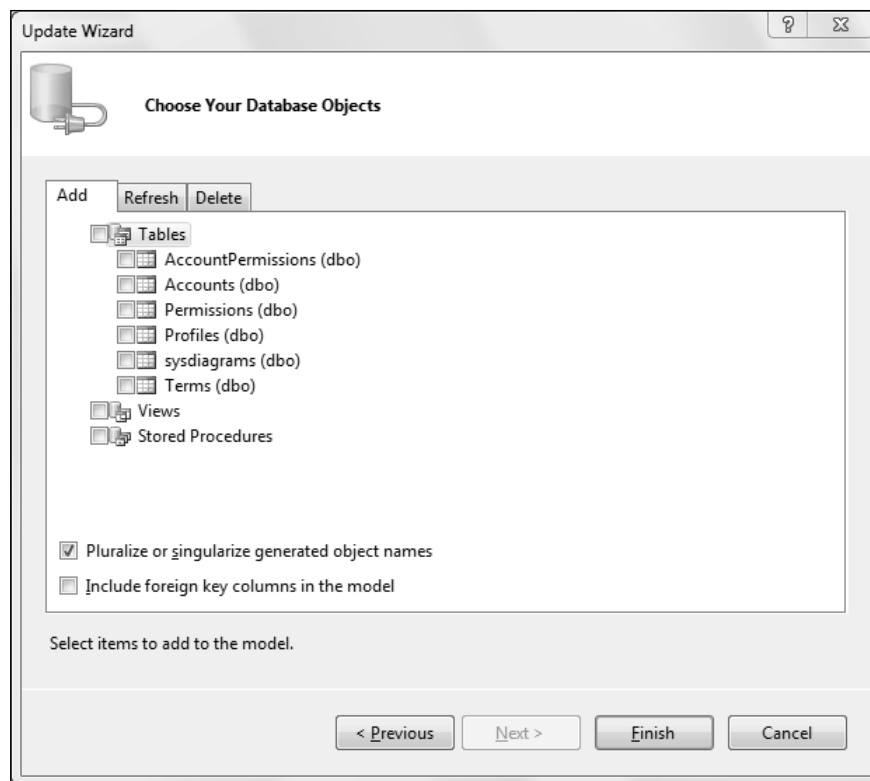
- Accounts and AccountPermissions
- Permissions and AccountPermissions
- Accounts and Terms

Implementing the data access layer

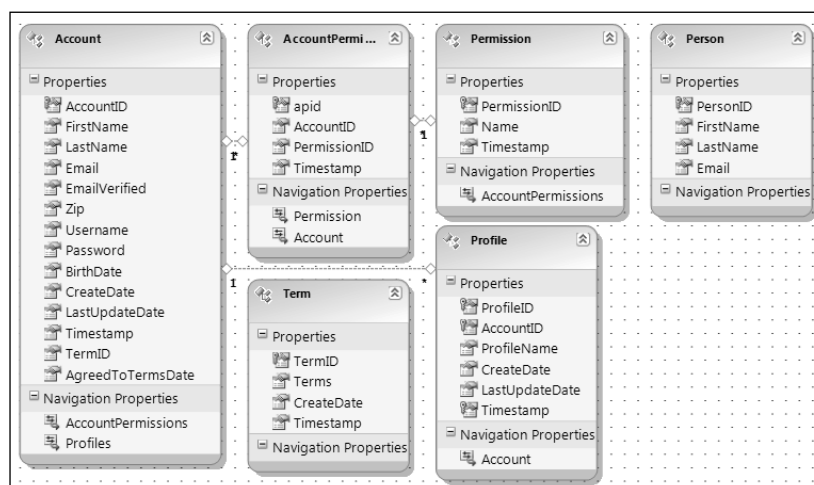
Now that we have our database defined for all the features required by this chapter, let's take a look at how we go about accessing that data! Keep in mind that this chapter will not have a step that we discussed in an earlier chapter, i.e. telling Entity Framework how to connect to our database so that it can generate Entity classes for us based on our table structure.

Update Model from Database

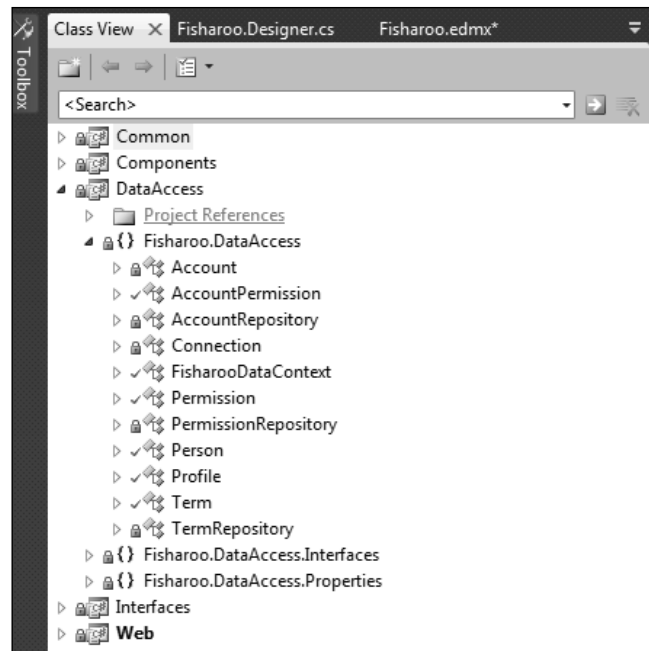
In Chapter 2, we created EDMX and added Person table from Database. In this chapter, so far we have discussed various tables and relationships that we created in SQL Server. Now it's time to get them reflected in our EDMX. This can be done by double clicking Fisharoo.EDMX (you need to first open the solution in Visual Studio 2010) and it will open it in designer interface. Right click the designer interface and select **Update Model from Database**. In the wizard that comes up (as shown in the following screenshot) select the necessary tables that need to be added.



Click **Finish** and the selected tables are added to the EDMX designer as shown in the following screenshot:



At this time VS 2010 will update `Fisharoo.Designer.cs` and generate Entity classes corresponding to all the tables selected. You won't see these classes as files in your project, but we can see them in the class viewer. Open your class viewer by going into the **View** menu and selecting **Class View**. Then expand the `DataAccess` project.



To start with, you should see a class for each table you put on the `Fisharoo.EDMX` design surface. These are partial classes that you can extend by making an additional partial class of the same name in the same namespace (we will do this in a while). Do not edit the generated classes directly as your additions will get lost the next time you generate them!

The only other item here that you should see other than the classes that represent your tables is `FisharooDataContext`. This class handles all the LINQ facilities for your tables and classes. It tracks what changes you have made to your objects, what objects can be worked with, how you can query those objects, and so on. Any time we work with our LINQ classes or data, we will be going through the `FisharooDataContext` class.

A Data Context wrapper

Now that we know `FisharooDataContext` is used by LINQ extensively, let's look at how we can work with this `DataContext` wrapper in a way that fits our overall design by limiting the knowledge required to use the `FisharooDataContext`. In Chapter 2 we already discussed a `Connection` wrapper that will return the `FisharooDataContext` to the caller without requiring the caller to know what goes into its actual creation.

Building repositories

Once we have a way to get to our `DataContext`, we can begin to look at how we work with the objects and data stored behind that `DataContext`. While we could just access our objects and the power of LINQ directly in our code, it would be very helpful down the road if we continued our layered approach by adding a **Repository** layer. A **Repository** provides us with a single place to go for our data (that doesn't necessarily have to be a database).

The repository layer is responsible for performing data access and data persistence. Each repository will be responsible for data related to a particular entity.

So let's start creating our first repository by looking at the `AccountRepository`. Navigate to `DataAccess` project and create a new class called `AccountRepository.cs`. Here is what the code looks like:

```
//Fisharoo/DataAccess/AccountRepository.cs
namespace Fisharoo.DataAccess
{
    [Export(typeof(IAccountRepository))]
    public class AccountRepository : IAccountRepository
    {
        private Connection conn;

        public AccountRepository()
        {
            conn = new Connection();
        }

        public Account GetAccountByID(int AccountID)
        {
            Account account = null;
            using (FisharooDataContext dc = conn.GetContext())
            {
                account = (from a in dc.Accounts
                           where a.AccountID == AccountID
```



```
                select a).FirstOrDefault();
            }
            return account;
        }

        public Account GetAccountByEmail(string Email)
        {
            Account account = null;
            using (FisharooDataContext dc = conn.GetContext())
            {
                account = (from a in dc.Accounts
                           where a.Email == Email
                           select a).FirstOrDefault();
            }
            return account;
        }

        public Account GetAccountByUsername(string Username)
        {
            Account account = null;
            using (FisharooDataContext dc = conn.GetContext())
            {
                account = (from a in dc.Accounts
                           where a.Username == Username
                           select a).FirstOrDefault();
            }
            return account;
        }

        public void AddPermission(Account account,
            Permission permission)
        {
            using (FisharooDataContext dc = conn.GetContext())
            {
                AccountPermission ap = new AccountPermission();
                ap.AccountID = account.AccountID;
                ap.PermissionID = permission.PermissionID;
                dc.AccountPermissions.AddObject(ap);
                dc.SaveChanges();
            }
        }

        public void SaveAccount(Account account)
```

```
{
    using(FisharooDataContext dc = conn.GetContext())
    {
        if(account.AccountID > 0)
        {
            dc.Accounts.Attach(new Account { AccountID =
                account.AccountID });
            dc.Accounts.ApplyCurrentValues(account);
        }
        else
        {
            dc.Accounts.AddObject(account);
        }
        dc.SaveChanges();
    }
}

public void DeleteAccount(Account account)
{
    using (FisharooDataContext dc = conn.GetContext())
    {
        dc.Accounts.DeleteObject(account);
        dc.SaveChanges();
    }
}

public List<Account> GetAllAccounts(Int32 PageNumber)
{
    IEnumerable<Account> accounts = null;

    using (FisharooDataContext dc = conn.GetContext())
    {
        accounts = (from a in dc.Accounts
                     orderby a.Username
                     select a).Skip((PageNumber - 1) * 10).
                        Take(10);
    }
    return accounts.ToList();
}
}
```

The first thing you will notice here is the `Export()` attribute of MEF as well as the `IAccountRepository` interface. As we may want to swap this repository out during testing, these are important. Also, don't forget that the use of MEF allows us to easily ensure that coupling is reduced once we start using the Repository. This means that we could technically swap out the entire repository without requiring a change to our code.

Then in the constructor we initialize the `Connection` object for use throughout the rest of the `AccountRepository` class.

Selecting accounts

Once our `Connection` object is ready for use, we can look at any of the methods from a generic point of view. Let's start with the `GetAccountById()` method.

This method is set up to retrieve an account with an ID. We first start out by defining our return variable (account in this case) outside the `using` statement. We then retrieve a `FisharooDataContext` inside a `using` statement, that ensures that the `DataContext` is disposed of once we have finished with it. We then move to the LINQ query itself inside the `using` statement. This looks very much like a standard SQL `SELECT` statement with a twist. We have to define the `from` statement first so that intellisense can interrogate the collection we are working with. This allows us to work with our query as though we were working with any other collection of objects using dot syntax. We then define a `where` clause to restrict what is returned. Finally, we select the object that we want to use.

You will then notice that the entire query is wrapped in parenthesis. This allows us to chain methods on top of the result set. In this case, we are calling the `FirstOrDefault()` function. This restricts our dataset to the first record returned by the query or default value if no element is found. As we know that there can only be one account associated with an ID, this should be acceptable here!



We could use `var` as the query result if we didn't know what type to expect back from our query.

```
var account = [your LINQ query here];
```

`var` is called an anonymous type. It is used heavily by LINQ. The caveat to using an anonymous type is that it can only be used locally. So if your intent is to use the queried objects outside of the scope from which they were retrieved, you will have to do some form of casting, looping, or otherwise, to move them away from their anonymous status.

The `GetAccountByEmail()` and `GetAccountByUsername()` methods are almost identical in the way they function. So we are not going to explain them in detail.

Saving an account

Now that we have a way to select `Account` objects out of the database in various ways, we now need to consider how we are going to get the data into the database. This brings us to our `SaveAccount()` method. We could have created two methods out of this one method – an `Insert()` and an `Update()` method. However, the only difference is in the one line of code between those two methods. So we chose to roll these two methods up and replace them with the `SaveAccount()` method.

As you will see, with all our Repository methods, we have wrapped the acquisition of the `FisharooDataContext` in a `using` statement. This makes our clean up automatic! (Technically speaking, that is!). Once we have our `DataContext` to work with, we interrogate the object that was passed in to see if it already has an `AccountID`. If the object does have an `AccountID`, it can't be new. If it doesn't have an `AccountID`, it must be new.

A new object is easy to work with. We simply call the `dc.Accounts.AddObject()` method and then we call `dc.SaveChanges()`. This tells the `DataContext` that it needs to persist all the changed data into the database – in our case it needs to save that new `Account` record.

Our code makes updating data look almost as easy as inserting new data. However, while updating, we first create stand-in `Account` object by assigning the `AccountId` of the received `Account` and then `Apply Values`. This is achieved by telling the `DataContext` that the object that is being passed in is the modified version of the current original. In the end we still need to call `dc.SaveChanges()`.

Deleting an account

Now that we have a way to add accounts into the system it only makes sense that we would also want to know a way to delete an account, which is achieved with the `DeleteAccount()` method.

We think deleting an object from the `DataContext` is one of the easiest things to do! Simply locate the collection that you want the object to delete from, pass the object to be deleted to the `DeleteObject()` method, and then call `dc.SaveChanges()`. It doesn't get any easier than that!

Adding permissions to an account

Shouldn't adding permission go in a `Permission` repository or something? No, not really. In reality, we are not really adding permission. We are creating a record in the non-entity table, `AccountPermissions`, to link a `Permission` to an `Account`. Recall that we had stated that we will not create non-entity repositories so that we can at least try and stick to DDD. So this leaves us to add permissions to accounts in the `Permissions` repository or in the `Accounts` repository. Adding permissions to the `Accounts` repository makes more sense to us!

The code is also pretty simple as you can see in the `AddPermission()` method (you will find that this is a recurring statement!).

```
public void AddPermission(Account account, Permission permission)
{
    using(FisharooDataContext dc = conn.GetContext())
    {
        AccountPermission ap = new AccountPermission();
        ap.AccountID = account.AccountID;
        ap.PermissionID = permission.PermissionID;
        dc.AccountPermissions.AddObject(ap);
        dc.SaveChanges();
    }
}
```

This method is going to simply link an `Account` object to a `Permission` object. To do this, it expects an `Account` and `Permission` object to be passed in. It then creates a new `AccountPermission` object and assigns the `AccountID` and `PermissionID` properties based on the objects that were passed in. This new `AccountPermission` object is then inserted into the `AccountPermissions` collection in the `DataContext`. Finally, the `SaveChanges()` method is called.

If you think back to our DDD discussions (covered in the Appendices), Entity objects are important enough to recreate and track with a unique ID. Value objects are less important and can't (or shouldn't) exist without a parent Entity object. In this case, the value object, `AccountPermission`, can exist with `Permission` or an `Account` as its parent. While this is a true statement, the overall design can be simplified by stating that `Accounts` can have `AccountPermissions` and that `Permissions` can't. This makes keeping track of the objects easier when they only have one entry point into the world.

Now, having said that, we can think of a scenario where we might need to be able to say: "For this permission, show me all the related accounts". This might be useful in an Administration console. We will see that when we get there. We could just as easily run a query that says: "Show me all the accounts with this permission".

The other repositories

Now that we have had a fairly detailed look at the `AccountRepository`, we are going to quickly cover the remaining repositories. We will discuss some interesting points here and there, but for the most part, once you have seen one repository, you have seen them all!

Permissions repository

In the `GetPermissionsByAccountID()` method of this repository, you will see an interesting LINQ query.

```
//Fisharoo/DataAccess/PermissionRepository.cs
public List<Permission> GetPermissionsByAccountID(Int32 AccountID)
{
    using (FisharooDataContext dc = conn.GetContext())
    {
        var permissions = from p in dc.Permissions
                           join ap in dc.AccountPermissions on
                               p.PermissionID equals ap.PermissionID
                           join a in dc.Accounts on
                               ap.AccountID equals a.AccountID
                           where a.AccountID == AccountID
                           select p;

        return permissions.ToList();
    }
}
```

This query introduces the concept of joining one set of objects with another set of objects exactly as one would do in SQL. In this case, we need to create a variable to reference each collection of objects.



For all you SQL people out there, think of this as a table alias.

Examples of this would be `p in dc.Permissions`, `ap in dc.AccountPermissions`, and `a in dc.Accounts`. Once you have your collections to work with, you can then define the join parameters with `on p.PermissionID equals ap.PermissionID`. This query basically says, "Give me all the Permissions related to these Account-Permissions, related to these Accounts, where the AccountID equals the passed in AccountID."

Another interesting method to look at is `GetPermissionByName()`, interesting because it uses Lambda expressions. We are sure from the statement itself you would have inferred the benefit of the Lambda expression.

```
//Fisharoo/DataAccess/PermissionRepository.cs public
List<Permission> GetPermissionByName(string Name)
{
    using (FisharooDataContext dc = conn.GetContext())
    {
```

```
        var permissions = from p in dc.Permissions.Where(
                            p=>p.Name.Equals(Name))
                            select p;
        return permissions.ToList();
    }
}
```

Key advantages with Lambda Expressions over standard query are type inference, simplicity, and increased readability of code.



Lambda Expression consists of a lambda operator '=>' (goes to), left side of which signifies the input parameter (for more than one input parameter, make it comma separated) and right side signifies the expression or statement block to be evaluated. The compiler performs automatic type inference based on the usage of arguments.

Terms repository

In the `GetCurrentTerm()` method of the `TermRepository()`, there is a new LINQ query statement item added.

```
//Fisharoo/DataAccess/TermRepository.cs
public Term GetCurrentTerm()
{
    using (FisharooDataContext dc = conn.GetContext())
    {
        Term term = (from t in dc.Terms
                     orderby t.CreateDate descending
                     select t).FirstOrDefault();
        return term;
    }
}
```

Here, you will see that we have an `orderby` clause introduced as well as a `descending` keyword. This allows us to take all the terms ever created and put the most recent ones at the top of the stack.

Implementing the application layer

This layer can be called as application or business logic or components or services layer. In our code, we have called this as **Components**. They are all one and the same. This layer should be relatively thin and lightweight. It is not supposed to hold any business logic or data access logic. It is more of a working layer that is responsible for keeping the business layer easier and cleaner to use. Often, it will combine several items from the business layer and several methods from the data layer to present an easy-to-use interface for a complex task.

An example of this would be our `AccountService`. The `AccountService` provides a few simple methods that utilize several of our more infrastructure-oriented classes. Here is the code:

```
//Fisharoo/Components/AccountService.cs
namespace Fisharoo.Components
{
    [Export(typeof(IAccountService))]
    public class AccountService : IAccountService
    {
        [Import]
        private IAccountRepository _accountRepository;
        [Import]
        private IPermissionRepository _permissionRepository;
        [Import]
        private IUserSession _userSession;
        [Import]
        private IRedirector _redirector;
        [Import]
        private IEmail _email;
        public AccountService()
        {
            MEFManager.Compose(this);
        }
        public bool UsernameInUse(string Username)
        {
            Account account =
                _accountRepository.GetAccountByUsername(Username);
            if(account != null)
                return true;
            return false;
        }
    }
}
```



```
public bool EmailInUse(string Email)
{
    Account account =
        _accountRepository.GetAccountByEmail(Email);
    if (account != null)
        return true;
    return false;
}
public void Logout()
{
    _userSession.LoggedIn = false;
    _userSession.CurrentUser = null;
    _userSession.Username = "";
    _redirector.GoToAccountLoginPage();
}
public string Login(string Username, string Password)
{
    Password = Password.Encrypt(Username);
    Account account =
        _accountRepository.GetAccountByUsername(Username);

    //if there is only one account returned - good
    if(account != null)
    {
        //password matches
        if(account.Password == Password)
        {
            if (account.EmailVerified)
            {
                _userSession.LoggedIn = true;
                _userSession.Username = Username;
                _userSession.CurrentUser =
                    GetAccountByID(account.AccountID);
                _redirector.GoToHomePage();
            }
            else
            {
                _email.SendEmailAddressVerificationEmail(
                    account.Username, account.Email);
                return @"The login information you provided
was correct
but your email address has not yet
been verified.
We just sent another email
verification email to you."
            }
        }
    }
}
```

```

        Please follow the instructions in
            that email.";
    }
}
else
{
    return " Your Password seems to be incorrect. Try
        again!";
}
}
return "Check your Username and try again!";
}
public Account GetAccountByID(Int32 AccountID)
{
    Account account =
        _accountRepository.GetAccountByID(AccountID);
    List<Permission> permissions =
        _permissionRepository.GetPermissionsByAccountID(AccountID);
    foreach (Permission permission in permissions)
    {
        account.AddPermission(permission);
    }
    return account;
}
}
}

```

Let's look at the `Login()` method. It expects a username and a password. From there it fetches the users' accounts by their usernames. It makes sure that the password that was provided matches what we have on the file for that account. It then makes sure that the user has verified their email address, and finally logs in the user.

This method can be extended further to use additional repositories or other services for future needs. The signature of the method could still be just as simple without muddying up the design.

Extension methods

As you probably noticed in the `Login()` method just seen, we had our first introduction to the `Cryptography` class. However, this method is called directly from a string. How does that work?



The subject of Cryptography is an extensive one, and is beyond the scope of this book. However, the `Cryptography` class that is included in this project is heavily commented if you want to understand `System.Security.Cryptography.Rijndael` a bit better! You can find that class in the **Common** project.

To start let's look at how we were able to call `Encrypt()` from a `string` variable. To achieve this is actually very simple. Although the `string` class is sealed, meaning that we can't technically extend it in any way that we are used too, we can use the feature of .NET called "extension methods".

```
//Fisharoo/Common/Extensions.cs
namespace Fisharoo.Common
{
    public static class Extensions
    {
        public static string Encrypt(this string s, string key)
        {
            return Cryptography.Encrypt(s, key);
        }
        public static string Decrypt(this string s, string key)
        {
            return Cryptography.Decrypt(s, key);
        }
    }
}
```

An extension method allows us to extend a class without affecting the way that it would normally work. The way to do this is by defining a static method in a static class. The thing to notice is that the first parameter of the method starts with the target type, a `string` in this case. Therefore we have effectively defined an `Encrypt()` and `Decrypt()` method for the `string` class. Note that the only difference between this method and one you would normally write is the `this` reference preceding the first parameter. It's that simple!

Implementing the domain layer

As we are using Entity Framework, it has facilities that are now part of the .NET framework; our business layer has been greatly simplified for us. We can recall that in our previous applications different sorts of data access layers were used, that required us to spend a great deal of time writing SQL in the database, connection logic, providers, and hydration and persistence logic for objects. In addition to all that, we would still need to define business objects. Of those objects, 95 percent of the logic was simply to shuttle data around in a more manageable manner.

With Entity Framework so much of this has gone away! We now have fully generated classes that take care of shuttling our data around. But what happens if we need custom logic? While we could simply add logic to the generated classes, this would not be the best route. The next time we make a change we will have to regenerate our classes. This would resort in the loss of all that custom functionality.

Fortunately for us the classes that are generated are partial classes. This means that we can make a new partial class file of the same name within the same namespace and extend our generated classes. Here is our custom `Account` object, which extends the generated `Account` object.

```
//Fisharoo/DataAccess/Account.cs
namespace Fisharoo.DataAccess
{
    public partial class Account
    {
        private List<Permission> _permissions = new
        List<Permission>();
        public List<Permission> Permissions
        {
            get{ return _permissions; }
        }

        public void AddPermission(Permission permission)
        {
            _permissions.Add(permission);
        }

        public bool HasPermission(string Name)
        {
            foreach (Permission p in _permissions)
            {
                if (p.Name == Name)
                    return true;
            }
            return false;
        }
    }
}
```

With this new partial class, we can now extend our existing generated `Account` class. We have added a few important features to the `Account` class. We now have methods for adding and checking permissions. We also have a property that returns a list of permissions.

This can easily be done for any partial class in our project!

Implementing the presentation layer

Most of the presentation layer is made up of very standard ASP.NET tools and principles. As this book isn't so much about how a button or label works, we will be focusing more on the non-standard features of our site. We will look at building a scalable UI using the MVP pattern.

Model view presenter

To start with, let's discuss the overall architecture of our presentation layer. We mentioned earlier in Chapter 2 that we will be using the MVP pattern. Information about this pattern can also be found under the separated names of Supervising Controller and Passive View.

The basic reason for this pattern is so that at the end of the day you can wrap a large percentage of your front end code with testing. It also allows you to easily swap out your UI without having to rewrite every aspect of the front end of your application. This can be useful in case we need a new UI or want to target additional devices like mobile. You will also find that this pattern significantly breaks up and compartmentalizes your logic, that makes working on the front end of your application more straightforward.

The MVP pattern in the ASP.NET world basically requires you to have four files (five if you are working in a web application project):

- The design or .aspx file
- Your code behind or .cs file
- An interface that defines the code behind (another .cs file)
- And a class (.cs) file called the presenter, that actually controls everything

Of course, the model portion of this pattern is generally referring to your domain objects that will constitute many other files!

The design file of course holds all your display logic such as a repeater, buttons, labels, and so on. It shouldn't have any server-side logic.

The code behind (or the view) is responsible for handling events from the page such as button clicks. It is also allowed to take care of simple display issues. The view also provides methods to the presenter to toggle the state of the various display items. When the page first starts up (generally on page load), the view initializes the presenter and passes a reference to itself, to the presenter. For every event that is triggered on the page, the view is simply responsible for informing the presenter so that it can decide what to do with the event.

The **View** passes a reference of itself to the presenter by way of the interface that defines the view. Using the interface provides us with a decoupled structure. This is what allows us to easily swap out our UI if we so choose. As long as the UI implements the interface appropriately it can use the presenter.

The presenter is the acting controller in this scenario. Once it is spun up and has a reference to the code behind, it can actively decide how to handle events in the front end. The presenter is also the only part of our front end that is capable of interacting with our domain logic (or model).

In the following sections, we will discuss the login process. This is more to illustrate how the MVP pattern works and less about ASP.NET, as the code itself is very simple.



An **Account** folder was created when we created the ASP.NET 4 application (in Chapter 2). It added basic files related to login, user registration, changing the password, etc. along with the code that uses ASP.NET membership services. However, since we wanted to write custom logic for authentication and authorization, we deleted all files in this folder and wrote custom logic as discussed in the following sections.

View

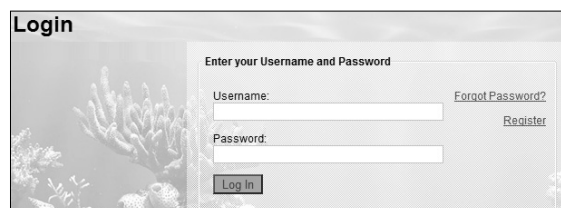
We will start with the front end ASP.NET code. It basically defines a username and password text box and a button to click for login. It also has two link buttons for simple navigational tasks—one to go to the recover password page and another to take you to the registration page.

```
//Fisharoo/Web/Account/Login.aspx
<%@ Page Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="Login.aspx.cs"
Inherits="Fisharoo.Web.Account.Login" %>

<asp:Content ContentPlaceHolderID="Content" runat="server">
    <div class="divContainer">
        <div class="divContainerBox">
            <fieldset class="login">
                <legend>Enter your Username and Password</legend>
                <asp:Table CssClass="tableLogin" runat="server">
                    <asp:TableRow>
                        <asp:TableCell HorizontalAlign="Left">
                            <asp:Panel ID="Panel1"
DefaultButton="btnLogin" runat="server">
                                <p>
                                    <asp:Label ID="UserNameLabel"
```

```
runat="server" AssociatedControlID="txtUsername">Username:</asp:Label>
    <br />
    <asp:TextBox ID="txtUsername"
runat="server" CssClass="textEntry"></asp:TextBox>
    </p>
    <p>
        <asp:Label ID="PasswordLabel"
runat="server" AssociatedControlID="txtPassword">Password:</asp:Label>
        <br />
        <asp:TextBox ID="txtPassword"
runat="server" CssClass="passwordEntry" TextMode="Password"></
asp:TextBox>
        </p>
        <p class="submitButton">
            <asp:Button ID="btnLogin"
runat="server"
            CssClass="loginButton" OnClick="btnLogin_Click"
            Text="Log In" />
        </p>
        <asp:Label runat="server"
ID="lblMessage" BackColor="Wheat" ForeColor="Red"></asp:Label>
    </asp:Panel>
    </asp:TableCell>
    <asp:TableCell HorizontalAlign="Right"
VerticalAlign="Top">
        <p>
            <asp:LinkButton ID="lbRecoverPassword"
runat="server" Text="Forgot Password?" OnClick="lbRecoverPassword_
Click" />
        </p>
        <p>
            <asp:LinkButton ID="lbRegister"
runat="server" Text="Register" OnClick="lbRegister_Click" />
        </p>
    </asp:TableCell></asp:TableRow>
</asp:Table>
<br />
</fieldset>
</div>
</div>
</asp:Content>
```

The login screen will look as shown in the following screenshot:



Normally we would not show you the interfaces in our application as they are very simple. But as the interface is very important to this pattern, we will make an exception this first time! This interface is what the code behind has to conform to in order to be able to interact with the presenter:

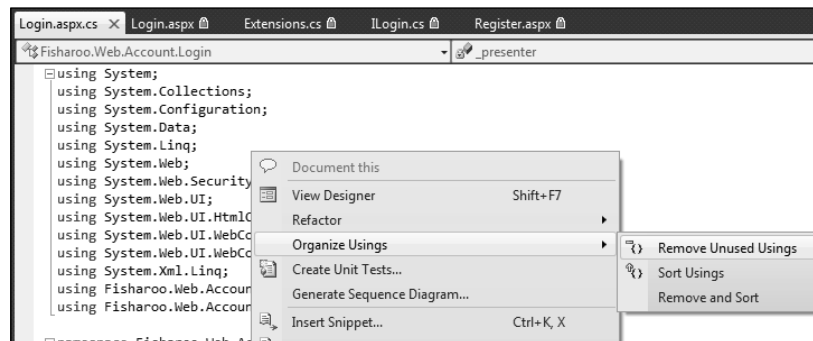
```
//Fisharoo/Web/Account/Interfaces/ILogin.cs
namespace Fisharoo.Web.Account.Interfaces
{
    public interface ILogin
    {
        void DisplayMessage(string Message);
    }
}
```

Here is the code behind for our application. Notice that it only handles display logic. It does not actually make any decisions. It defers all decision making to the presenter.

```
//Fisharoo/Web/Account/Login.aspx.cs
namespace Fisharoo.Web.Account
{
    public partial class Login : System.Web.UI.Page, ILogin
    {
        private LoginPresenter _presenter;
        protected void Page_Load(object sender, EventArgs e)
        {
            _presenter = new LoginPresenter();
            _presenter.Init(this);
        }
        protected void btnLogin_Click(object sender, EventArgs e)
        {
            _presenter.Login(txtUsername.Text, txtPassword.Text);
        }
        protected void lbRecoverPassword_Click(object sender,
                                                EventArgs e)
        {
            _presenter.GoToRecoverPassword();
        }
        protected void lbRegister_Click(object sender, EventArgs e)
        {
            _presenter.GoToRegister();
        }
        public void DisplayMessage(string Message)
        {
            lblMessage.Text = Message;
        }
    }
}
```


You will notice above that in the `Page_Load()` we initialize our presenter. Once we have the presenter spun up, we immediately pass a reference of the code behind to the presenter in the `_presenter.Init(this)` method call. You should also notice that there is a button-click event captured by the code behind. But all that this method does is notify the presenter that it needs to perform the `Login()` method and passes up the raw username and password values. Lastly, notice that the code behind does implement the interface with its `DisplayMessage()` method. As the presenter has access to the code behind class, it will be able to utilize any public method as it needs to.

Another aspect to note is that when you start writing code, Visual Studio will add a lot of using statements on the top of the file. Visual Studio 2010 refactoring support provides for removing unused using statements. You may want to try this to keep things organized.



We did it and ended up with only three using statements as against an earlier list of fourteen statements.

Presenter

Here is the presenter code:

```
//Fisharoo/Web/Account/Presenters/LoginPresenter.cs
namespace Fisharoo.Web.Account.Presenters
{
    public class LoginPresenter
    {
        private ILogin _view;
        [Import]
        private IAccountService _accountService;
        [Import]
        private IRedirector _redirector;
        public void Init(ILogin view)
        {
            _view = view;
        }
    }
}
```

```

        MEFManager.Compose(this);
    }
    public void Login(string username, string password)
    {
        string message = _accountService.Login(username,
                                                password);
        _view.DisplayMessage(message);
    }
    public void GoToRegister()
    {
        _redirector.GoToAccountRegisterPage();
    }
    public void GoToRecoverPassword()
    {
        _redirector.GoToAccountRecoverPasswordPage();
    }
}

```

Note that in the `Init()` method, the presenter sets up the objects that it needs to get its job done by using MEF framework. If a page needs to display data on its initial load, this is where it would happen. Beyond the initialization of the presenter, you will notice that the presenter has three other methods: `Login()`, `GoToRegister()`, and `GoToRecoverPassword()`.

The `Login()` method handles the button-click event that the **View** passes to it. Note that even here we do not have a lot of logic to manage. The presenter is quick to pass off the responsibility of logging the user in to the `AccountService` object that we discussed earlier. It simply expects a friendly message back from the `AccountService` to describe how the login process went. As we know, if it gets a message back, it means that the login failed; otherwise the `AccountService` will redirect the user appropriately. Once the login is complete, the presenter uses the view's `DisplayMessage()` method to inform the user of its status.

The `GoToRegister()` and `GoToRecoverPassword()` methods simply utilize the `Redirector` object to send the user to the appropriate page on the site. While this may seem a bit extreme, remember that it follows the good design principles. If you follow this across your entire site, you will reap the following three benefits:

1. You can easily swap out the UI and expect the same results with minimum efforts.
2. As your redirection code is in one place, when several links use the same method to redirect to a location, you can change this redirection in that place and impact all the links across your site.
3. This aids the testability of your site!

Here are the added Redirector methods:

```
//Fisharoo/Components/Redirector.cs
...
    public void GoToAccountRegisterPage()
    {
        Redirect("~/Account/Register.aspx");
    }
...
    public void GoToAccountRecoverPasswordPage()
    {
        Redirect("~/Account/RecoverPassword.aspx");
    }
...
```



We hope you are noticing that as each file is responsible for a very specific set of tasks, each file is also short and sweet. While this is a complex way of thinking about things, it is very nice to work with!

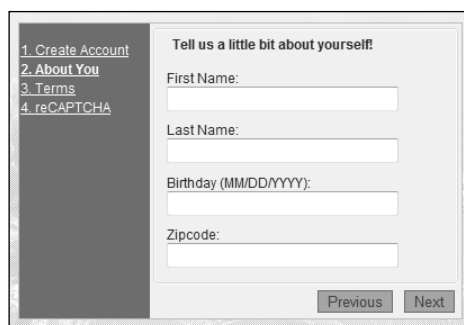
Registration page

We will admit that using the standard .NET controls to create an account is so much easier! Having said that, it was quite a bit of fun creating the registration page for this site. We ended up using a wizard control to display the various steps of the registration process and also used the out-of-box formatting for it. Our steps are as follows (as the ASPX code takes up lot of space, we have put images of how the screens look instead. For code refer to the code associated with this chapter):

1. We always start by grabbing the email, username, and password of the users. There is validation in place for all of these. We want to validate their email addresses for their authenticity. We also want to make sure that their username conforms to some length rules. Then we allow them to enter their password and require them to re-enter their password to verify that what they have entered is what they meant to enter. And of course, all these fields are required!

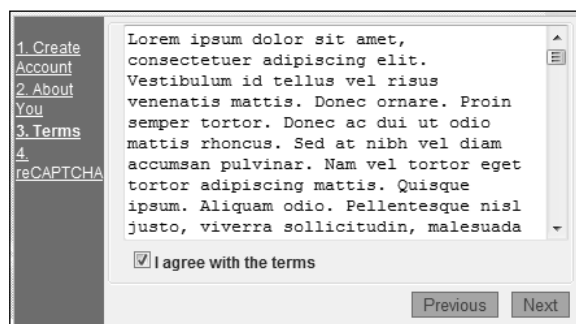
The screenshot shows a registration wizard. On the left, a sidebar lists four steps: '1. Create Account' (highlighted), '2. About You', '3. Terms', and '4. reCAPTCHA'. The main content area is titled 'Let's start by creating your login'. It contains four text input fields labeled 'Email:', 'Username:', 'Password:', and 'Verify Password:'. A 'Next' button is located at the bottom right of the form.

- The next step is to get some descriptive information about the users such as their first names and last names. When building a community site of any type, it is usually important that you also harvest their birthday and zip code or postal code. This lets you know what is appropriate for them and where in the world they are. There is validation in place to make sure that the date of birth they enter is a valid date (MM/DD/YYYY format) and that the zip code is a valid format for US. Though we have validated for only US zip codes, in a community site you will need to validate based on different formats for different countries. All these fields are also required fields.



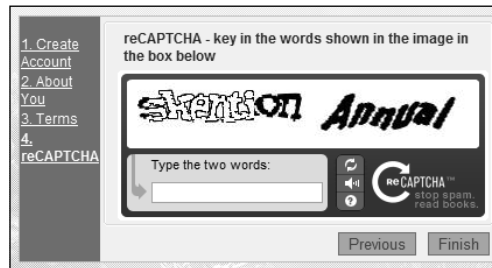
The screenshot shows a web form titled 'Tell us a little bit about yourself!'. On the left is a sidebar with a list of steps: '1. Create Account', '2. About You' (highlighted), '3. Terms', and '4. reCAPTCHA'. The main form area contains four input fields: 'First Name:', 'Last Name:', 'Birthday (MM/DD/YYYY):', and 'Zipcode:'. At the bottom right of the form are two buttons: 'Previous' and 'Next'.

- The third step presents the terms and conditions. This is the one step that requires you to fetch some data for display. This data is retrieved from the `TermRepository.GetCurrentTerm()` (which we covered earlier). All the user needs to do here is read the terms (most of your users won't do this of course!) and check the box indicating that they agree with your terms (we have kept it checked by default).



The screenshot shows a web form for the 'Terms' step. The sidebar on the left has the same list of steps, with '3. Terms' highlighted. The main form area displays a block of Lorem Ipsum placeholder text. Below the text is a checkbox labeled 'I agree with the terms', which is checked by default. At the bottom right are 'Previous' and 'Next' buttons.

4. Finally, we present the reCAPTCHA step so that we can make sure that the person signing up is a person and not a spam bot! Using reCAPTCHA is fairly straightforward. You need to use the ASP.NET specific control that is provided and get the necessary keys from the reCAPTCHA site (<http://www.google.com/recaptcha/whyrecaptcha>).



While we have made this entire information mandatory during registration, you can decide based on your specific needs. There is always a balance required of not overdoing the required information during registration as that may put off some users from registering. Users can always go to the profile page to provide more information once they have registered and logged into the site.

Knowing that the code behind is really just a middle man responsible for passing data to and from the presenter, we are going to show you only the relevant code.

```
//Fisharoo/Web/Account/Register.aspx.cs
namespace Fisharoo.Web.Account
{
    public partial class Register : System.Web.UI.Page, IRegister
    {
        . . . . .

        protected void wizRegister_FinishButtonClicked(object sender,
                                                         EventArgs e)
        {
            _presenter.Register(
                txtUsername.Text, ViewState["password"].ToString(),
                txtFirstName.Text, txtLastName.Text, txtEmail.Text,
                txtZipcode.Text, Convert.ToDateTime(txtBirthday.Text),
                Page.IsValid, chkAgreeWithTerms.Checked,
                Convert.ToInt32(lblTermID.Text));
        }

        public void LoadreCaptchaSetting(bool value)
```

```

        {
            recaptcha.SkipRecaptcha = value;
        }

        . . . .

    }
}

```

The reCAPTCHA validation is integrated with the page validation and hence if `Page.IsValid` returns `true`, we know that the reCAPTCHA validation has succeeded. Also note that for using reCAPTCHA we need to add a new register directive in our ASPX page:

```

<%@ Register TagPrefix="recaptcha" Namespace="Recaptcha"
Assembly="Recaptcha" %>

```

During development we will not want to keep getting the reCAPTCHA validation screen. To prevent this we set the `SkipRecaptcha` property on the control to `false` for the debug configuration. This makes the reCAPTCHA control always return `true` for validation. We will want the control to work properly during final deployment and hence we have used the `web.release.config` file for this purpose. The appropriate tag in that file of interest is:

```

//Fisharoo/Web/Web.Release.Config
<appSettings>
    <add key="reCaptcha" value="false" xdt:Transform="Replace"
xdt:Locator="Match(key)" />
</appSettings>

```

As we publish our site with release configuration, the `Replace` transform mentioned above will replace the value from the main `web.config` file.



We had briefly touched upon multiple configuration files in a previous chapter. This is a new feature for ASP.NET 4 applications with Visual Studio 2010. Appropriate build tasks take care of replacing the values from respective configuration files to the main one. To know more about this read here: <http://msdn.microsoft.com/en-us/library/dd465326.aspx>.

Now we get to the meat and potatoes of this page. Once we have gathered all the data, we need to process it. Enter the presenter (that was fun to say!). We will also focus only on the `Register()` method in the code as that is where most of the action happens.

```
//Fisharoo/Web/Account/Presenters/RegisterPresenter.cs
namespace Fisharoo.Web.Account.Presenters
{
    public class RegisterPresenter
    {
        . . .

        public void Register(string Username, string Password,
            string FirstName, string LastName, string Email,
            string Zip, DateTime BirthDate, bool isCaptchaValid,
            bool AgreesWithTerms, Int32 TermID)
        {
            if (AgreesWithTerms)
            {
                if (isCaptchaValid)
                {
                    Fisharoo.DataAccess.Account acc = new Fisharoo.
                        DataAccess.Account();
                    acc.FirstName = FirstName;
                    acc.LastName = LastName;
                    acc.Email = Email;
                    acc.BirthDate = BirthDate;
                    acc.Zip = Zip;
                    acc.Username = Username;
                    acc.Password = Password.Encrypt(Password);
                    acc.TermID = TermID;
                    //TODO: For development marked as verified already
                    acc.EmailVerified = true;

                    if (_accountService.EmailInUse(Email))
                    {
                        _view.ShowErrorMessage("This email is already
                            in use!");
                        _view.ToggleWizardIndex(0);
                    }
                    else if (_accountService.UsernameInUse(Username))
                    {
                        _view.ShowErrorMessage("This username is
                            already in use!");
                        _view.ToggleWizardIndex(0);
                    }
                }
            }
        }
    }
}
```

You will notice that most of this code is just more validation or navigation logic such as "did they agree with the terms?", or "did they enter the correct reCAPTCHA?". Nothing fancy here!

www.PacktPub.com/asp-net-4-social-networking/book

The new thing here, that is somewhat interesting, is the mention of the `Email` object. You may recall when we built the `Email` object a while back. It provides us with the facilities to send an email in various ways. What we have done here is extended the `Email` object so that it also encapsulates the messages that are sent by the system. Here is the new code for the `Email` object that allows us to send an email verification of the validity and ownership of an email address.

```
//Fisharoo/Components/Email.cs
public void SendEmailAddressVerificationEmail(string Username, string
To)
{
    MEFManager.Compose(this);
    string rootURL = _configuration.GetConfigurationSetting(
        typeof(string), "RootURL").ToString();
    string encryptedName = Cryptography.Encrypt(Username, "verify");

    string msg = "Please click on the link below or paste it into a
        browser to verify your email account.<BR><BR>" +
        "<a href=\"" + rootURL + "Account/VerifyEmail.
        aspx?a=" +
        encryptedName + "\">" +
        rootURL + "Account/VerifyEmail.aspx?a=" +
        encryptedName + "</a>";

    SendEmail(To, "", "", "Account created! Email verification
        required.", msg);
}
```

Also notice the link that is embedded here encrypts the registrant's username with a salt of "verify". This way we know who we are dealing with after they receive the email and follow the link back to our site (more about this in the next section).

In the Wizard's stepped environment, it is very easy to present small chunks of data like this without having too much coding overhead. While it is not as easy as the .NET membership widgets, we think it is quite a bit more flexible. Also, we can easily test this whole process now.

Email verification

We lightly touched upon this subject in the previous section. Basically, the registration process triggers an email to be sent to the newly registered user asking them to verify their email address. This process usually sends an email to the email address that the user provided us when they signed up. If the user can receive the email on their end, then we know that the email address is valid. If they can click on the link that is embedded in the email, then we know that they have access to the email as well. This doesn't necessarily mean that they own the account, but we can't really verify that, and hence we can't really worry about it.

The item we didn't cover above is the page that receives the click from the link in the email. This series of code is relatively simple. So to start, we are going to list all of the files in order of use (design, code behind, and presenter). Then we can discuss it.

```
//Fisharoo/Web/Account/VerifyEmail.aspx
<%@ Page Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="VerifyEmail.aspx.cs"
Inherits="Fisharoo.Web.Account.VerifyEmail" %>
<asp:Content ContentPlaceHolderID="Content" runat="server">
    <asp:Label ID="lblMsg" runat="server" ForeColor="Red"></asp:Label>
</asp:Content>
```

```
//Fisharoo/Web/Account/VerifyEmail.aspx.cs
namespace Fisharoo.Web.Account
{
    public partial class VerifyEmail : System.Web.UI.Page,
        IVerifyEmail
    {
        private VerifyEmailPresenter _presenter;
        protected void Page_Load(object sender, EventArgs e)
        {
            _presenter = new VerifyEmailPresenter();
            _presenter.Init(this);
        }
        public void ShowMessage(string Message)
        {
            lblMsg.Text = Message;
        }
    }
}
```

```
//Fisharoo/Web/Account/Presenters/VerifyEmailPresenter.cs

using System.ComponentModel.Composition;
using Fisharoo.Common;
using Fisharoo.DataAccess.Interfaces;
using Fisharoo.Web.Account.Interfaces;

namespace Fisharoo.Web.Account.Presenters
{
    public class VerifyEmailPresenter
    {
        [Import]
        private IWebContext _webContext;
        [Import]
```

```
private IAccountRepository _accountRepository;

public void Init(IVerifyEmail _view)
{
    MEFManager.Compose(this);
    string username = Cryptography.Decrypt (_webContext.
    UsernameToVerify, "verify");

    Fisharoo.DataAccess.Account account = _accountRepository.
    GetAccountByUsername(username);

    if(account != null)
    {
        account.EmailVerified = true;
        _accountRepository.SaveAccount(account);
        _view.ShowMessage("Your email address has been
        successfully verified!");
    }
    else
    {
        _view.ShowMessage("There appears to be something wrong
        with your verification link! Please try again. If
        you are having issues by clicking on the link, please
        try copying the URL from your email and pasting it
        into your browser window.");
    }
}
}
```

The reason that we listed out the code this way was to show you that all the logic is pretty much lodged in the presenter (as it should be!). Notice that we attempt to get the username from the WebContext (query string in this case) and decrypt it with our "verify" salt. Once we have this username, we attempt to retrieve the Account using AccountRepository.GetAccountByUsername(). If we get an account back, we toggle the Account.EmailVerified property to true and save it back into the repository.

Password recovery

This is another simple page that we can quickly show you the code for.

```
//Fisharoo/Web/Account/RecoverPassword.aspx
<%@ Page Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="RecoverPassword.aspx.cs"
Inherits="Fisharoo.Web.Account.RecoverPassword" %>
```

```

<asp:Content ContentPlaceHolderID="Content" runat="server">
    <div class="divContainerSmall">
        <asp:Panel ID="pnlRecoverPassword" runat="server">
            <fieldset class="recoverPassword">
                <div class="divContainerTitle">
                    Please enter your email address below
                </div>
                <p>
                    <asp:Label ID="EmailLabel" runat="server"
                        AssociatedControlID="txtEmail">Email:</asp:Label>
                    <asp:TextBox CssClass="textRegister" ID="txtEmail"
                        runat="server"></asp:TextBox>
                </p>
                <asp:Button CssClass="recoverPwdButton"
                    ID="btnRecoverPassword" Text="Recover Password"
                    runat="server" OnClick="btnRecoverPassword_Click"
            />
            </fieldset>
        </asp:Panel>
        <asp:Panel Visible="false" ID="pnlMessage" runat="server">
            <asp:Label ID="lblMessage" runat="server"
                ForeColor="Red"></asp:Label>
        </asp:Panel>
    </div>
</asp:Content>

```

We will skip the code for `RecoverPassword.aspx.cs` and `IRecoverPassword.cs` as there is nothing to highlight there and jump to the `RecoverPasswordPresenter.cs`.

```

//Fisharoo/Web/Account/Presenters/RecoverPasswordPresenter.cs
namespace Fisharoo.Web.Account.Presenters
{
    public class RecoverPasswordPresenter
    {
        private IRecoverPassword _view;
        [Import]
        private IEmail _email;
        [Import]
        private IAccountRepository _accountRepository;
        public RecoverPasswordPresenter()
        {
            MEFManager.Compose(this);
        }
        public void Init(IRecoverPassword View)
        {

```

```
        _view = View;
    }
    public void RecoverPassword(string Email)
    {
        Fisharoo.DataAccess.Account account =
        _accountRepository.GetAccountByEmail(Email);
        if(account != null)
        {
            _email.SendPasswordReminderEmail(account.Email,
            account.Password, account.Username);
            _view.ShowRecoverPasswordPanel(false);
            _view.ShowMessage("An email was sent to your
            account!");
        }
        else
        {
            _view.ShowRecoverPasswordPanel(true);
            _view.ShowMessage("We couldn't find the account you
            requested.");
        }
    }
}
```

This page asks the user to provide their email address. It then looks up the account with that email address. If it finds the account it then uses the `Email.SendPasswordReminderEmail()` method to send the user's decrypted password to their email account.

The `SendPasswordReminderEmail()` method looks like this.

```
//Fisharoo/Components/Email.cs
public void SendPasswordReminderEmail(string To,
    string EncryptedPassword, string Username)
{
    string Message = "Here is the password you requested: " +
        Cryptography.Decrypt(EncryptedPassword, Username);
    SendEmail(To, "", "", "Password Reminder", Message);
}
```

Edit account

To save a bit of space we are going to forgo showing you the design, the code behind, and interface portion of this code. It is just a basic form with text boxes and the like, and has the same validation requirements as the registration form did. It allows you to change the information and on successfully saving displays an appropriate message and prevents further editing (user can go to another page and come back here to edit again). There are a couple of aspects however in the `UpdateAccount()` method that we describe below.

```
//Fisharoo/Web/Account/Presenters/EditAccountPresenter.cs

namespace Fisharoo.Web.Account.Presenters
{
    public class EditAccountPresenter
    {
        . . .

        public void UpdateAccount(string OldPassword, string
            NewPassword, string Username,
            string FirstName, string LastName, string Email,
            string ZipCode, DateTime BirthDate)
        {
            //verify that this user is the same as the logged in user
            if(OldPassword.Encrypt(OldPassword) == account.Password)
            {
                if (Email != _userSession.CurrentUser.Email)
                {
                    if (!_accountService.EmailInUse(Email))
                    {
                        account.Email = Email;
                        //TODO: for development disable sending
                        emails, so setting the following to true
                        //in production this will be set to false
                        account.EmailVerified = true;
                        _email.SendEmailAddressVerificationEmail(account.Username, Email);
                    }
                }
                else
                {
                    _view.ShowMessage(false, "The email your
                        entered is already in our system!", account.
                        UserName);
                    return;
                }
            }
        }
    }
}
```

```
        if(!string.IsNullOrEmpty(NewPassword))
            account.Password = NewPassword.
            Encrypt(NewPassword);

        account.FirstName = FirstName;
        account.LastName = LastName;

        account.Zip = ZipCode;
        account.BirthDate = BirthDate;

        _accountRepository.SaveAccount(account);
        _view.ShowMessage(true, "Your account has been
        updated!" , account.UserName);
    }
    else
    {
        _view.ShowMessage(false, "The password you entered
        doesn't match your current password! Please try
        again." , account.UserName);
    }
}
}
```

One thing to notice with the presenter is that when it is first initialized, it loads the current user's account details and passes that data to the view for initial display. Then once the user edits their data, there are several validation steps that occur. The most important is that of the password and the email.

If the password is not changed, we want to make sure that we do not store an empty value to the system!

For the email, if a user changes it, we want to make sure that we resend the verification email again and flag the account as not having a validated email address.

Beyond that we are simply updating the account object via the `AccountRepository.Save()` method.

Implementing security

Now that we have all of our plumbing in place, we are at a point that we can lock down our site. Up until now someone could go wherever they wanted to on the site and we would not be able to stop them at all!

SiteMap

The primary .NET widget that we will use to lock down our site is the ASP.NET sitemap. This is a wonderful tool that can be used not only for security but also to display breadcrumb trails, your primary navigation, and many other useful page/file-oriented tasks.

A sitemap file is made up of several `siteMapNodes`. Each node contains things such as URL, title, description, and roles by default. You can also add your own custom attributes. In our site we will use attributes for identifying links that belong in the topnav, the footer nav, as well as allowing the `siteMap` to help us with each page's title. Our current `siteMap` looks like this:

```
//Fisharoo/Web/Web.Sitemap
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/
    SiteMap-File-1.0" >
    <siteMapNode url="default.aspx" title="Home" description="Home
        page" pageTitle="Welcome to Fisharoo.com!"
        roles="PUBLIC">

<!-- TOP NAV NODES -->
    <siteMapNode url="/account/default.aspx" title="My Account"
        description="" pageTitle="" roles="PUBLIC">
    <siteMapNode url="/account/EditAccount.aspx" title="Edit
        Account" description="" pageTitle="" roles="PUBLIC" />
    <siteMapNode url="/account/Login.aspx" title="Login"
        description="" pageTitle="" roles="PUBLIC" />
    <siteMapNode url="/account/RecoverPassword.aspx"
        title="Recover Password" description="Recover Your
        Password" pageTitle="Recover your password"
        roles="PUBLIC" />
    <siteMapNode url="/account/Register.aspx" title="Register"
        description="" pageTitle="" roles="PUBLIC" />
    <siteMapNode url="/account/VerifyEmail.aspx" title="Verify
        Email" description="Verify your email address"
        pageTitle="Email Verification" roles="PUBLIC" />
    <siteMapNode url="/account/AccessDenied.aspx" title="Access
        Denied" description="Access Denied"
        pageTitle="Access Denied" roles="PUBLIC" />
</siteMapNode>
<siteMapNode url="/profile/default.aspx" title="Profile"
    description="" topnav="1" pageTitle=""
    roles="PUBLIC"></siteMapNode>
<siteMapNode url="/friends/default.aspx" title="Friends"
    description="" topnav="1" pageTitle=""
    roles="PUBLIC"></siteMapNode>
<siteMapNode url="/mail/default.aspx" title="Mail"
```



```
        description="" topnav="1" pageTitle=""
        roles="PUBLIC"></siteMapNode>
<siteMapNode url="/galleries/default.aspx" title="Galleries"
    description="" topnav="1" pageTitle=""
    roles="PUBLIC"></siteMapNode>
<siteMapNode url="/groups/default.aspx" title="Groups"
    description="" topnav="1" pageTitle=""
    roles="PUBLIC"></siteMapNode>
<siteMapNode url="/virtualtanks/default.aspx" title="Virtual
    Tanks" description="" topnav="1" pageTitle=""
    roles="PUBLIC"></siteMapNode>
<siteMapNode url="/forum/default.aspx" title="Forum"
    description="" topnav="1" pageTitle=""
    roles="PUBLIC"></siteMapNode>
<siteMapNode url="/blogs/default.aspx" title="Blogs"
    description="" topnav="1" pageTitle=""
    roles="PUBLIC"></siteMapNode>
<!-- /TOP NAV NODES -->
<!-- FOOTER NODES -->
    <siteMapNode url="AboutUs.aspx" title="About Us"
        description="About Us" footernav="1"
        pageTitle="" roles="PUBLIC"></siteMapNode>
    <siteMapNode url="Advertisers.aspx" title="Advertisers"
        description="Click here to learn more about
        advertising on our site" footernav="1"
        pageTitle="" roles="PUBLIC"></siteMapNode>
    <siteMapNode url="Help.aspx" title="Help" description="Click
        here to enter our help section" footernav="1"
        pageTitle="" roles="PUBLIC"></siteMapNode>
    <siteMapNode url="Privacy.aspx" title="Privacy"
        description="Click here to learn about our
        privacy policy" footernav="1" pageTitle=""
        roles="PUBLIC"></siteMapNode>
    <siteMapNode url="Terms.aspx" title="Terms" description="Click
        here to learn about our terms and conditions"
        footernav="1" pageTitle=""
        roles="PUBLIC"></siteMapNode>
<!-- /FOOTER NODES-->
<!-- NONE NAVIGATION NODES -->
    <siteMapNode url="Search.aspx" title="Search"
        description="Click here to perform a site
        search" pageTitle=""
        roles="PUBLIC"></siteMapNode>
    <siteMapNode url="Error.aspx" title="Error" description="An
        error has occurred" pageTitle=""
        roles="PUBLIC"></siteMapNode>
<!-- /NONE NAVIGATION NODES -->
</siteMapNode>
</siteMap>
```

SiteMap wrapper

As with all of the other controls and classes that .NET exposes to us, it is a good idea to wrap the `SiteMap` class. We did this by creating a `Navigation` class. It not only exposes all the properties that `SiteMap` does, but it also adds a bit more control to the way we interact with our nodes.

```
//Fisharoo/Components/Navigation.cs
namespace Fisharoo.Components
{
    [Export(typeof(INavigation))]
    public class Navigation : INavigation
    {
        [Import]
        private IUserSession _userSession;
        [Import]
        private IRedirector _redirector;
        private Account _account;
        public Navigation()
        {
            MEFManager.Compose(this);
        }

        public List<SiteMapNode> AllNodes()
        {
            List<SiteMapNode> nodes = new List<SiteMapNode>();
            nodes.Add(SiteMap.RootNode);
            foreach (SiteMapNode node in SiteMap.RootNode.ChildNodes)
            {
                nodes.Add(node);
            }
            return nodes;
        }

        public List<SiteMapNode> PrimaryNodes()
        {
            List<SiteMapNode> primaryNodes = new List<SiteMapNode>();
            foreach (SiteMapNode node in AllNodes())
            {
                if (node["topnav"] != null &&
                    CheckAccessForNode(node))
                    primaryNodes.Add(node);
            }
            return primaryNodes;
        }

        public List<SiteMapNode> FooterNodes()
        {
            List<SiteMapNode> footerNodes = new List<SiteMapNode>();
            foreach (SiteMapNode node in AllNodes())
            {
                if (node["footernav"] != null &&
```

```
        CheckAccessForNode(node))
            footerNodes.Add(node);
    }
    return footerNodes;
}
private bool CheckAccessForNode(SiteMapNode node)
{
    if (!node.Roles.Contains("PUBLIC"))
    {
        if (_account != null && _account.Permissions != null
            && _account.Permissions.Count > 0)
        {
            foreach (string role in node.Roles)
            {
                if (!_account.HasPermission(role))
                    return false;
            }
            return true;
        }
        else
            return false;
    }
    return true;
}
public void CheckAccessForCurrentNode()
{
    bool result = CheckAccessForNode(CurrentNode);
    if(result)
        return;
    else
        _redirector.GoToAccountAccessDenied();
}
public SiteMapNode RootNode
{
    get { return SiteMap.RootNode; }
}
public SiteMapNode CurrentNode
{
    get
    {
        return SiteMap.CurrentNode;
    }
}
}
```

All nodes

By default the `SiteMap` class doesn't return all nodes so to speak. It provides you with a call to the `RootNode` and a call to its children. As you can see in our first method, we simply created an `AllNodes ()` call that returns "all nodes".

Navigation

Our site will have several navigation sections. Here we have:

- Top navigation
- Primary navigation
- Secondary navigation
- Left navigation
- Footer navigation

If we had to dig through all of the navigation collections, each time we needed them we may find it quite cumbersome. Instead we will add methods to the classes that produce the required sub-selection of nodes.

The `PrimaryNodes ()` method is the first example of such a method. It produces a list of nodes that go in the primary navigation section by iterating through all the nodes returned by the `AllNodes ()` method looking for each node with a custom `topnav` attribute. You will notice a special filter though in addition to this. With each `topnav` node that is found, a security check is performed to see if the current user should have access to this node. If not, the node is not displayed.

The `FooterNodes ()` method is exactly the same as the `PrimaryNodes ()` method with the exception that it looks for a `footernav` attribute. This method also checks to make sure that the user has access to a specified collection of nodes.

Checking access

This brings us to the `CheckAccessForNode ()` method, that we are using in our other methods. This method looks at the passed in node and checks its `Roles` collection. It first checks to see if the `PUBLIC` role is specified. If so, all remaining checks are not performed. We then move to see if there is an account present, that is, whether any user has logged in. If there is a user, we check their `permissions` property. If that exists, we check to see if there are any permissions in the permission list. We then iterate through each role specified in the node and check to make sure that the account has that permission. If the account doesn't contain any of the specified permissions we return `false`. If all the permissions are valid then we return `true`.

Security

Up until now we have discussed navigational aspects of this class. But seeing how security is rolled into this so deeply, it makes sense that we would also have something to check the current node for security reasons rather than just displaying links. This brings us to the `CheckAccessForCurrentNode()` method.

The `CheckAccessForCurrentNode()` method wraps the `CheckAccessForNode()` method and passes in the current `SiteMap` node. If there is sufficient access to the current node, no action is performed. However, if access to the current node is denied, then the user is automatically redirected to the access denied page by way of the `Redirector` class.

Implementing navigation and security

With this wrapper in place we now have a way to easily restrict where our users go and what forms of navigation they see. All we have to do is make calls into this class to get a list of nodes for the appropriate navigation section. We also need to make a call into the `CheckAccessForCurrentNode()` method at some global point.

In our case these calls will be made from our master page as it controls both global access and navigational display. So the first thing we will do is add a call to the `CheckAccessForCurrentNode()` in the `Page_Load()` method of the `Site.Master.cs` page.

```
protected void Page_Load(object sender, EventArgs e)
{
    _navigation.CheckAccessForCurrentNode();
    ...
}
```

For navigational purposes (not really covered too much to this point) we have a simple repeater that will iterate over `SiteMapNodes`. In the design view we have a repeater that looks like this:

```
<asp:Repeater ID="repPrimaryNav" OnItemDataBound="repPrimaryNav_
ItemDataBound" runat="server">
  <ItemTemplate>
    <asp:HyperLink ID="linkPrimaryNav" CssClass="PrimaryNavLink"
      runat="server"></asp:HyperLink>
  </ItemTemplate>
</asp:Repeater>
```

Then for the `Page_Load()` method, we have the following binding code in the Master page's code behind:

```
repPrimaryNav.DataSource = _navigation.PrimaryNodes();
repPrimaryNav.DataBind();
```

If we only had this code, we wouldn't have any navigation. This is where the `OnItemDataBound="repPrimaryNav_ItemDataBound"` property comes in handy. It basically states that the `repPrimaryNav_ItemDataBound()` method will be our `OnItemDataBound` event handler.

This method will be responsible for displaying all the links. It also takes care of formatting the links to properly show which section you are in.

```
protected void repPrimaryNav_ItemDataBound(object sender,
RepeaterItemEventArgs e)
{
    HyperLink linkPrimaryNav = e.Item.FindControl("linkPrimaryNav")
                                as HyperLink;
    SiteMapNode node = (SiteMapNode) e.Item.DataItem;
    linkPrimaryNav.Text = node.Title;
    linkPrimaryNav.NavigateUrl = node.Url;
    if (node == _navigation.CurrentNode || node ==
        _navigation.CurrentNode.ParentNode)
    {
        linkPrimaryNav.CssClass = "PrimaryNavLinkActive";
    }

    //TODO: For Chapter 3 these links will be kept disabled
    linkPrimaryNav.Enabled = false;
}
```

Summary

In this chapter we implemented user registration. This allowed us to gather data about our users so that they could become a member of our community. In addition to gathering the data, we briefly covered the ways to store some of the more important information. We also created a reCAPTCHA tool to reduce the amount of spam our community would have to deal with. We also provided some tools for the newly registered users so that they could remind themselves of their passwords and edit their account data. Once the registration tools were put in place, we then discussed and implemented an easy way to manage site-wide navigation and security.

With the account creation and management tools in place, we can now move on to other chapters. It was important to get this chapter under our belts as all the following chapters will use many of the features we created here.

Where to buy this book

You can buy ASP.NET 4 Social Networking from the Packt Publishing website:

<https://www.packtpub.com/asp-net-4-social-networking/book>

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.PacktPub.com/asp-net-4-social-networking/book