ASP.NET Content Management System

FHS in Computer Science
3rd Year Project

Candidate No: 28379

Revision 69 – Last Modified on 18 May 2005

Contents

1 Overview

The growth of the internet has resulted in a great wealth of information being made available online that previously would have been found in books and research papers. Unfortunately, the amount of data available is simply so vast, *finding* relevant material in a simple way is becoming increasingly difficult. From my own personal experience of running a website[1] with hundreds of pages of content, viewed millions of times a month, I know that this problem affects those trying to *provide* this sort of content even more acutely.

This problem is compounded for many "community" sites, that have to deal with an added problem of not having a closed group of people contributing to the site – content can come from hundreds if not thousands of different users – some of whom may be totally anonymous, others who may be more "trusted" by the site owner. There are various solutions currently available, but these are either grossly inadequate – in terms of both functionality and customizability - or prohibitively expensive. One such commercial example is a product called SiteCore[2] that lists prices upwards of £10,000; and this for their "basic" package.

This project therefore aims to create a system that addresses some of the shortcomings of these products, targeting the needs of a site that has a large array of content (in terms of quantity, type and subject matter) from numerous sources – whilst retaining as much flexibility as possible to allow it to be customized to individual sites needs.

This poses a few interesting problems

- How we store and *organise* content to make both its management and discovery as easy as possible.
- How we *secure* such a system, and allow multiple users with differing levels of trust access to it.
- How we can *deliver* content to a user – aside from a basic website.
- What design decisions should be made in order to make the system as *flexible* and extensible as possible, both from the information we store, and the way it is presented.
- How we address any *performance* and scalability considerations when dealing with a site with potentially thousands of pieces of content and high traffic loads.

2 Plan

Intending to create a totally comprehensive content management system within the scope of a 3rd year project would be somewhat unrealistic, given the potential range of requirements. Instead, the aim will be to create a framework consisting of the "essential" elements for such a system (identified in the requirements section), with the flexibility to significantly enhance and build on this in the future. Ultimately, my plan will be to introduce the system on my own web site, where the existing system currently leaves something to be desired in terms of functionality, and exhibits many of the problems that we discuss.

A brief note before we begin; a number of standard acronyms are used in this document. If you are unfamiliar with any terms, you may wish to refer to the glossary found on page 26.

3 Requirements

3.1 Content Creation

The term "content" in the context of this system can be somewhat wide-ranging, from formatted text in a variety of formats (LaTeX, PDF, Word, HTML etc) to images and compressed files. However, the key to being able to deal with any of these content formats effectively – from both the discovery and management point of view – is establishing the "meaning" of a document within some context. Unfortunately, such knowledge management systems are still some way off, and so we have to adopt the next best thing. Instead of storing just a chunk of data, we will want to add metadata[3] to add context to the content – whether this is simply the ISBN of a book, a list of keywords or categorization according to some hierarchical structure. Within the site, there may be many different types of content – obviously an ISBN attribute wouldn't apply to something that was recorded as being an article - and yet, many CMS systems don't even provide a way to control the metadata fields, let alone vary them based on the content type.

[1] The site is in fact the only developer community based in the UK, with over a thousand resources and more than 1 million page views per month. See http://www.developerfusion.co.uk/

[2] SiteCore, a .NET Content Management System – see http://www.sitecore.com/

This system will therefore need an effective way to define what metadata we should store for different types of content, and then be able to create and store large quantities of content according to these schemas. Another common problem is ending up being tied to a particular database or method for storing the content, so we should avoid this too.

3.2 Content Management & Categorization

Once the content is in the system, a logical system needs to be in place to manage it. Simply having one page listing all the content stored isn't feasible when you have over a thousand entries! One of way of tackling this is to simply allow the content to be sorted and filtered by different constraints on its metadata; for example, content created by a particular author, or published in a particular date range. In addition to this, we want to be able to organise and categorize content in a structured way, and add relations between individual pieces of content.

3.3 Content Workflow & Control

Some CMS systems assume that a piece of content is simply published on the site or doesn't exist at all – which is a vast over simplification for many sites. Content might need to go through several different stages before being published on the site, effectively a kind of "workflow". This workflow could vary based on the original author, or the type of content, or the user moving the content through the workflow - we don't necessarily want every user of the system to be able to immediately add content, move it through all the stages, and make it appear on the site.

We therefore need to be able to define a workflow from when the content is originally added, through to it being published on the site, and perhaps eventually deleted – and restrict the content to these processes. We might also want some form of versioning control as the content moves through these stages, along with an audit trail so that we can track the users who have modified the content. Finally, we need to prevent two users from editing the same piece of content at the same time, and overwriting each others changes.

3.4 User Authentication, Administration & Permissions

Access to the content management portion of the site obviously needs to be restricted, so that only authorised users may make changes to the system. We not only want to be able to specify that they can access the administration section, but what actions they may perform and which particular sections they can use. When you have a small team or just a single individual running a site - however large - managing this is trivial. However, once you start allowing anonymous users to register – and possibly submit content for review – this becomes far more of an issue. You don't want to have to manually specify permissions as to which parts of the site they are authorised to use for every single user. Nor do you want to be scrolling through a list of 30,000 users to find the one you want – and be limited to searching by username. We will therefore expect the same searching and filtering functionality as we have when dealing with content – allowing us to place constraints on any of the user account fields (which again, should be customizable).

3.5 Content Delivery

Finally, we need a method to make the content stored within our system accessible to outsiders. The most common method for this would be through a standard web site interface, and so it makes sense that the majority of our system's public functionality will be provided through this.

3.5.1 Web Interface

The layout of our interface will need to be fully customizable, as this will effectively establish the site's "brand" – and we don't want look identical to every other site using the system, even if we are providing similar functionality. The interface should also be accessible to as wide a range of web browsers and users as possible[1]. Assuming the visitor arrives at the "home page" of this web interface, we should provide functionality to allow them to find the content they are looking for as easily as possible. This principally includes the ability to list and sort content according to specific metadata, for instance

- content written by a specific author
- content published recently
- all content of type X categorised below node Y written by author Z

[1] Under the 2004 revision of the Disability Discrimination Act it is now a legal requirement in the UK for companies to create sites that are accessible to users with disabilities. Our system should be flexible enough to allow a site to be created within these guidelines.

- content containing a particular set of keywords in its metadata fields

Once a visitor has found the content they are looking for, we obviously want to be able to display it. The layout of the page might need to vary depending on the type of content, and we'll also want to take advantage of any additional information from the back end – such as it's location in the categorization tree. The URL at which the content is accessible should also be meaningful – some CMS systems end up generating URLs such as `/article.aspx?id=3ABCD244-CC7C-4CED-B64E-BCF05191CDAB` which isn't the most memorable of addresses. Content objects may also contain more than one "page" of information, and so we need to provide a way to navigate through these pages within the content object we are viewing.

A simple measure of success for a website is how long an average user remains on the site. One way to maximize this is to display links to some additional related resources also on the site, in case the content isn't necessarily exactly what they are looking for, or if they're interested in finding some similar content. This could be

- Simply related to the categorization of the article – for instance, "this is an article on Java – click here to see other articles on Java"
- Other content (either within the system, or at an external URL), that the author has marked as being directly related to the article.
- Content within the system that has most often been viewed in conjunction with this piece of content.

3.5.2 Alternative Delivery Methods

The web interface is generally targeted by existing CMS systems to the exclusion of all others - but we don't want to be restricted entirely to this. Other possibilities for delivery that we might want to support could include an alternative site for use from a mobile device, providing a notification mechanism for when new content is added to the site, exposing an interface to allow a third party to extract information, or providing our own desktop client through which to access our system.

3.6 Scalability

As mentioned in the overview, this system is going to be targeted at running sites with large numbers of both content and users, and so the solution needs to be scalable for both of these elements.

4 Design

4.1 Content Creation

Each piece of content stored in our system will have a unique identifier, and a set of predefined metadata fields possibly depending on the type of content we're storing. For example, we might require that all content objects consist of a Title, Description and AuthorID, and then specify that an extended Book content type would also store metadata such as ISBN and PublisherID. These specifications should be as easy as possible to create – although for the scope of this project, we won't require that the specification is necessarily modifiable once processed and the data structure in place.

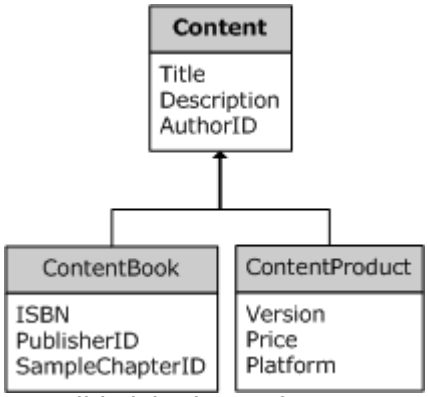


Figure 1 - Possible inheritance from Content objects

We do expect, however, to be able to add, modify and delete these content objects from the system through the presentation layer, into the underlying data store. As we are targeting a web front-end, some of the metadata might be

storing XHTML content, rather than plain text. For these fields, it's unlikely the author will want to write (or even be capable of writing) the XHTML by hand, and so a WYSIWYG interface will be provided for authoring these. We will also provide the ability to dynamically define some relations between users and particular types of content – for example to record the fact that a user has read a particular book, or attended/spoken at/organised a particular event.

Each content item can also have some associated resources – such as images referenced within the XHTML, or an underlying binary file such as a PDF or ZIP file – and so we therefore need to be able to add such binary files to the system. In addition, we will need to be able to add relations between the content objects.

4.2 Content Management & Categorization

In order to organise content, we will provide a non-linear structure in the form of a hierarchical tree, with an unlimited number of nodes, and then allow content objects to be associated with one or more nodes in the tree. In this way, we've then instantly turned what was effectively a linear list (possibly sorted or filtered, but still linear), into a far more manageable hierarchical structure. We will expect to be able to perform many standard tree operations – such as finding a path through the tree to a particular node and changing a nodes parent.

Using this tree and the predefined metadata attributes, we'll provide the ability to search and filter content (for example, "show me all content objects authored by Fred Bloggs that have not yet been approved"), and then edit and modify the content in the underlying data store as required. The interface for editing our content will vary according to the type of content and the metadata it expects.

4.3 Content Workflow & Control

To manage the content workflow through the system, we need to be able to specify what state a content object starts in, what stages it goes through, and who can move it between these stages. For example, we might want to define a process such as this:

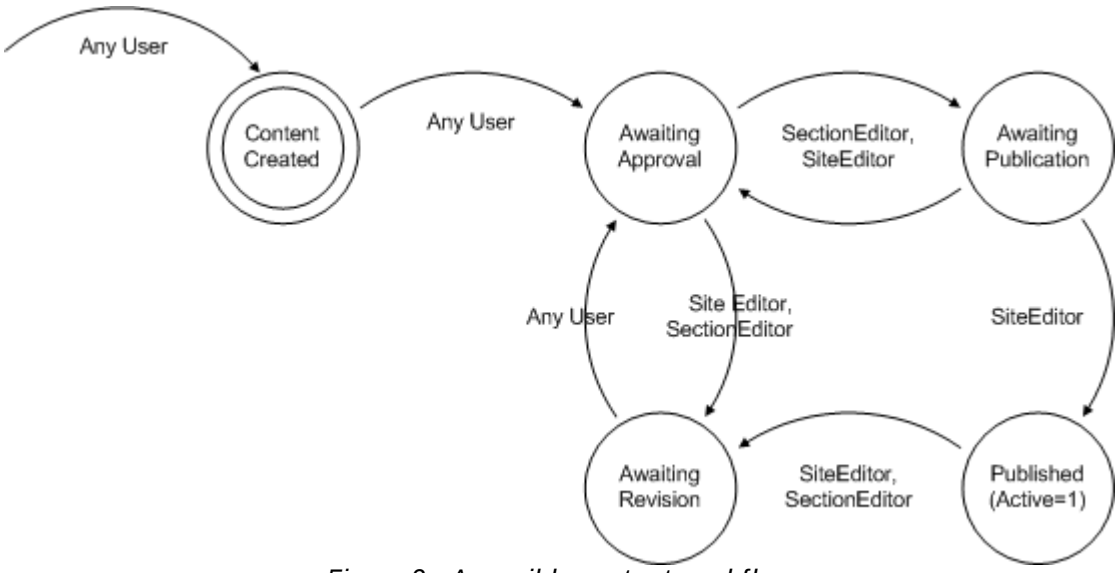


Figure 2 - A possible content workflow

We will therefore store the possible states a piece of content can be in, along with the transitions permitted by particular user roles (discussed below). We will also record content as it moves through the workflow to provide an audit trail. At some future point, we could also save versioning information at each transition so that earlier versions of the content may be restored. To prevent multiple users editing the content, we'll record a timestamp with each request for content, and ensure that this is the latest timestamp issued when an update request is sent back to the system.

4.4 User Authentication, Administration & Permissions

4.4.1 Authentication

In order to access the protected section of the site, the visitor must authenticate themselves by entering a username and password. These account details could be set up in one of two ways:

- By an administrator, from the administration view (see User Administration).
- By the visitor herself, after filling out a "registration" form (if this method was chosen to be permitted by the site owner). This registration process may contain additional checks in order to validate the user (such as checking an email address is valid), and would obviously not default to full administrative permissions.

There will also be a method for regaining access to the system if the user forgets their password.

4.4.2 Permissions

Access to particular sections of the CMS system will require user membership to a particular "role" in the system. Generally, it is expected that roles will be hierarchical – but implicitly, rather than explicitly. For example, a user that belongs to an "Administrator" role would also belong to a lesser "Site Editor" role, and the most basic "User" role. Once successfully authenticated and roles determined, this user will have access to any appropriate sections of the administration system, as configured by their account details.

4.4.3 User Administration

Assuming sufficient permissions, the authenticated user should be able to add, modify and delete user accounts. As with the content, accounts should be searchable and filterable by any metadata for that user (such as the date the account was created).

We also need to be able to edit a user's association with specific roles and the permissions for these roles. Permissions within roles will be explicitly allowed, explicitly denied, or undefined. If undefined, then the user will only be permitted to perform the action if she belongs to another role that explicitly allows the action. Importantly, a user should not be able to grant (or revoke) another user's permission that they themselves do not have (and in addition to this, will also require membership to a role that actually lets them modify permissions in the first place).

4.5 Content Delivery

4.5.1 Web Interface

Providing the web interface ultimately consists of generating a set of plain text HTML tags and sending them to the client - generally a remote machine running a web browser – which is then rendered on their screen. In order to make the site as compliant as possible, we will use the stricter XHTML format, which will also ensure a certain level of accessibility for disabled users. We will also provide a template-based system for customizing the appearance of these pages.

Specific content pages will be accessible, at a minimum, by their unique identifier (such as /article/2039/) but we will also provide a mechanism to map content to simpler addresses (such as /article/beginningasp/).

4.5.2 Alternative Delivery Methods

In addition to the web interface, we will also provide some site functionality through alternative means, which will include

- Exposing the latest content available on the site via an RSS (really simple syndication)[1] feed. Users can then "aggregate" this feed in order to be notified of new content without visiting the site.
- Expose access to a limited set of functionality through "web services" - possibly authenticated - that will allow third party websites, and our own desktop client, to access the web server and retrieve content directly from it.

[1] See Appendix F – References (Web Services and RSS)

4.6 Architecture & Scalability

From the requirements, it is clear that the system needs to be flexible and scalable enough to be able to support various data stores, multiple "views" on this data, and the ability to easily extend the system to store new types of objects. A simple "one-tier" design, with all the data access, logic and presentation code in one place is often a sensible solution, both in terms of speed of development, performance, and maintainability. However, the complexity of these can often spiral out of control as requirements become more demanding. A website may start out only having to deal with 10 concurrent users[1] – but could easily rapidly grow to dealing with 1000 – and suddenly a previously responsive application grinds to a halt. You might be able to upgrade the hardware on the machine running the service – but with everything in the same place, you're limited to running the entire application on one server. Likewise, we may start out only storing 4 different types of object in a database – but suppose it grows to 50, and then we want to be able to have 10 different ways of accessing this information – suddenly, we start losing any logical structure to the system.

An alternative is what is known as an "n-tier" design. This involves the application being split into several different pieces – each performing a distinct task, such as displaying a user interface, or data access. In most situations, applications consist of three such tiers:

- A **Data Access Layer**, which interfaces with our data storage, whether this is XML, a relational database, or some remote web service.
- A **Business Logic Layer** that performs any validation or additional processing required when sending and receiving information from the Data Access Layer. We can also perform caching between here and the Data Access Layer to improve performance.
- A **Presentation Layer**, responsible for displaying information to the user, and allowing the user to interact with the business logic layer.
- The data will be passed between each of these layers using instances of **business entity** classes, which abstract away the underlying data structure to provide a simple data structure with which to work.

The great thing is that these layers don't have to reside on the same server, and we can "plug in" new layers with new functionality, refactor the code in specific layers, or perhaps replace the entire data access layer with one that uses a different data source – without breaking the code in any of the other layers. For a small application, this can be total overkill, resulting in a bloated application, wasted development time, and a system that actually performs worse than the simpler design with a small number of users. Conversely, we can also gain a significant level of abstraction and the flexibility to build on and extend our system in the future, and although we are forced to trade off some raw performance against this level abstraction, we also gain far more scope for scalability by spreading the load over multiple servers.

[1] The number of users simultaneously connected to the server requesting a file or service

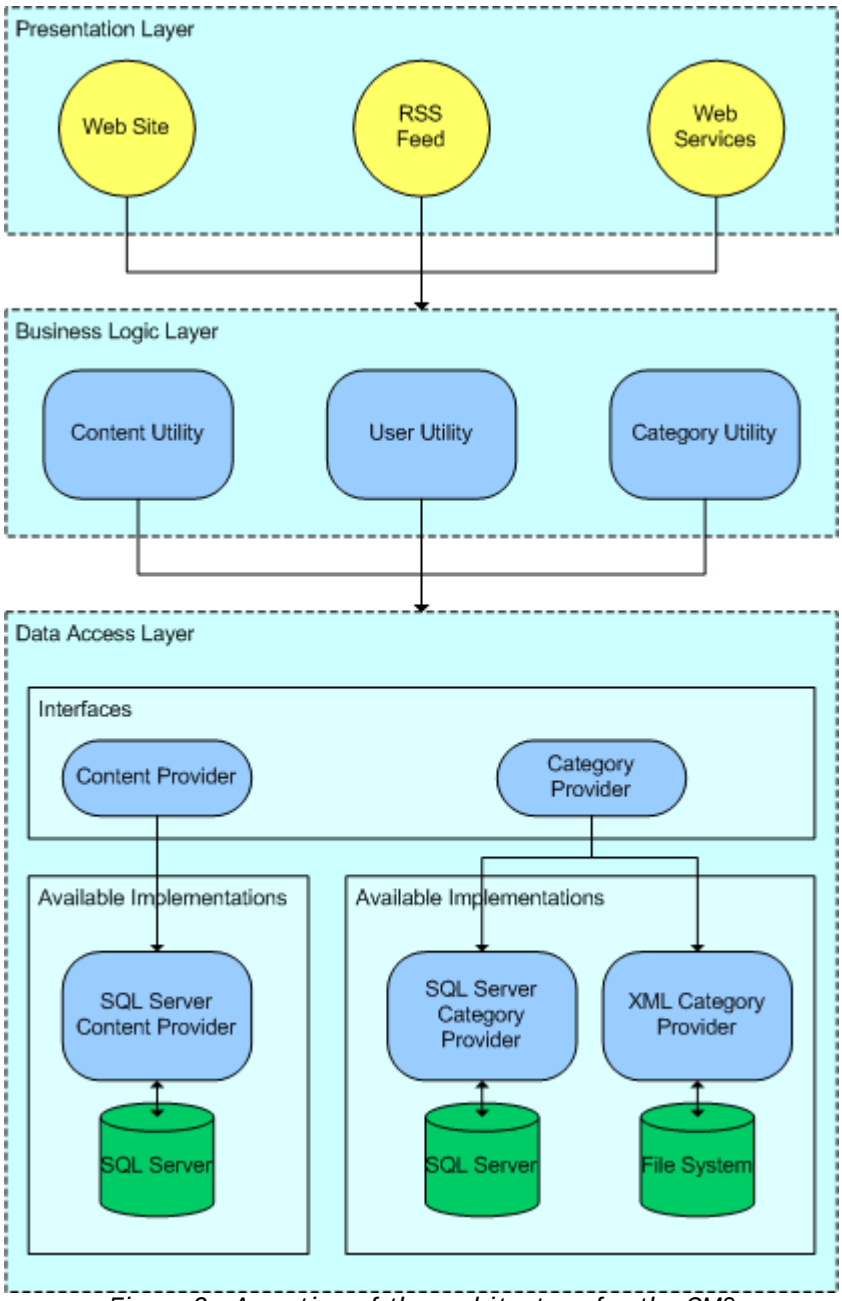


Figure 3 - A portion of the architecture for the CMS

5 Implementation

There are two major application frameworks for creating a system such as this – J2EE (from Sun) and the .NET Framework (from Microsoft) – both of which are object oriented. Which is "best" for the job can become a heated debate amongst their evangelists – but at the end of the day, they both do pretty much the same thing, the issue comes down to what your workforce has experience in[1] – and in my case, this was .NET. Although we could run .NET on a Linux system[2], and use an open source database such as MySQL, we'll simply use the software that .NET is designed to work best with – SQL Server for the data store, and Windows 2003 for the OS. These are actually fairly irrelevant functionality-wise, as these design choices won't limit the platforms from which this system can be accessed.

Although the .NET framework is very powerful, much of its time saving features are only applicable to instances when we are happy to break the n-tier model – for instance binding the presentation layer directly to the underlying data

[1] In a commercial environment, the cost of the relevant platform pales to insignificance against the cost of retraining your staff in a new language or technology.

[2] There is actually an alternative Linux implementation of the .NET Framework called Mono, supported by Novell. See Appendix F – References (.NET Framework)

source. As such, adopting the n-tier model does add a considerable amount of additional work. However, we will offset some of this work by building a system to automatically generate the "standard" segments of code for each of the tiers.

5.1 Business Entities

In our CMS system, each of our business entities will be "light-weight" classes – we will not be including any processing logic whatsoever. For specifications of the business entities we shall be using, see Appendix D – Business Entities. Each of these business entities will contain the appropriate properties representing the information we wish to store, and will inherit from either another business entity or a `CommuneObject` class, which defines a common `UniqueID` property. At both ends of the n-tier model (the presentation and data access tiers) we will need to dynamically get and set properties based on a string. Although .NET has a built-in feature called reflection[1] for dynamically calling methods, setting properties etc at runtime, these can be very expensive operations. We will therefore include two methods for dynamically getting and setting properties, and one for simply fetching the list of available properties. These will be overridden by each inheritor of the class to extend the same functionality to the additional properties the object defines. As we are going to be automatically generating our business entity classes (discussed later in this section), including these properties involves minimal additional work, whilst significantly improving performance.

5.2 Data Access Layer

The data access layer (or DAL) is responsible for dealing with our underlying data store – in this instance, Microsoft SQL Server. However, one of our requirements was that we don't want the entire infrastructure of the system to be tied to this specific product. To this end, the data access layer will consist of two parts.

The first is a set of interface classes defining a set of standard data access methods for each of our business entities. As defined in the specification, there are a number of operations we expect to be able to perform for all of our business entities, and so we will define a generic[2] interface `IObjectProvider<T>` (where T is a CommuneObject), consisting of the following (in C#):

- `T CreateDefault();`
- `T Get(int uniqueID);`
- `int Add(T obj);`
- `bool Update(T obj);`
- `bool Remove(int uniqueID);`
- `List<T> GetAll();`
- `List<T> GetAll(string sortExpression);`

One other thing to note is that we are providing the option for an implementation of the GetAll method that accepts a sort expression. In general, we try to perform sorting as close to the data access layer as possible – either within the database itself (assuming we are using a database), or before the data is returned to the business logic layer.

The interface for that specific object will include any additional operations we expect specific to that object, such as

```
List<Content> GetContentByUserID(int userID);
```

for the Content objects.

The second part of the data access layer will be an implementation of the interfaces for a specific database system - in this case, SQL Server, the specific implementation details of which will be discussed later. The idea is that the person developing the data access layer can simply implement these interfaces for whichever data store they wish, allowing this new data layer to be "plugged in" without the application even knowing that it's now pointing to a new source for its data. It will be our business logic layer that is responsible for selecting the appropriate data access layer, and interacting with it.

5.3 Business Logic Layer

The business logic layer (BLL) is responsible for interfacing with the DAL, and essentially acts as a go-between for the presentation layer and data access layer - providing any additional logic and validation required with the business entities, such as requiring a non-zero UniqueID of the business entity before attempting an update.

---

[1] See Appendix A – The .NET Framework (Reflection)
[2] See Appendix A – The .NET Framework (Generics)

We will have a "Utility" class provided for each of the business entities – this time consisting of a set of static methods providing the expected operations for each entity. Each of these utility classes will then inherit from an abstract utility class that provides the common operations, as we did in the data access layer. This abstract class accepts two type parameters – one for the type of the business entity, and another for the DAL interface we are dealing with.

```
public abstract class CommuneObjectUtility<T, P> where T : CommuneObject
                                                 where P : IObjectProvider<T>
```

This utility class supplies a "Provider" property that returns a concrete implementation of the DAL interface, which is then used both for the standard operations (fetching, updating etc), and by the business entity-specific operations in the business logic layer. This also provides the presentation layer with direct access to the DAL in any situation where there is no business logic necessary.

In order that the business logic layer knows which concrete implementation of the DAL interface to load, we take advantage of ASP.NET's ability to store configuration information in a special XML file. For example, in order to instruct the BLL to use the SQL Server implementation for storing our user information, the XML file would contain a line that looked something like this:

```
<add key="UserProvider" value="vCommune.Database.SqlServer.UserProvider,vCommuneSqlServer"/>
```

The BLL then loads the value of "UserProvider" from the configuration file[1], which represents the namespace, type name and assembly in which the implementation resides. Mentioned earlier was a feature called reflection, which allows us to perform various dynamic type operations. Although we chose not to use this Reflection feature for dynamically setting properties on our business entities, the performance hit of "reflecting" a type once, and then storing a reference to it, is minimal. So, in this instance, we can use reflection to load the UserProvider type from the configuration setting, check that it implements the expected interface, and then save a reference in a cache for future use.

5.3.1 Data Caching

Roundtrips to the database are relatively expensive, and so our business layer will also be responsible for sensibly caching data, so that it can avoid requests to the DAL when possible. This is actually fairly straightforward to implement, provided the only changes to the underlying structure in the database will be made through this layer. Of someone sends a User object to update, for instance, we know we should invalidate our cached version of that object with that particular identifier. When using the .NET caching features, we can also specify that an object is automatically removed from the cache if it hasn't been accessed for a particular length of time – ensuring we aren't unnecessarily storing infrequently used data in the memory.

5.4 Object Templating

When writing an n-tiered system, especially one that integrates with a relational database as its data store, a large quantity of code that needs to be written is simply "plumbing", with a few special cases. We will want to be able to create, modify, delete, and fetch objects – and all these operations have corresponding occurrences in each of the tiers. To this end, a templating system will be created to automatically generate this plumbing, both for the different tiers, and for any database-specific code we must write. This will also allow us to satisfy the requirement of being able to easily define content types and specifications.

Each type of business entity we wish to store will have an XML file defining its properties, and their corresponding data types. Our requirements defined a number of one-to-many relationships, including:

- Each content object is associated with a particular author
- Each content object can have multiple content pages

We therefore will need to be able to record these relationships in our XML definitions. Furthermore, we may need to store additional information specific to a data provider – for example, using SQL Server we also need to specify a type size for some data types, and in addition, we can't always infer the best SQL Server data type to store the information from the corresponding .NET data type in the business entity.

[1] Our abstract class would know we were looking for a User provider from the name of the type T that was provided by the type parameters.

Below is a basic example of such an XML file:

```
<?xml version="1.0" encoding="utf-8" ?>
<CommuneObject>
    <Name>Content</Name>
    <Properties>
        <Property Name="Title" TypeCode="String" />
        <Property Name="AuthorID" TypeCode="Int32">
            <Relation Object="User" />
        </Property>
        <Property Name="DateCreated" TypeCode="DateTime">
            <ProviderInfo Name="SqlServer" TypeCode="SmallDateTime" Default="GetDate()" />
        </Property>
    </Properties>
</CommuneObject>
```

In order to generate the files we require, one obvious option would have been to apply various XSL[1] transformations to the file. However, the current specifications for XSL are still lacking in a number of features, and even basic things such as fetching the current date/time have to be passed as parameters by the caller. We therefore decided to go for a more flexible option, and take advantage of ASP.NET's[2] own processing power. We can then define a series of ASP.NET pages, acting as templates that generate C# and SQL rather than HTML.

The question is how to actually retrieve this generated code after being processed by ASP.NET. One simple option would be for our template generator to simply request a page using HTTP (as a remote web browser would), and saving the result to a file instead. However, in order for the templates to work we need to pass some information to the templates as to the object we are generating – we'll use an instance of a class representing the particular XML object definition we're processing, which will be loaded using XML serialization[3]. Although HTTP has some basic mechanisms for transmitting data to a particular page, we can't easily turn the data back into a useable .NET object. We therefore interact directly with the ASP.NET runtime, and convince it that our own application is in fact a web server requesting it to render a page. As a result, we can send a page request - along with any contextual variables we need - and retrieve the processed text returned, without placing these files within an actual web server.

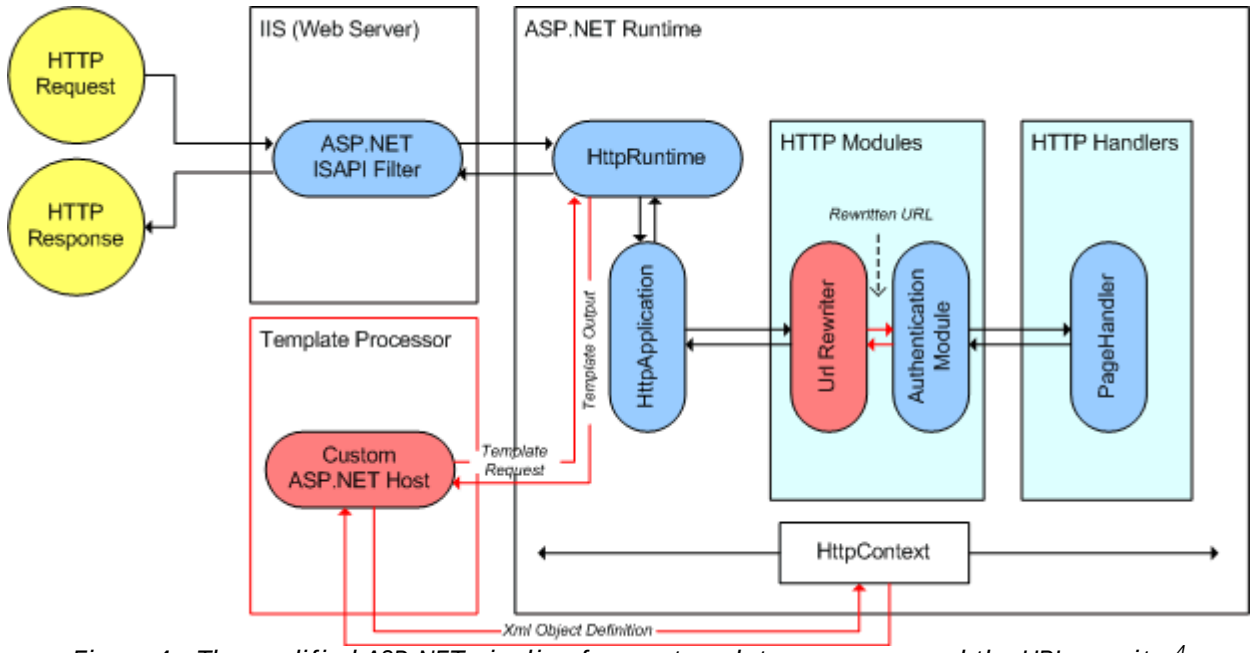


Figure 4 - The modified ASP.NET pipeline for our template processor, and the URL rewriter[4]

[1] XSL is a language for transforming XML documents
[2] ASP.NET is the .NET framework's engine for web based presentation layers (rather than windows client based). See Appendix B – ASP.NET
[3] See Appendix F – References (Serialization in .NET)
[4] See Presentation Layer (URL Rewriting)

Using this technique, we can automatically generate

- A business entity consisting of all the appropriate properties and data types, and the necessary code for those methods that allow dynamic fetching and setting of the properties.
- The appropriate DAL interface that any provider for this business entity must implement. The automatically-generated portion of this will include standard Get, GetAll, Update, and Remove operations, along with any relation-specific operations such as "GetContentByUserID".
- The "utility" class in the BLL that will instantiate an instance of a provider that implements the interface defined above.

We will also use this system to automatically generate some database-specific "plumbing", which includes

- An implementation of the provider interface for each object, specific to SQL Server.
- The SQL necessary to create database tables to store our objects, and the stored procedures to fetch them from the database.

Any additional operations that are required (and have to be coded manually) are defined in a separate file, and joined at compile time with the generated portion using partial classes[1].

Although outside the scope of this project, there could be the possibility of making incremental changes to the system structure – such as adding new fields to our business entities, and content types - and generating appropriate scripts to make the necessary changes to the database, without losing the existing data.

5.5 Microsoft SQL Server Provider

The implementation of our SQL Server provider for the DAL interfaces consists of two parts. First, we need to define the actual .NET classes necessary to implement the DAL interfaces. Along a similar vein to the DAL and BLL, we again take advantage of an abstract base class `ObjectProvider<T>` that takes a type parameter T and provides a standard implementation of `IObjectProvider<T>` from the data access layer interfaces. The class includes methods

```
virtual List<SqlParameter> GetSqlParameters(T obj, QueryType queryType)
abstract String GetObjectName();
```

These are implemented by each concrete implementation of ObjectProvider in order to allow the base class to implement the standard operations – which it can't do without knowing the database-specific parameters for that object. These classes interact directly with the database using the SQL Server-optimised data access libraries that .NET provides.

5.5.1 Basic SQL Queries

Alongside our .NET provider classes, we have a set of SQL statements to create an appropriate database structure, and perform predefined queries[2] (known as stored procedures), with each operation defined in the DAL interfaces having a corresponding stored procedure (as a minimum, the standard Add/Delete/Update/Get/GetAll operations). Our provider class then calls these procedures passing the appropriate parameters without worrying about the specific implementation details. The use of these stored procedures and parameters also protect the system against SQL injection attacks[3]. Again, this is all automatically generated using our templating system, with a few additional operations for special cases where needed.

5.5.2 Dynamic Relations

We saw earlier that the requirements suggested a number of one-to-many relationships between particular objects, which were defined in our XML object definitions. There are also a number of many-to-many relationships in our requirements, which we represent using an intermediary table[4] in the database.

A more interesting case is when we want to define "dynamic" one-to-many relations between User objects and Content objects. This will allow the administrator to easily define relations such as "Attended (Event), Presented At (Event), Recommends (Book), Reads (Blog)" and then allow their visitors to add relationships between themselves on particular content objects – restricted depending on the content type.

---

[1] See Appendix A – The .NET Framework (Partial Classes)
[2] See Appendix C – Relational Databases and SQL Server (Structured Query Language (SQL))
[3] See Appendix C – Relational Databases and SQL Server (SQL Injection Attacks)
[4] See Appendix C – Relational Databases and SQL Server (Many to Many Relationships)

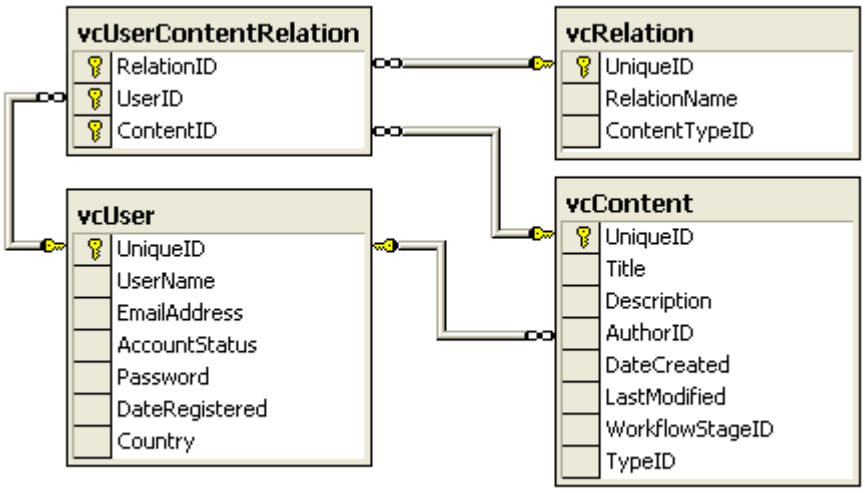


Figure 5 - Dynamic relations between Users and Content

5.5.3 Storing Trees

We have two non-linear data structure types that need to be represented in our relational database: the first, a tree representing the category hierarchy. Relational databases don't generally deal with hierarchical information very well. Therefore, we are forced to store our data structure in a linear format, with enough information to be able to restore the structure – and perform meaningful queries over the structure with minimal overhead.

The naïve method of a "parent" relation requires extensive use of recursion. After evaluating a number of possibilities[1], it was decided to perform a "modified pre-order traversal" of our tree, which associates two integers "left" and "right" values with each node.

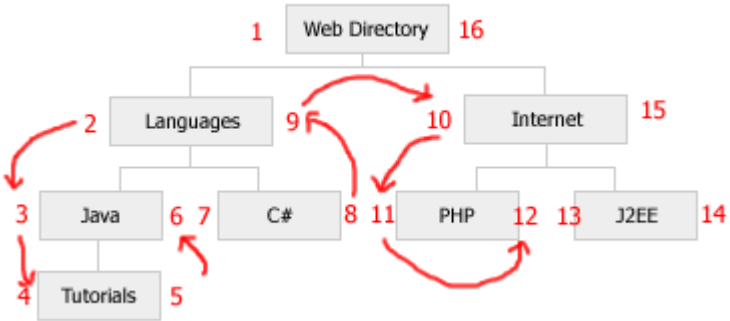


Figure 6 - A possible tree annotated with the pre-order traversal

These values are maintained by the stored procedures we have to call in order to add nodes to the tree. Although this is not the immediately obvious way of representing a tree in the database, and we're forced to recalculate a large number of these node values when modifying the tree (adding a new node to the left of the root would result in the left and right values of all nodes in the tree being updated), the majority of queries – fetching information from the tree – can be performed with a straightforward SQL statement. In addition, we gain an explicit ordering over the nodes (unlike an adjacency list model), and have minimal additional information to store.

5.5.4 Storing Finite State Machines

The second non-standard data structure to be stored in the database is our finite state machine representing the content workflow process. The solution to this is pretty much implied in the design – we shall use one table to represent the different states in the finite state machine, and another to represent the valid transitions between states.

---

[1] See Appendix C – Relational Databases and SQL Server (Storing Trees) for details of alternative methods, how we maintain this structure, and some example queries over the structure.

It is more than likely, given most editorial processes, that the finite state machine we're storing will contain cycles. However, at any one time, we will only be performing a single transition, controlled entirely by the user, and so we don't need to worry about the potential for infinite loops.

5.5.5 Storing Passwords

In most instances, the underlying type used by our storage mechanism (be it XML, SQL Server etc) is essentially the same as that of the one defined in the business entity[1]. However, when it comes to storing the password field of our User object, storing it as plaintext in each database record isn't a good idea – instead, we store a 128-bit hash[2] of the password. Obviously, if we're only storing the hash of the password, then we lose the original password itself – however, this doesn't stop us from implementing all the required operations.

- In order to check whether a user has entered the correct password, we simply create an MD5 hash of the password that has been entered, and checks whether it matches the one in the database.
- Although we can't allow the user to retrieve their original password if they forget it, we can simply generate a new random password, and send that to their registered email account. After logging in using their new password, they can then change it if they so wish.

5.5.6 Class Hierarchies

Storing and updating our content objects also provide some interesting problems. For the other database entities we are storing, the provider being used knows exactly what type of object it is fetching, and what properties it contains. However, when requesting a Content object from the database, neither the calling object, nor the provider initially know what (sub)type of content is going to be returned – and therefore which Content class to instantiate for whatever additional metadata is being stored.

In order to implement this, the stored procedure for fetching content checks the content type for the item we are requesting, and then performs a query on the appropriate view in the database (joining the main content table with the appropriate table that stores the additional metadata, along with a table storing information about the different content types and their corresponding business entities). The Content provider in the DAL then checks the ContentType field, instantiates the correct Content object using reflection[3] (again, caching the type), and populates its fields with the ones returned by the stored procedure.

[1] Although in theory XML is storing everything as strings, the schema for that XML file would specify the data types such as integers.
[2] See Appendix C – Relational Databases and SQL Server (Storing Passwords)
[3] See Appendix A – The .NET Framework (Reflection)

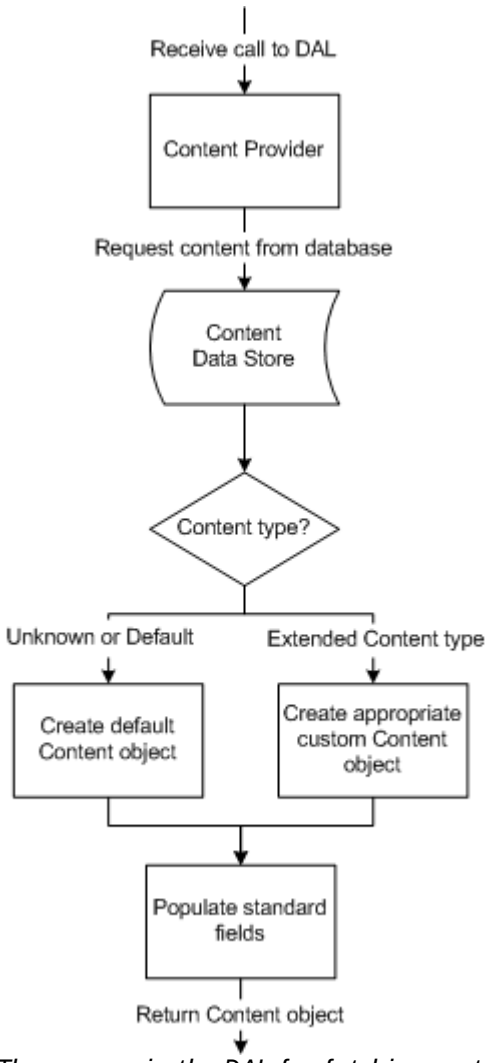


Figure 7 - The process in the DAL for fetching content objects

5.6 File System Provider

For the binary resources related to our content objects, it doesn't generally make sense to store these in a database, simply due to the quantity of data likely to be stored. Therefore a portion of the DAL interfaces were actually implemented in a separate File System provider (which can just be plugged in thanks to our extensible model).

5.7 Presentation Layer

In order to generate the HTML we need for the web interface portion of our application, we'll use the ASP.NET portion of the .NET framework[1].

5.7.1 Two-way Data Binding

One of the most common themes when dealing with a data store and a web interface as the "front-end" is the process of getting the data from the data store into some input fields in a web interface – and fetching them back again in order to update the data store. In ASP.NET this is known as "two-way data binding", and provides a number of controls to make this process a little easier – allowing you to "bind" particular input fields (such as a text box, or a drop down list) to a field from some object – often a direct result set from a database (which would seriously break our tiered model), or alternatively a business entity.

The downside of these is that the controls it provides require to be told exactly what type to load and what methods to call for each of the add, update, etc operations – and then uses reflection in order to perform these operations. Given the performance implications of reflection, and the fact that we actually know this information by inspection (as we've generated the entire set of objects), we extend these controls with a "GenericDataSource" that accepts a type parameter holding the business entity type, and automatically provide the necessary code to statically bind these properties. The end result is much better performance – and much less code in the ASP.NET pages.

5.7.2 Page Templates

In addition to the "dynamic" portions of the HTML we're generating, there will also be a consistent set of HTML that we generate with minimal changes for every web page on the site – a standard header and footer, if nothing else. We'll use a feature known as "MasterPages" which allows us to define a standard template for a portion of the site, which pages then automatically load – inserting content into appropriate places such as the main body of the page. These can be configured using a site-wide (or directory specific) XML configuration file[2].

5.7.3 Authenticated Access

There are two basic scenarios in which our web application might be accessed – anonymously, or authenticated. In order to authenticate themself, the user will enter their username and password via a login page. The details they submit will be forwarded to the BLL to be validated, and if successful the roles they belong to also returned. The user is then granted a security token which is stored in the form of a cookie[3]. This token contains a cached list of the roles the user belongs to (updated periodically through the users session), protected by encryption and a MAC (to prevent chosen plaintext attacks). We can then allow access to particular sections of the site according to the roles recorded in this cookie.

5.7.4 Retrieving User Passwords

If the user has lost their login password, then we needed an easy way to let them regain access. Some sites solve this by forcing users to specify a secret question and answer to validate their identity, and then let them choose a new password – but this just opens up another line of attack in that the answer to this question is also likely to be obvious. Instead, we're going to simply send a new password on request to their email address. Although this would allow a third party to force a new password to be generated for another account, security is maintained – in that only the "real" user will receive the notification – assuming their email account is secure.

In order to add some protection against a scripted attack that could try to enter hundreds of email addresses (perhaps scraped from the website itself) in an attempt to just flood all the users of the system with new passwords, we will use a system known as CAPTCHA. This is a test essentially designed to check whether or not a user is human, by displaying a slightly distorted image with a collection of words and numbers, and asking the user to enter what they see. Obviously

[1] See Appendix B – ASP.NET for an introduction to how ASP.NET pages work
[2] For more information on Master Pages, see Appendix F – References (ASP.NET Controls)
[3] A "cookie" is a small piece of information that a browser allows a website to store on the visitor's machine to maintain state across multiple pages – without introducing any additional performance problems at the server side for potentially storing thousands of these.

this isn't foolproof, and there's a trade off between the level of distortion in the image (to protect against an attacker using optical character recognition), and the difficulty a real human has reading them.

Figure 8 - The CAPTCHA lost password form, used for protection against automated "bots"

5.7.5 Tabbed Dialogs

Just as you would define a graphical component in Java to appear in a window, ASP.NET allows you to define "Web" control classes that render themselves as HTML when requested. In order to provide the GUI for the application, we shall create a number of these re-usable controls.

As our administration system needs to provide access to a reasonable number of pages, we created a tabbed based system. Our TabControl loads an XML definition file, determines the currently selected tab from the URL, and displays the appropriate child tabs. This makes it incredibly easy to provide a consistent – and easily customizable – interface for users[1]:

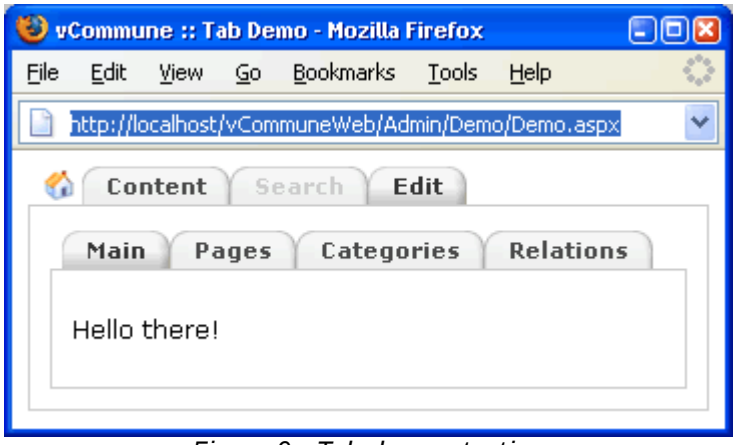


Figure 9 - Tab demonstration

5.7.6 Dynamically Loading Content Templates

When providing our content editing UI, we also need to dynamically load the appropriate fields for editing content based on the content type that has been selected – in the case of the figure below, the ISBN.

[1] For a demonstration of the XML file and an ASP.NET page using the control, see Appendix E – Code Snippets (Tabbed GUI Control)

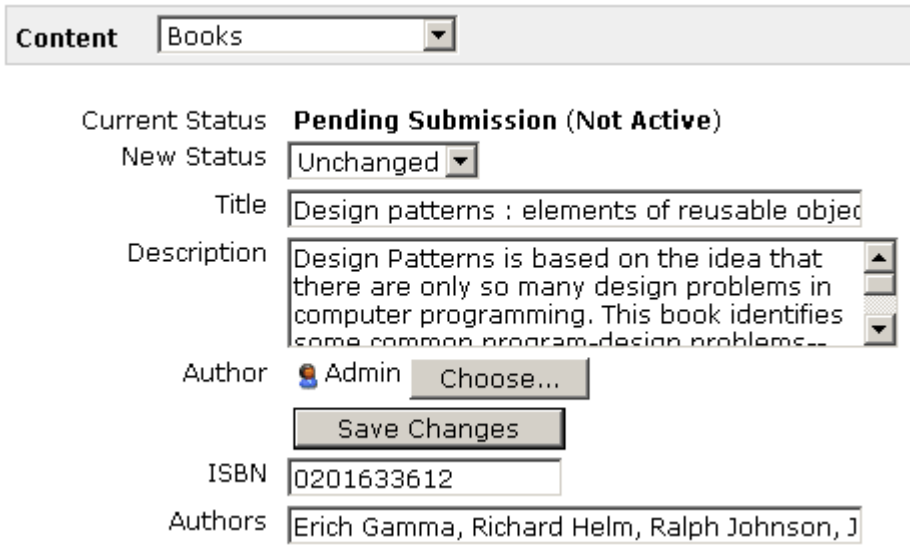


Figure 10 - Dynamically Loading Content Templates based on a "Book" content type

5.7.7 Concurrency

Users are prevented from making changes to a Content element when it has changed since it was fetched from the system. This is to prevent an older version of the Content accidentally overwriting a new one (that was either created by themselves or another authorised logged-in user).

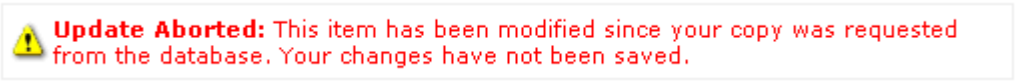


Figure 11 - Message displayed when an item has been modified between requests

5.7.8 Tree View

In order to manage our category hierarchy, we obviously need a graphical representation of the tree, and so we created a control that fetches the entire tree structure from the database, and renders children appropriately[1].

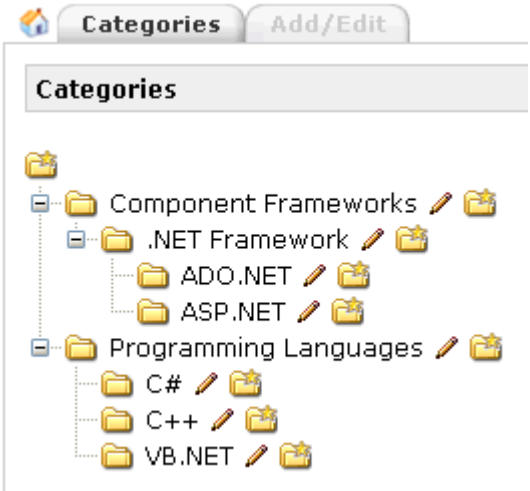


Figure 12 - Editing the category hierarchy

In addition, we need a slight variation on the control to allow content to be associated with particular nodes in the tree, as shown below. Instead of having to make separate database roundtrips for each item selected, we send a comma delimited list to the database of selected categories, which then parses these and executes the appropriate SQL statements[2].

---

[1] In order to produce this, we use a stack-based algorithm to process the rows from the database and display the appropriate graphics

[2] For details of this process, see Appendix C – Relational Databases and SQL Server (Passing Lists to Stored Procedures)

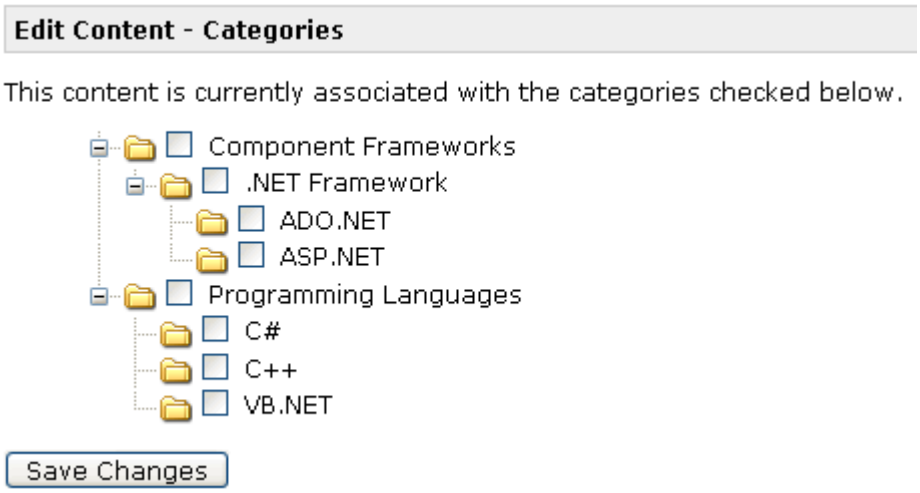


Figure 13 - Picking Category Associations

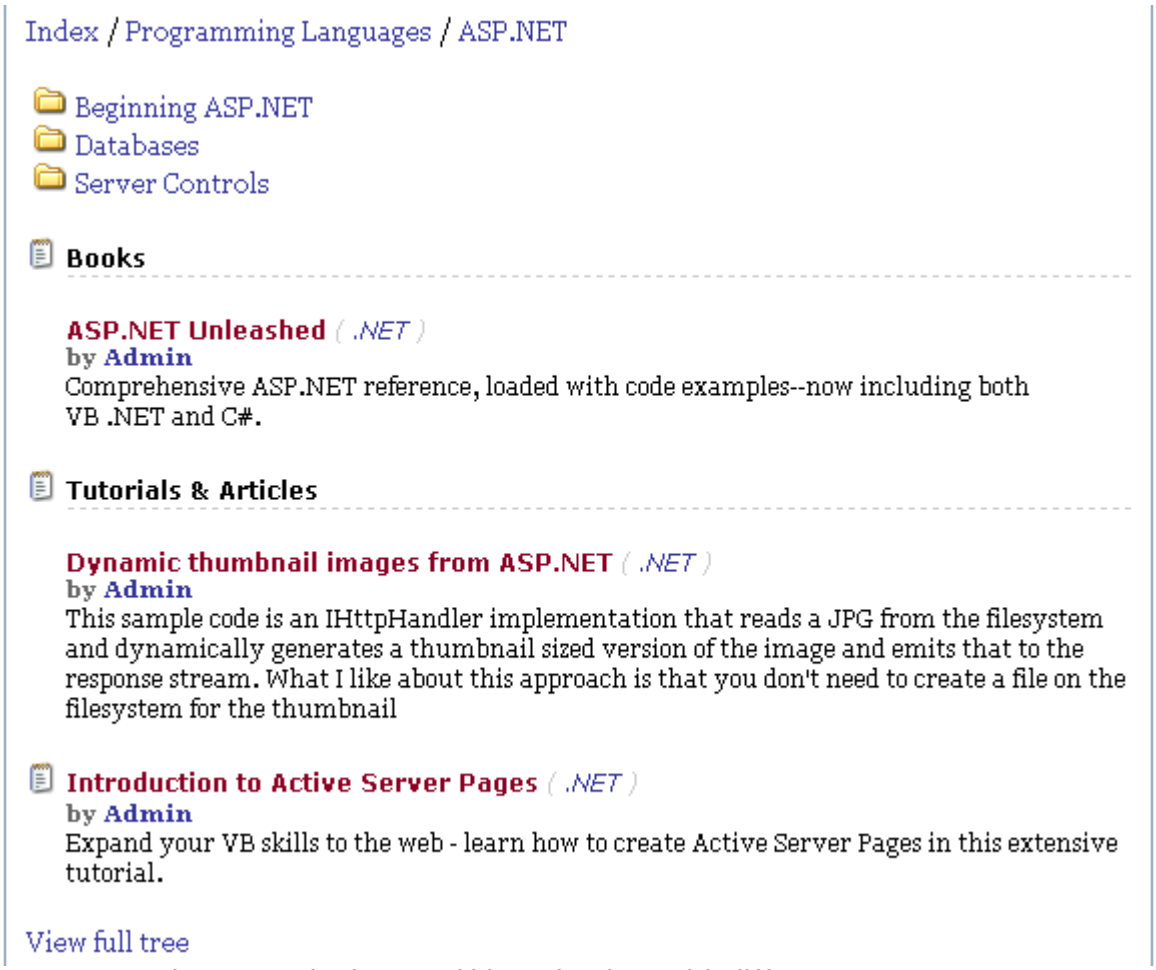


Figure 14 - The front-end hierarchy view, with different content types

5.7.9 Relation Picker

Not all of the fields that we are providing for editing are simply text based; many are relations pointing to other objects. Although in theory we could simply provide a text box and expect the user to enter the associated identifier for the object (as is required for the database-entity fields), this isn't particularly user friendly. Nor do we want the user to have to navigate to another page, losing any changes they may have made in the current window, in order to select this object.

In some situations, if the choice of potential objects is small (such as a user role, perhaps) then a simple drop-down box populated with the possible options is appropriate. In others, the range of choice is far wider, and perhaps we need to provide some option for searching for a particular object.

Unfortunately this is easier said than done in a web context, and can take a lot of effort to work across multiple browsers. We therefore wrapped the functionality in two controls that allows a "popup" window to be displayed – to select an object, such as a User – and for it to automatically return the identifier for this object to the original page – along with some sort of meaningful identifier (such as the UserName) to update the page. For example, when editing content, you would see the following option:

Figure 15 - Relation Picker (Demo 1)

Clicking 'Choose', results in a popup window from which to choose an appropriate user:

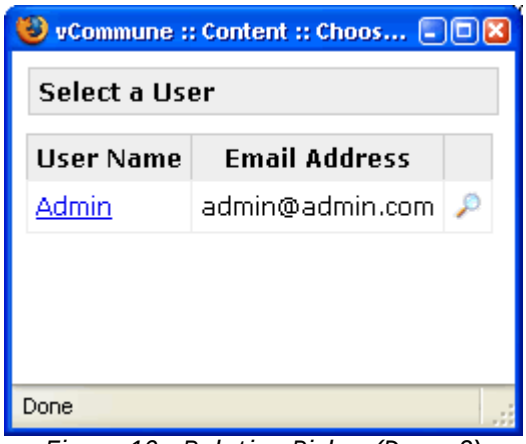


Figure 16 - Relation Picker (Demo 2)

Clicking the appropriate hyperlink closes the popup window, and the original window has been automatically updated to read:

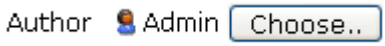
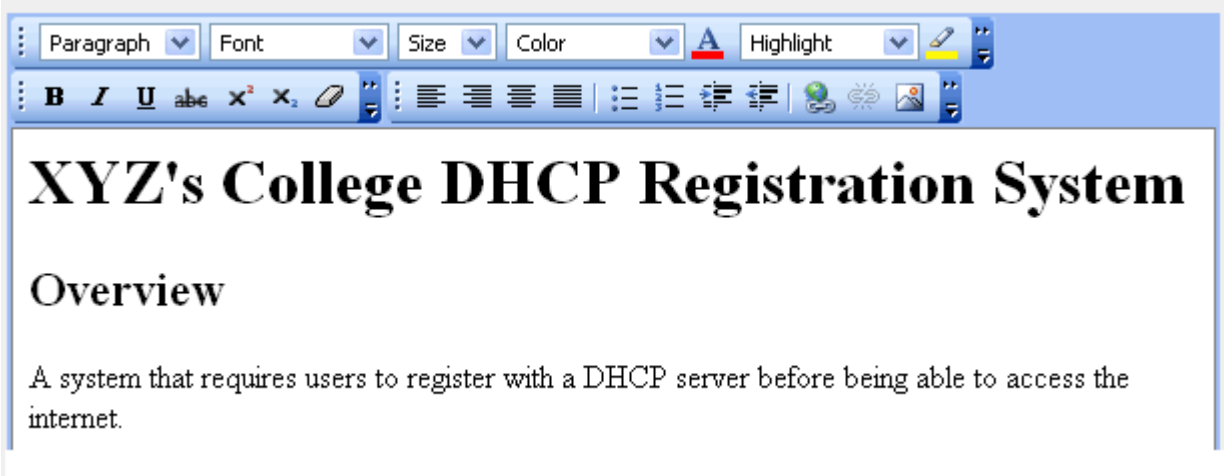


Figure 17 - Relation Picker (Demo 3)

An underlying hidden field is automatically updated to store the actual UserID value that will be returned to populate our Content business entity.

5.7.10 WYSIWYG Editing

In order to provide the WYSIWYG editing, we simply plugged in a free ASP.NET control whenever a field contained XHTML rather than plain text.



5.7.11 URL Rewriting

In order to map "friendly" URL's to real ones, we need to intercept requests for particular pages, and then "rewrite" the URL before the ASP.NET runtime attempts to serve the page. In order to do this in .NET, we write a custom HttpModule[1] which can be "plugged in" to the system – with no changes elsewhere. Our implementation stores an XML file of URL mappings using a combination of regular expressions and custom code to both match and replace the URLs, adding any necessary parameters to the query strings of the pages. For example,

- `/directory/visualbasic/tutorials/` mapped to `/directory.aspx?id=#IdOfTutorials#`
- `/article/beginningasp/` mapped to `/show.aspx?type=article&alias=beginningasp`

5.7.12 Content Colourization

For developer-orientated websites, many of the content pages being stored in the system are likely to contain programming code. It would be very beneficial for visitors if this code could be "colourized" according to the appropriate language. If this code is embedded within an XHTML field, then this should already be tagged using the HTML <code> tag. We can extend this tag to include an attribute "language" which specifies a programming language.

```
<code lang="csharp">void doSomething();</code>
```

To do this we add a total of 3 lines of code to our BLL so that it inserts the necessary HTML tags (using an existing component) in order to colourize the code before passing it on to the presentation layer that requested it.

This sample code is an IHttpHandler implementation that reads a JPG from the filesystem and dynamically generates a thumbnail sized version of the image and emits that to the response stream. What I like about this approach is that you don't need to create a file on the filesystem for the thumbnail

```
public class ImageHandler : IHttpHandler
{
    // the max size of the Thumbnail
    const int MaxDim = 120;

    public void ProcessRequest(HttpContext ctx)
    {
        // let's cache this for 1 day
        ctx.Response.ContentType = "image/jpeg";
        ctx.Response.Cache.SetCacheability(HttpCacheability.Public);
        ctx.Response.Cache.SetExpires(DateTime.Now.AddDays(1));

        // find the directory where we're storing the images
        string imageDir = ConfigurationSettings.AppSettings["imageDir"];
```

Figure 18 - Automatic code colourization in action - the underlying document contains no formatting information whatsoever, apart from a language identifier.

5.7.13 Web Services & Client Interface

A restricted set of methods from the BLL were exposed for public (or authenticated) remote use through "web services"[2]. We then wrote a simple client application to demonstrate some of this basic functionality, as shown below. All of the information on the screen is "live" and fetched through the public BLL on the server.

[1] See Figure 4 - The modified ASP.NET pipeline for our template processor, and the URL rewriter
[2] See Appendix A – The .NET Framework (Web Services)

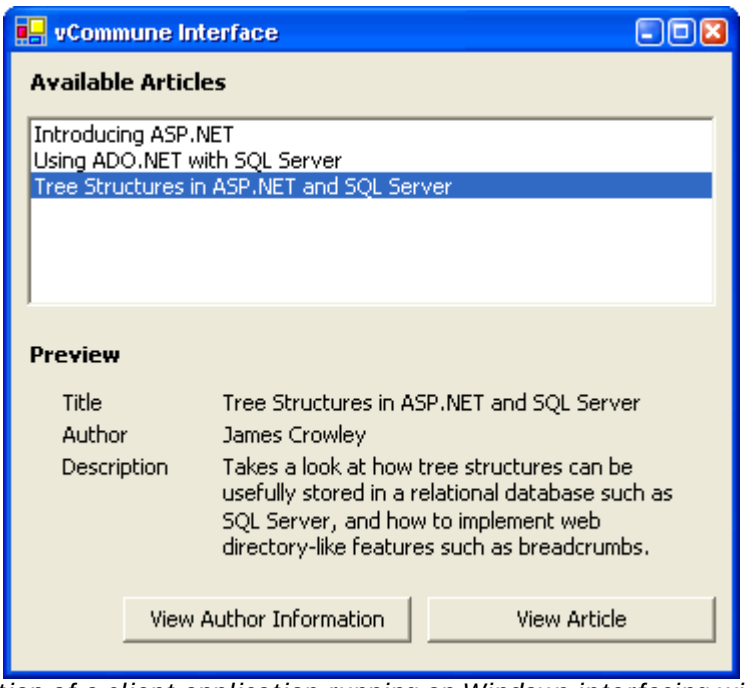


Figure 19 – A demonstration of a client application running on Windows interfacing with the CMS system's BLL through web services.

Although this client is also written in .NET and running on Windows, it could just as easily have been a Java program running on Linux, interfacing with the same set of public web services.

6 Testing

In an ideal world, we would be able to design an entire system on paper – hand it over to a team of developers, and then have a flawless working solution. Sadly this isn't the case, and problems can be introduced – whether it's because the code being written simply didn't match the specification, or the existing frameworks being used have "undocumented" behaviour, at least some testing needs to take place in order to establish everything is working as expected. In addition, we also need to ensure that everything is scaling as expected.

6.1 Custom Controls

For example, one of the things we discovered was that we could gain a significant performance boost (far greater than expected), by creating an entirely custom control in order to take advantage of our built-in Get/SetProperty methods, (instead of using the built-in "repeater" control) in the many sections of our anonymous front end that consisted of lists bound to our business objects.

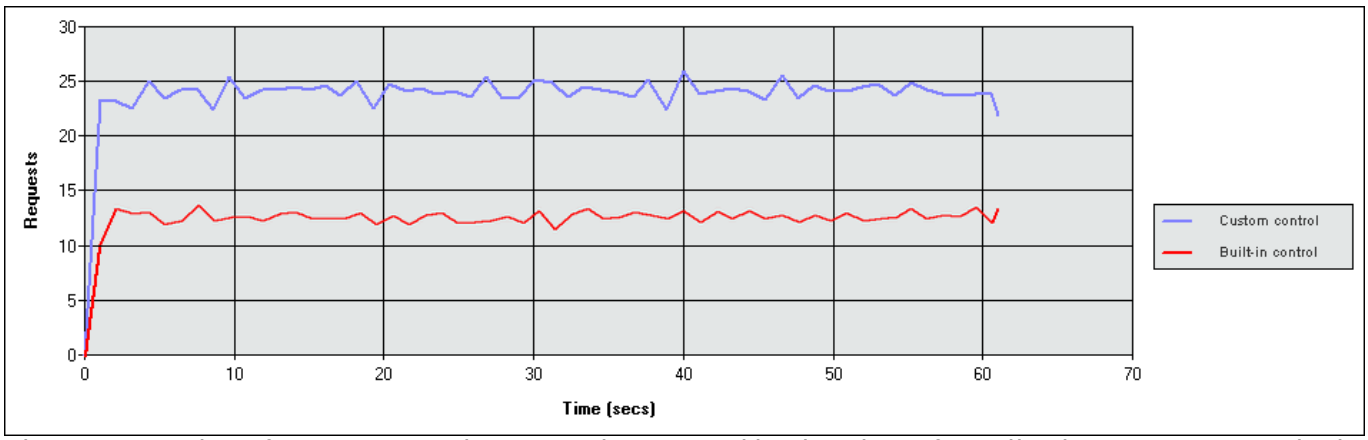


Figure 20 – Number of requests served per second, generated by the Microsoft Application Center Test tool. The test was performed displaying a list with 1000 items using the built-in ASP.NET control (using reflection), compared to using our custom control (using the compile-time generated Get/SetProperty methods)

6.2 Web Interface

In an ideal world, implementing a web interface - according to standards published by a recognised body - should make it straightforward to provide a consistent UI and functionality across browsers and platforms. Unfortunately, this can be notoriously difficult. Every browser interprets the "standards" in different ways, and with differing levels of support. Pages can be rendered differently, and even the same browsers on different platforms (notably IE for Windows and Mac) have this problem. In the end you have to work very hard to find some common ground on which to work, and try to ensure that at the very least, if a particular browser doesn't support certain features, then this will fail gracefully rather than preventing the page loading entirely.

6.3 Scalability

In order to check that the system remained usable with large quantities of data, we imported data from a real-world site consisting of over 1000 pieces of content, hundreds of relations between content objects, and over 25,000 users. Fortunately the filtering, searching and association methods that we implemented meant that despite this large quantity of data, we were still able to manage the content with ease.

7 Conclusion

This project had a fairly broad reach in terms of topics – covering issues from object oriented programming & design, security, databases, and algorithms (when dealing with hierarchical structures). It was the first time I'd ever undertaken a software project of this size and the approach to solve a problem like this contrasts significantly to that of our practicals – and made me fully appreciate the benefits of simply specifying requirements, enabling you to then ensure that as you progress through the project the "solution" is indeed satisfying your original needs. Keeping an entire plan for the project in your memory doesn't tend to scale up to large projects – and so having a logical design allows you to have some comprehension as to how everything fits together before the individual components have been written (which for that matter would probably be written by different people in a team).

I took about 170 hours on this project, of which 20% of the time went into the design, 50% into the coding and refinement, 10% into testing, and 20% into the report. The large proportion of time spent coding was predominantly due to problems encountered with the development tools I was using – in fact, the version of Microsoft's .NET Framework that I used was only in its early beta stages[1] when the project started. I'd adopted this technology as it introduced some significant new language features – such as generics[2], anonymous and partial classes – and improvements in their ASP.NET scripting model that allowed for some far more elegant solutions than would have otherwise been possible. Obviously using beta software introduced its own set of problems, with unexpected "features" and limited documentation significantly increasing the time spent coding. On the positive side, it also allowed me to impact Microsoft's product cycle by reporting bugs I discovered, and suggesting changes to the framework itself[3] (and more importantly, having some of these changes taken on board!).

One of the main challenges of this project was establishing a workable architecture – whilst satisfying our extensibility requirements (from plugging in different tiered layers, to data providers and numerous content types). Sticking rigidly to the 3-tier model resulted in some trade-offs. ASP.NET is great for "plug and play" style coding – getting something that works with the minimum amount of development time possible, but unfortunately these savings are lost the moment you try and abstract away details from each layer in the model.

The actual planning of the database structure for this system was a fairly minor part of this project - and straightforward given how well documented the process is for doing this sort of thing. The one exception was our requirement for storing the trees, and dynamically loading different content objects from a database query. There were certainly no hard and fast rules for this, and although there were various suggestions, there were little specifics – especially for SQL server – and it took a little time to figure out the best approach to the problem, and the queries that were required to maintain these.

---

[1] In fact, .NET 2.0 was originally scheduled by Microsoft to ship in the second half of 2004 – and is now scheduled for the second half of 2005!

[2] Unlike Java, the Generics implementation in .NET is not simply type checking at compile time, but fully supported by the runtime – and therefore we gain performance benefits of avoiding unnecessary type casting and the associated checks.

[3] You can find the bugs and suggestions I reported to Microsoft in the process of this project at http://tinyurl.com/5fbcg

A lot of time and effort also went into getting the templating system working satisfactorily. However, its use in this project made me realise just how beneficial being able to automatically generate portions of the code can be in a "real world" project. Once you've got a piece of code that meets the specification for one object, this can then be templated and duplicated to implement similar functionality for all other objects - and there is immediately less chance of problems creeping in. If you need to refactor your code, you modify a few templates, and with a click of a button, tens or hundreds of files are re-generated – to create the database with the new structure and potentially all 3-tiers of .NET code to interface with it.

In the end though, what was produced is a polished product, with all the key features required from a CMS, and the flexibility to easily add new functionality as required. The system is now up and running on a dedicated server[1], eventually to be rolled out across a number of large websites, including my own, and I'm confident it will be fully capable of dealing with the demands of millions of page views a month, whilst representing a great improvement on the existing systems reducing the management problems often associated with running such large websites.

[1] You can see it in action at http://beta.developerfusion.co.uk/ (Username: Admin, Password: public)

8 Glossary

- **.NET -** A framework provided by Microsoft for developing software. See Appendix A – The .NET Framework.
- **ASP.NET -** A subset of the .NET framework for developing web-based front-ends. See Appendix B – ASP.NET.
- **BLL -** Business Logic Layer, part of the n-tier design. See 5.3 Business Logic Layer
- **CAPTCHA -** Completely Automated Public Turing Test to tell Computers and Humans Apart.
- **CMS -** Content Management System
- **DAL -** Data Access Layer, part of the n-tier design. See 5.2 Data Access Layer
- **HTML -** HyperText Markup Language (HTML) is a markup language designed for the creation of web pages and other information viewable in a browser. This isn't so much a standard, as an evolution of what was seen to be useful at the time, and it was up to browsers to determine the "intent" of the page designer. Since then, increasingly stricter standards are being introduced – the latest of which is XHTML which enforces the very strict checking from the world of XML.
- **HTTP -** HyperText Transfer Protocol, the primary method used to convey information on the World Wide Web. The original purpose was to provide a way to publish and receive HTML pages.
- **J2EE -** Java 2 Platform, Enterprise Edition is a system provided by Sun Microsystems for developing distributed multi-tier architecture applications, based on modular components running on an application server.
- **MAC -** Message authentication code, which essentially generates a hash of the data so that its integrity can be verified.
- **MD5 -** Message-Digest algorithm 5, a cryptographic hash function with a 128-bit hash value.
- **Metadata -** "data about data".
- **OOP** – Object oriented programming.
- **RSS -** Really Simple Syndication, an XML format for web syndication used by news websites and weblogs to allow users to be notified of new content on a site.
- **SQL -** Structured Query Language, used for performing queries over relational databases. See Appendix C – Relational Databases and SQL Server (Structured Query Language (SQL))
- **SQL Server -** A relational database written by Microsoft.
- **WYSIWYG -** "what you see is what you get", referring to the correspondence between what you see on a computer screen, and what you see when the document is printed out. This is expected of most word processors these days – but not necessarily when editing HTML or XHTML documents.
- **XHTML –** eXtensible HyperText Markup Language, an extension of XML designed to replace HTML – the non-standard that browsers use to render web pages.
- **XML -** Extensible Markup Language, a standardized general-purpose markup language. Its primary purpose is to facilitate the sharing of structured text and information across the Internet.
- **XSL -** eXtensible Stylesheet Language, an XML language for transforming XML documents from one syntax to another.

9 Appendix A – The .NET Framework

The .NET framework consists of

- a common environment for building, deploying, and running both web and client applications.
- a common language runtime and common class libraries – such as ADO .NET (accessing databases) ASP .NET (server-side web pages) and Windows Forms (client-side applications) - to provide advanced standard services that can be integrated into a variety of computer systems.
- The .NET Framework is language neutral. Currently it supports over 45 different languages, including C++, C#, Visual Basic, and J#[1]

The framework is too extensive to discuss in detail here, but we'll examine a few of the features that we take advantage of in this content management system.

9.1 Generics

Generics is essentially the ability to have type parameters on your type, and were covered in the OOP course. With generics, you can define a standard List class that accepts a type parameter T (generally denoted as List<T>) which specifies the type the list will be storing. Now in your class, instead of returning type Object when you request an item in the list, you are given a strongly typed object of type T. Assuming the underlying runtime supports these generic types[2] (rather than just the compiler), then the runtime no longer needs to perform the type checking that would have taken place when casting to the appropriate type.

9.2 Partial Classes

Partial classes allow the definition of a class to be spread across multiple files. In order to do this, the class is simply marked using the "partial" keyword, and at compile time these files are automatically combined. The great benefit of this is that you can automatically generate a portion of a class in an entirely separate file without having to worry about overwriting custom code that has been written.

9.3 XML Serialization

The .NET framework supports a feature known as XML serialization which allows us to turn an instance of a .NET class into an XML file – and load it back again. More precisely, it persists the value of any properties or public fields that .NET knows how to "serialize" to XML. These include the basic data types (String, Int32 etc), arrays of these and any classes which consist only of such properties. Custom serialization can also be implemented to more precisely control how an object is serialized – and we use this feature when dealing with our XML object definitions for the provider specific information, so that we can serialize each XML attribute (unknown at design time) into an array of values.

9.4 Reflection

Whenever a .NET assembly is compiled, "metadata" describing the methods and classes present in the assembly is also stored. Reflection is the ability to read this metadata at runtime, discover the properties and methods of a type, and dynamically invoke them. This means that instead of hard coding accesses to properties, we can for instance load a "property name" from a configuration file, and set it to a specific value dynamically. However, performing an operation like this is obviously far more expensive then using a precompiled property set statement.

9.5 Web Services

A web service is a collection of protocols and standards used for exchanging data between applications that are potentially written in different languages and platforms. This allows us to expose certain aspects of functionality from our application across the internet.

---

[1] See http://www.dotnetpowered.com/languages.aspx for a full list of languages that have been implemented in the .NET framework

[2] Java is an example where this is not the case – although the compiler checks that your code is dealing with objects of the correct type, it then replaces all the generic types with the type Object – and so the runtime still has to perform unnecessary checks as to whether the object being casted is actually of that type. Version 2.0 of the .NET framework on the other hand supports generics at the runtime level – and so these type checks can be avoided, improving performance.

10 Appendix B – ASP.NET

ASP.NET is another component of the .NET Framework, responsible for delivering web-based presentation layers. An ASP.NET page consists of two files: a "visual" file that declares controls on the page, and a "code-beside" file – written in whichever .NET language you desire - that defines the behaviour of the page.

ASP.NET 2.0 takes advantage of partial classes[1] allowing it to automatically generate .NET code that results in the output defined in our "visual" file. This is then combined with our "code-beside" file into a single class. The benefit of this is that both files can refer to members in each, whilst retaining separation between layout and code logic.

The visual file can contain a combination of standard HTML tags, and "server controls" that are marked with a "runat=server" attribute. These server controls are actually classes within ASP.NET and may be instantiated, have methods run on them and properties changed just like any standard class. They are very similar to the standard controls you would use to create a form as a client application – except that they render to HTML rather than to pixels on a graphical display.

The code below demonstrates a basic ASP.NET page with its code-beside file, and a custom server control.

HelloWorld.aspx

```
<%@ Page Language="C#" CompileWith="HelloWorld.aspx.cs"
         ClassName="HelloWorld_aspx" Inherits="System.Web.UI.Page" %>
<%@ Register Namespace="vCommune.Framework.Controls" TagPrefix="vc" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
          "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Demonstration ASP.NET Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <p><asp:TextBox ID="txtName" Runat="server" Text="James Crowley" />
            <asp:Button ID="cmdButton" Runat="server"
                        Text="Click Me!" OnClick="cmdButton_Click" /></p>
        <p><vc:Demo ID="resultText" Visible=false runat="server" /></p>
    </div>
    </form>
</body>
</html>
```

This is a very basic layout page, consisting of some standard HTML tags, and some server controls that generate a TextBox, a Button, and a custom server control (defined shortly). At the top of the page we have a special "Page" tag at the top which gives ASP.NET information such as what "code-beside" file to compile the page with, the name to give the class, and what class it inherits from.

HelloWorld.aspx.cs

```
using System;
using System.Web;

public partial class HelloWorld_aspx
{
    void cmdButton_Click(object sender, EventArgs e)
    {
        resultText.Name = txtName.Text;
        resultText.Visible = true;
    }

}
```

In this instance, the "code-beside" file only has one small bit of code – which will be automatically called whenever the Button defined in our layout page is clicked. In this file we could also choose to override members from the inherited

[1] See Appendix A – The .NET Framework (Partial Classes)

class (defined in the layout page), introduce private or public variables, and anything else you'd expect from an object oriented language.

Demo.cs

```
using System;
using System.Web.UI;

namespace vCommune.Framework.Controls
{
    public class Demo : System.Web.UI.Control
    {
        string _name = "";

        public Demo() {}

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        protected override void Render(HtmlTextWriter writer)
        {
            writer.WriteFullBeginTag("b");
            writer.Write("Hello " + _name + "!");
            writer.WriteEndTag("b");
        }
    }
}
```

Finally, this is the definition of our custom server control. These do not have a layout defined, and are instead responsible for generating the HTML directly – however, ASP.NET includes utilities for loading "templates" from external files.

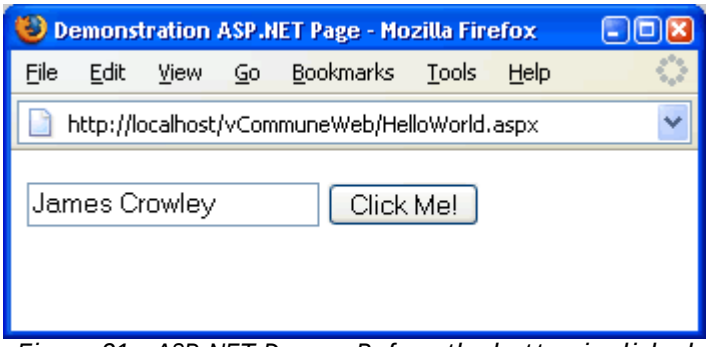


Figure 21 - ASP.NET Demo - Before the button is clicked

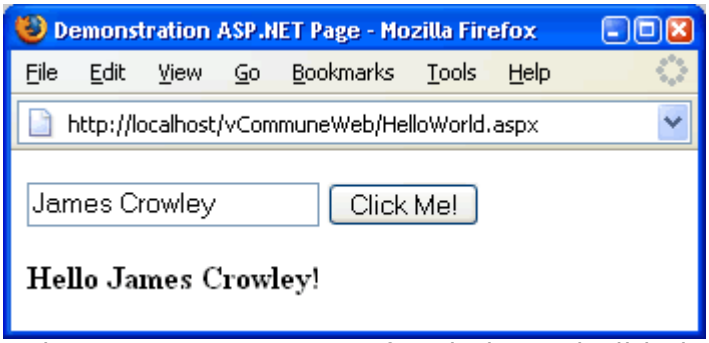


Figure 22 - ASP.NET Demo - After the button is clicked

11 Appendix C – Relational Databases and SQL Server

A relational database stores information within tables. Each table consists of a set of columns – which are normally defined with information such as a name, and a data type – and a set of rows. Each row contains a value for each of the defined columns – though some columns (depending on their definition) may allow NULL (or unknown) values. We can also define relations between columns in different tables – for instance, one table of articles may have an "`AuthorID`" column, which we could define as being related to a column "`UserID`" in a table of users.

11.1 Structured Query Language (SQL)

Queries can be performed over these tables using a standard language known as the Structured Query Language (SQL). This language is used not only to fetch data from one or more tables, but also to update, add and remove rows from the tables. Unfortunately, as with most "standards" most vendors have implemented their own extensions to the language – so Microsoft, Oracle, Postgres etc all have slightly different twists. The basics of this language were covered in the B1 Databases course.

11.2 Stored Procedures

Stored procedures are a feature in most relational databases – including SQL Server, Oracle and more recently MySQL – that allow us to predefine queries and store them in the database. Like any other sort of procedure, these can accept parameters, and generally return one or more result sets (and possible a return value) However, because of the limited set of permutations, the database can prepare the execution plan beforehand.

The following stored procedure would fetch any rows from the table "`vcUser`" where its `UniqueID` column matched the value we were passed in the `@UniqueID` parameter.

```
CREATE PROCEDURE vcGetUser ( @UniqueID INT ) AS

SELECT * FROM vcUser WHERE UniqueID=@UniqueID

RETURN
```

We can then externally call "`vcGetUser 1`" which will perform the query and return any rows with the `UniqueID` of 1.

11.3 SQL Injection Attacks

A common method of attacking web-based applications with a back-end database is a technique known as SQL "injection". This involves submitting specially crafted input from a web page – such as a login form – which ends up being executed on the database. This essentially occurs due to un-validated user input being placed directly into an SQL query – for example, suppose our code generated an SQL query for a login attempt as follows

```
mySqlQuery = "SELECT id FROM users WHERE username='" + userInput["username"]
                                  + "' AND password='" + userInput["password"]
```

that was then executed on the database, and if a row returned, then it was assumed that the username and password combination was valid. However, suppose the user input for "username" was in fact

```
' OR 1=1 --
```

In this case, the final SQL query would look something like this

```
SELECT id FROM users WHERE username='' OR 1=1 --' AND password='password'
```

This would end up returning all rows in the users table. Furthermore, if the code then assumed the user was the first "id" in the resultset (because normally there would be only one returned), the chances are the first user in the database would be the administrator account!

However, by using stored procedures, and forcing our code to pass parameters of a particular type, we can prevent these sorts of attacks – as both the data provider connecting to the database, and the stored procedure itself will treat

the parameters purely as input, and not SQL. See Appendix F – References (Using Databases in .NET) for more information.

11.4 Triggers

Triggers are essentially a special case of stored procedures that are executed automatically when particular events in the database occur – notably when a row in a table is inserted, updated or removed. When executing these stored procedures, the database gives the stored procedure access to additional information such as the old and new versions of the rows that have been modified.

11.5 Many to Many Relationships

We saw earlier that the specification suggested a number of one-to-many relationships between particular objects, which were defined in our XML object definitions. There are also a number of many-to-many relationships in our specification, which we represent using

- Each content object is associated with one or more categories
- Each user is associated with one or more roles

For these, we create an additional table that maps content UniqueID's to category UniqueID's. This table therefore contains two foreign keys. In order to fetch all the categories to which a particular content object is associated, we simply perform a JOIN query between these tables.

These intermediary tables are not reflected directly in the business entities (and hence no definitions in the XML files) – instead, our .NET provider classes provide some additional methods (in addition to the standard Get/Update etc) that allow us to fetch objects according to these relationships.

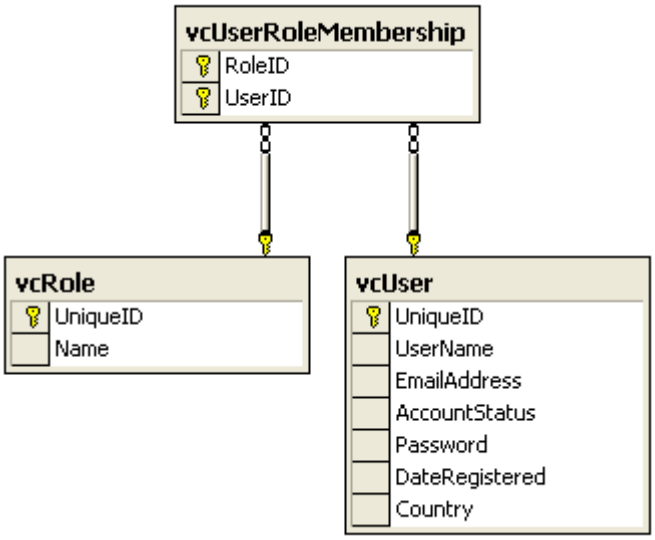


Figure 23 - An intermediary table being used to implement a many-to-many relationship

11.6 Storing Trees

There are a number of possibilities for storing trees within a relational database.

- An adjacency list model, which stores a list of nodes, each with a unique identifier, and an associated parent id. This is very straightforward, but to perform queries in SQL (such as finding the path through the tree to a particular node) requires extensive use of recursion.
- The adjacency list model can also be extended so that we store additional data (generally a "depth" and "lineage" field) representing the depth of the node, and the path from the root to that node – automatically maintained using SQL triggers. This allows meaningful queries to be performed using a single SQL query using pattern matching. However, the size of the lineage column for each node grows in length with the depth of the tree, and it can still be difficult to perform aggregation operations over the tree.
- The final option is to ignore the adjacency model entirely, and use a modified pre-order traversal, which associates two integers "left" and "right" values with each node. Although updates to the structure of the tree

are relatively expensive (potentially updating all left and right node values to add a single node), in our scenario this is going to be fairly uncommon when compared with the frequency of performing selects over the structure.

Using the last model, we can perform some of the following queries.

11.6.1 Adding a new node to the left

```
ALTER PROCEDURE vcAddTreeNode(@Name VARCHAR(20),@ParentNode INT,@AddRight BIT) AS

DECLARE @nodeValue INTEGER;

-- get the current right hand node
IF @AddRight=1 BEGIN
	IF @ParentNode=0
		-- get the maximum rgt value (or 1 if NULL)
		SELECT @nodeValue=ISNULL(MAX(rgt) + 1,1) FROM vcCategory
	ELSE
		-- just get the rgt value of the node we're adding to
		SELECT @nodeValue=rgt FROM vcCategory WHERE UniqueID = @ParentNode
END ELSE BEGIN
	IF @ParentNode=0
		-- start with a lft value of 1
		SET @nodeValue=1
	ELSE
		-- get the value of lft we'll be using
		SELECT @nodeValue=lft+1 FROM vcCategory WHERE UniqueID = @ParentNode
END

-- make space for this new node

UPDATE vcCategory
	-- inserting a new node affects all nodes that
	-- have a "right" value (if adding to right,
	-- or "left" + 1 value for left) greater or equal to
	-- the position that we're inserting at

	-- our new node uses two new values
	-- so for each of these nodes
	-- their left and right values get increased
	-- by two - provided they're actually
	-- greater than the position we're inserting
	-- (we don't want to change the left hand value
	-- of the parent nodes)
	SET lft = CASE WHEN lft >= @nodeValue
			THEN lft + 2
			ELSE lft
		END,
	    rgt = CASE WHEN rgt >= @nodeValue
			THEN rgt + 2
			ELSE rgt
		END
WHERE rgt >= @nodeValue

-- insert the actual new node
INSERT INTO vcCategory (Name, lft, rgt)
	VALUES (@Name, @nodeValue, (@nodeValue + 1))

RETURN
```

11.6.2 Deleting a node

```
CREATE PROCEDURE vcDeleteTreeNode(@UniqueID INT) AS

DECLARE @leftValue INTEGER
DECLARE @rightValue INTEGER
-- get the left/right values of the node we're deleting
SELECT @leftValue=lft,@rightValue=rgt FROM vcCategory WHERE UniqueID=@UniqueID
-- remove the node, and all it's children
DELETE FROM vcCategory WHERE lft >= @leftValue AND rgt <= @rightValue
```

```
-- update the tree so that the lft/rgt values are adjusted
-- to account for the loss of (right-left+1) values
-- we keep these accurate so we can still calculate the
-- depth and number of children of a node
-- if this wasn't important, we wouldn't have to do
-- this step
UPDATE vcCategory
       SET lft = CASE WHEN lft >= @rightValue
                      THEN lft - (@rightValue - @leftValue + 1)
                      ELSE lft
                 END,
           rgt = CASE WHEN rgt >= @rightValue
                      THEN rgt - (@rightValue - @leftValue + 1)
                      ELSE rgt
                 END

WHERE rgt >= @rightValue

RETURN
```

11.6.3 Getting path through tree to @NodeID

```
SELECT P1.Name
   FROM vcTree AS P1, vcTree AS P2
  WHERE P1.lft <= P2.lft AND P1.rgt >= P2.rgt
        AND P2.UniqueID=@NodeID
  ORDER BY P1.lft
```

11.6.4 Number of children

```
SELECT P1.Name, ((P1.rgt-P1.lft-1)/2) AS children
   FROM vcTree AS P1
  ORDER BY P1.lft;
```

11.6.5 Node depths

```
SELECT COUNT(c2.Name) AS depth, c1.Name
   FROM vcCategory AS c1, vcCategory AS c2
  WHERE c1.lft BETWEEN c2.lft AND c2.rgt
  GROUP BY c1.Name,c1.lft
  ORDER BY c1.lft;
```

11.7 Storing Passwords

Storing passwords as plaintext in a database record generally isn't a good idea from a security point of view. A common technique instead is to use a one-way hash algorithm to store the password. In our case, we use MD5 which takes a string and returns a 128-bit hash. However, there is in fact one further security risk, despite using the hashing algorithm, and this is what is known as a "dictionary" attack. Supposing an attacker gains access to the database, although they cannot get back to the original password, the chances are that many user passwords are very common. Therefore, the hacker could create a dictionary of common passwords, compute their hash, and check if any of the accounts have a matching password field. If they match – then he has the original password from the dictionary. Here the power of our database system works against us, as the hacker could simultaneously check whether *any* user passwords stored in the table match a particular entry with a single SQL query.

Fortunately, we can make this more difficult by adding "salt" to the hash. The idea behind this is to add a second piece of information to the hash that is non-changing and unique for every user – such as a username. The consequence of this is that now even user accounts with the same password has a different password hash. Even if the hacker knows how we "salt" the hashes, he can only attack one password at a time – with a different SQL query for each user and each password.

11.8 Class Hierarchies

There are basically two approaches for storing class hierarchies in SQL.

- Create a separate table for each class, and simply duplicate the shared field names. This is generally adopted if the base class doesn't make sense on it's own
- Create a table with all the common fields, and then a separate table for each class's additional fields – and then join these together as needed.

Both of these approaches are straightforward – and we created the stored procedure such that it would automatically fetch the fields from the appropriate view (generated using the templating system). Our main problem arose with deciding how to deal with this "dynamic" data in the DAL in order to restore the hierarchy of objects inheriting from the correct set of business entities.

11.9 Passing Lists to Stored Procedures

Stored procedures don't support receiving arrays as parameters – which means if for example we wanted to record all categories that a user has selected to be associated with some content, we'd have to call the procedure for each category that needed to be associated with it. Obviously, this is far from ideal and causes an unnecessary number of database round trips.

One thing stored procedures do allow to be passed, however (at least in SQL Server) is an in-memory table – but we can't pass one of these from our DAL. Therefore, we created a function (in T-SQL) that converts a comma delimited list into a table with each entry in a new row.

```
CREATE FUNCTION ConvertStringListToTable(@stringList VARCHAR(200))
RETURNS @StringListTab TABLE ( Selection INT ) AS
BEGIN
    DECLARE @@i INT
    DECLARE @@length INT

    SET @@i = CHARINDEX (',', @stringList, 0)
    SET @@length = CHARINDEX (',', @stringList, 1) - @@i

    --Makes sure that there is still data
    WHILE CHARINDEX (',',@stringList,@@i+1) > 0
    BEGIN
        SET @@length = CHARINDEX (',', @stringList, @@i+1) -
                       CHARINDEX (',', @stringList, @@i)
        IF (SUBSTRING(@stringList, @@i+1, @@length-1) <> '')
        BEGIN
            INSERT @StringListTab
                SELECT SUBSTRING(@stringList, @@i+1 , @@length-1)
        END
        SET @@i = CHARINDEX (',', @stringList, @@i+1)
    END
RETURN
END
```

Now, given a parameter in a stored procedure that contains a list of these values (@categoryIDs), we can then use this function in an insertion SQL statement, with a SELECT subclause:

```
INSERT INTO vcContentAssoc(ContentID, CategoryID)
    SELECT
        @ContentID,
        Selection
    FROM
        ConvertStringListToTable(@categoryIDs)
```

12 Appendix D – Business Entities

Column Name	Data Type (Size)	Description
User, represents an authenticated user		
UniqueID	Integer	A unique identifier
UserName	String (20)	A unique user name
EmailAddress	String (50)	A unique email address
AccountStatusID	Integer (AccountStatus)	The status of the account
Password	String	The password for the account. Stored as 128-bit

		hash in the database
Role, a role that a User belongs to		
UniqueID	Integer	A unique identifier
Name	String (20)	A name for the role
Description	String (200)	A description for the role
WorkflowStage, a stage within the content workflow		
UniqueID	Integer	A unique identifier
Name	String	A name for this state
Active	Boolean	Indicates whether Content is “published” at this stage in the workflow.
WorkflowTransition, a valid transition from one WorkflowStage to another		
UniqueID	Integer	A unique identifier
TransitionFromID	Integer (WorkflowStage)	The stage that this transition goes from
TransitionToID	Integer (WorkflowStage)	The stage that this transition goes to
RoleID	Role	The role that is permitted to perform this transition.
WorkflowTransitionLog, a record of a transition occurring.		
UniqueID	Integer	A unique identifier
OldStateID	WorkflowStage	The stage that we moved from
NewStateID	WorkflowStage	The stage that we moved to
UserID	Integer (User)	The user that performed the transition
ContentID	Integer (Content)	The content that this relates to
TransitionDate	DateTime	The date and time at which the transition occurred.
Content, the basic content object.		
UniqueID	Integer	A unique identifier
Title	String	The title of the content
Description	String (300)	A description of the content
AuthorID	Integer (User)	The author of the Content
TypeID	Integer (ContentType)	The type of the Content
WorkflowStageID	Integer (WorkflowStage)	The position in the workflow the content is at
LastModified	DateTime	The date the content was last modified.
DateCreated	DateTime	The date the content was created.
ContentBook, a specific type of content, extending from Content (just as an example)		
ISBN	String (10)	The ISBN of the book
Publisher	Integer (User)	The publisher of the book.
ContentRelation, relating one piece of content to another.		
UniqueID	Integer	A unique identifier
ContentID	Integer (Content)	The Content this relation refers to
RelatedContentID	Integer (Content)	The related Content
RelatedContentURL	String	The related URL
RelatedContentName	String	The related name of either the URL or the Content (nb: view column)
ContentResource, a binary resource for a piece of content.		
UniqueID	Integer	A unique identifier
Name	String	The name of the resource
Public	Boolean	Indicates whether or not this should be listed for public download.
Category, a node in the hierarchical tree		
UniqueID	Integer	A unique identifier
Name	String (50)	A label for the node
Depth	Integer	A value calculated from the underlying lft/rgt values in the structure of the tree.
ChildrenCount	Integer	A value calculated from the underlying lft/rgt values in the structure of the tree.

13 Appendix E – Code Snippets

13.1 Templating

13.1.1 Sample Definition

```
<?xml version="1.0" encoding="utf-8"?>
<ObjectConfig>
	<Name>ContentEvent</Name>
	<Inherits>Content</Inherits>
	<Properties>
		<Property Name="Location" TypeCode="String" TypeSize="100" />
		<Property Name="OrganiserID" TypeCode="Int32" >
			<Relation Object="User" />
		</Property>
		<Property Name="EventStart" TypeCode="DateTime">
			<ProviderInfo Name="SqlServer" TypeCode="SmallDateTime" />
		</Property>
		<Property Name="EventEnd" TypeCode="DateTime">
			<ProviderInfo Name="SqlServer" TypeCode="SmallDateTime" />
		</Property>
		<Property Name="Cost" TypeCode="Int32" />
	</Properties>
</ObjectConfig>
```

13.1.2 Sample Template

```
<%@ Page Language="C#" Inherits="vCommune.Configuration.CodeGeneration.ObjectCodeTemplate" %>
<% OutputDestination = String.Format(@"~\vCommuneDALInterface\Generated\I{0}Provider.cs",
ObjectConfig.Name); %>
// This code was generated by vCommune on <%# DateTime.Now.ToLongDateString() %>
using System;
using System.Collections.Generic;
using System.Text;
using vCommune.Framework.Objects;

namespace vCommune.Framework.Providers
{
    // Standard provider implementation
    public partial interface I<%# ObjectConfig.Name %>Provider : IContentProvider
    {
        <asp:Repeater runat="server"
            DataSource=<%# ObjectConfig.DirectProperties.FindAll(HasRelationsDelegate) %>>
        <ItemTemplate>
        List<<%# ObjectConfig.Name %>> GetBy<%# Eval("Name") %>( <%# Eval("TypeCode") +
          " " + ToLowerInitial(Eval("Name")) %>);
        </ItemTemplate>
        </asp:Repeater>
    }
}
```

13.1.3 Sample Output

```
// This code was generated by vCommune on 20 April 2005
using System;
using System.Collections.Generic;
using System.Text;
using vCommune.Framework.Objects;

namespace vCommune.Framework.Providers
{
    // Standard provider implementation
    public partial interface IContentEventProvider : IContentProvider
    {
        List<ContentEvent> GetByOrganiserID(Int32 authorID);
    }
}
```

13.2 Presentation Layer

The examples in this section demonstrate the use of some of the controls that were built for the CMS framework at the presentation layer.

13.2.1 Two-way Data Binding

This is a very basic example of two-way data binding from the presentation layer to the BLL using our extended controls. We simply add all the necessary fields to <EditItemTemplate> (which itself could be automatically generated), and this will form a page on which a user can modify objects (with no additional code).

```
<asp:FormView ID="mainFormView" DataKeyNames="UniqueID" DataSourceID="userSource"
runat="server">
    <EditItemTemplate>
        User Name: <asp:TextBox ID="UserName" runat="server" Text='<%# Bind("UserName")%>' />
        Email Address: <asp:TextBox ID="EmailAddress" runat="server"
                            Text='<%# Bind("EmailAddress")%>' />
    </EditItemTemplate>
</asp:FormView>
<vc:UserDataSource ID="userSource" runat="server" CommandType="Get" />
```

13.2.2 Tabbed GUI Control

Our tabbed GUI control allows us to define a page with some content that appears within the control, and we can then automatically display a selection of (possibly nested) tabs with the appropriate item highlighted according to the URL of the page. The code below demonstrates a possible ASP.NET page, and the corresponding XML configuration file. (For a screen shot, see the main body of the document).

Demo.aspx

```
<%@ Page Language="C#" Title="vCommune :: Tab Demo" CompileWith="Demo.aspx.cs"
ClassName="Default_aspx" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>vCommune :: Tab Demo</title>
    <link href="/vCommuneWeb/tabs.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
        <vc:TabControl TabFileSource="tabs.xml" Runat="server">
            <p>Hello there!</p>
        </vc:TabControl>
    </form>
</body>
</html>
```

Tabs.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfTabItem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <TabItem ImageUrl="~/res/home.gif" CssClass="home" HyperLink="~/Admin/" />
    <TabItem Text="Content" HyperLink="~/Admin/Demo/Default.aspx" />
    <TabItem Text="Search" HyperLink="~/Admin/Demo/Search.aspx" Enabled="false" />
    <TabItem Text="Edit" HyperLink="~/Admin/Demo/Edit.aspx">
            <Tabs>
                <TabItem Text="Main" HyperLink="~/Admin/Demo/Demo.aspx" />
                <TabItem Text="Pages" HyperLink="~/Admin/Demo/EditPages.aspx" />
                <TabItem Text="Categories" HyperLink="~/Admin/Demo/EditCategoryAssoc.aspx" />
                <TabItem Text="Relations" HyperLink="~/Admin/Demo/EditRelations.aspx" />
            </Tabs>
    </TabItem>
</ArrayOfTabItem>
```

13.2.3 Relation Picker

As described in our presentation layer, this allows a relation item to be chosen. Below we demonstrate the code required to get the behaviour described in our presentation layer documentation.

EditContent.aspx

This page displays a button allowing the user to choose the appropriate author.

```
<asp:Label ID="AuthorName" Text=<%# Eval("AuthorName") %> Runat="server" />
<vc:PopupField Runat="server" ID="AuthorID"
    Value='<%# Bind("AuthorID") %>'
    PopupWidth=400 PopupHeight=800
    ScriptSource="~/res/popup.js"
    FeedbackFields="AuthorName"
    ButtonText="Choose..."
    DialogUrl="ChooseAuthor.aspx" />
```

ChooseAuthor.aspx

This page displays the possible options. When an item is clicked, the window is closed and control returned to the parent window, with the HTML element dynamically updated to display the author name (and record the unique identifier in the PopupField control).

```
<vc:PopupController id="popupController" Runat="server" />
<!-- and then for each possible option -->
<a onclick="<%#
popupController.GetClickEvent((string)Eval("UniqueID"),(string)Eval("UserName")) %>">
    <%# Eval("UserName") %></a>
```

13.2.4 Searching

We can create a simple search page using the following.

```
<h2>Search</h2>
<div class="formInput">
    <label>Keyword</label><asp:TextBox ID="TitleFilter" Runat="server" /><br />
    <label></label><asp:Button Runat="server" ID="cmdSearch"
       Text="Perform Search" OnClick="cmdSearch_Click" /><br />
</div>
<h2>Search Results</h2>
    <vc:ExtendedRepeater CssClass="promo" ID="ContentList" Runat="server">
        <ItemTemplate>
            <asp:HyperLink runat="server"
                NavigateUrl='<%# Container.Eval("UniqueID","/show/{0}/") %>'
                Text='<%# Container.Eval("Title") %>' />
        </ItemTemplate>
    </vc:ExtendedRepeater>
```

And in the "code behind" page,

```
protected void cmdSearch_Click(object sender, EventArgs e)
{
    ContentList.DataSource = ContentUtility.GetFilteredContent(TitleFilter.Text, "",
        0, 0, "Title ASC");
}
```

Additional fields to search by can be easily added by simply including additional inputs on the page.

14 Appendix F – References

14.1 .NET Framework

Introducing .NET – Wrox Press
http://www.developerfusion.co.uk/show/1678/
Mono – an alternative implementation of .NET
http://www.mono-project.com/about/index.html

14.2 Using Databases in .NET

Tree structures in ASP.NET and SQL Server – James Crowley
http://www.developerfusion.co.uk/show/4633/
Using ADO.NET with SQL Server - James Crowley
http://www.developerfusion.co.uk/show/4278/
Improving the Security in Encrypting Passwords using MD5 - Scott Mitchell and Thomas Tomiczek
http://aspnet.4guysfromrolla.com/articles/112002-1.aspx
SQL Injection Attacks by Example - Stephen J. Friedl
http://www.developerfusion.co.uk/show/4656/

14.3 Serialization in .NET

XML Serialization in .NET – Anthony Hart
http://www.developerfusion.co.uk/show/3827/
Customize XML Serialization using IXmlSerializable – James Crowley
http://www.developerfusion.co.uk/show/4639/

14.4 Web Services and RSS

What is RSS? – Mark Pilgrim
http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html
Building XML Web Services Using C# and ASP.NET - James Yang
http://www.developerfusion.co.uk/show/3114/

14.5 ASP.NET Controls

Working with the ASP.NET 2.0 ObjectDataSource Control - Stephen Walther
http://msdn.microsoft.com/asp.net/default.aspx?pull=/library/en-us/dnvs05/html/asp2objectdatasource.asp
Creating Custom Web Controls - Venugopal Mallarapu
http://www.codeproject.com/aspnet/VenusCustomWebControls.asp
Introducing ASP.NET 2.0 Master Pages - Thiru Thangarathinam
http://www.devx.com/dotnet/Article/18042
Using CAPTCHA Images to Prevent Automated Form Submission - Mike Hall
http://www.codeproject.com/aspnet/CaptchaImage.asp
Breaking a Visual CAPTCHA - Greg Mori and Jitendra Malik
http://www.cs.berkeley.edu/~mori/gimpy/gimpy.html

14.6 .NET Language Features

An Introduction to C# Generics - Juval Lowy
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/csharp_generics.asp

Hosting the ASP.NET Runtime - Rick Strahl
http://www.west-wind.com/presentations/aspnetruntime/aspnetruntime.asp

14.7 Architecture and Patterns

Provider Design Pattern – Rob Howard
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspnet/html/asp04212004.asp
n-Tier Applications and .NET - Bipin Joshi
http://www.dotnetbips.com/displayarticle.aspx?id=213