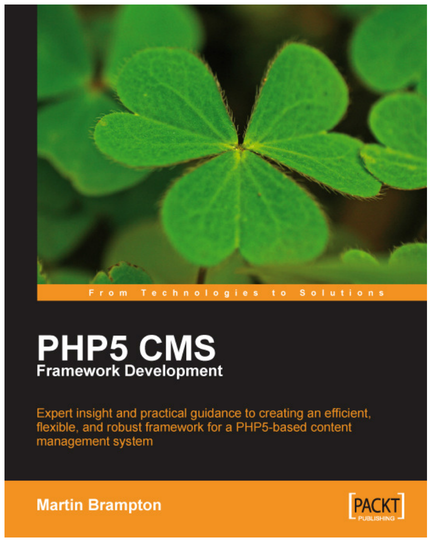
PHP5 CMS Framework Development


From Technologies to Solutions
PHP5 CMS
Framework Development
Expert insight and practical guidance to creating an efficient, flexible, and robust framework for a PHP5-based content management system
Martin Brampton
PACKT PUBLISHING

Martin Brampton

Chapter No. 6
"Access Control"

About the Author

Martin Brampton, an internationally known IT Industry Analyst, has an unrivalled grasp of the complexities of modern day system architectures built on both research and practical experiences. Martin's knowledge of the importance of scalable frameworks is founded on the early days of his career. After studying mathematics at Cambridge University, he built major software systems in both financial and technical application areas. Several of his systems were acclaimed as "legendary" in their reliability—some of which are still in use today.

After a decade of heading IT for an accountancy firm, a few years as a director of a leading analyst firm, and an MA degree in Modern European Philosophy, Martin finally returned to his interest in software, but this time transformed into web applications. He found PHP5, which fits well with his prejudice in favor of programming languages that are interpreted and strongly object oriented.

Utilizing PHP, Martin took on development of useful extensions for the Mambo (and now also Joomla!) systems, then became a team leader for developing Mambo itself. More recently, he has written a complete new generation CMS named Aliro, many aspects of which are described in this book. He has also created a common API to enable extensions to be written with a single code base for Aliro, Joomla (1.0 and 1.5) and Mambo (http://www.acmsapi.org).

All in all, Martin is now interested in too many things and consequently has little spare time. But his focus is on object oriented software with a web slant, much of which is open-source. He runs Black Sheep Research, which provides software, speaking and writing services, including "The Brampton Factor", a monthly column for silicon com (http://silicon.com/comment/martinbrampton) where he is politely described as a veteran analyst.

PHP5 CMS Framework Development

This book guides you through the design and implementation decisions necessary to create a working architecture for a PHP5-based content management system. Each of the major areas and decision points are reviewed and discussed. Code examples, which take advantage of PHP5's object oriented nature, are provided and explained. They serve as a means of illustrating the detailed development issues created by a CMS. In areas where the code is too voluminous to be reproduced in detail, the design principles are explained along with some critical pieces of code. A basic knowledge of PHP is assumed.

All of the code samples are taken from a frozen version of the Aliro development project, and you can visit a site running on that version at http://packt.aliro.org. Apart from being a demonstration of the code in action, the site provides access to the whole of the code both through a class browser, built using Doxygen and a code repository, powered by Subversion.

What This Book Covers

Chapter 1: This chapter introduces the reasons why CMS frameworks have become such a widely used platform for websites and defines the critical features. The technical environment is considered, in particular the benefits of using PHP5 for a CMS. Some general questions about MVC, XHTML generation, and security are reviewed.

Chapter 2: This chapter takes us from a general overview of the CMS framework into the specifics of user management. Every CMS-based site needs to make distinctions between different types of user, if only between administrators and visitors. Often the requirements are much more complex. The framework can provide a sound platform on which more elaborate mechanisms can be built

Chapter 3: This chapter explores class and code loading strategies to decrease bloat and increase security. Focus is placed on extensible approaches that can support additions to the system.

Chapter 4: This chapter addresses and dispels the mystique of session management. Very often continuity is needed, whether it is to support user login, or to allow the operation of something like a shopping cart. The standard way to handle this is with sessions, and we look at ways to provide a robust and secure basis for session handling.

Chapter 5: This chapter provides a basis for effective data handling in the applications that use our CMS framework. The heart of a CMS is its database, and although PHP can connect to databases, we look at services that can be built to make access easier. Likewise, a standard abstract class for data objects corresponding to database rows can considerably aid the development of the rest of the CMS.

Chapter 6: This chapter shows an outline of a highly flexible role-based access control system. The culmination of much research and experimentation into access control mechanisms is the role-based access control system. We look at an implementation specifically designed for the CMS environment.

Chapter 7: This chapter focusses on defining a uniform architecture to support functionality that is actually visible to the user. One of the reasons for building a CMS is to use the same code repeatedly. But it will often be desirable to add another application to the framework, and for this we need to look at standardized mechanisms for installing and managing extensions.

Chapter 8: This chapter helps us gain efficiency by building specialized handlers. A powerful way to make a CMS more efficient is to use a cache. This can be done in various ways, and we look at the most profitable and at efficient code for their implementation.

Chapter 9: This chapter shows how the CMS framework can provide all the basic mechanisms for menu handling. While the styling of the menu, or equivalent navigational device, is outside the core of a CMS framework, we can look at standard mechanisms for handling the raw data that drives menus. If this is done well, building attractive displays will be much easier.

Chapter 10: In more and more cases, software needs to cater for use of different languages and other local standards. The CMS is no exception, and here we explore a powerful mechanism for language and locale hand

Chapter 11: How best to create the final XHTML is an area rife with controversy. In this chapter, we will look at the strengths and weaknesses of approaches such as templating and widgets, along with the code needed to create them.

Chapter 12: This chapter describes the basic principles of a generalized configuration system. There are a number of small but important services that are well provided by a CMS framework. We look at mail, file system management, XML handling, and several others.

Chapter 13: This chapter reviews the handling of the inevitable errors that go with software systems. Error handling is an area where a good CMS framework can be very helpful to applications by trapping and logging errors, making it relatively easy to present user friendly messages and avoid giving away information that would compromise security.

Chapter 14: The actual content that is organized by a CMS may be extremely varied. In this chapter, we look at the most popular areas with a brief review of the implementation issues for each. Less significant areas are discussed in outline. A simple text handling application is described in some detail to illustrate the principles involved, and ways in which it could be made more sophisticated are discussed.

Appendix A: This appendix describes how to create the setup files that are used by the install

6
Access Control

Now we have some ideas about database, we quickly run into another requirement. Many websites will want to control who has access to what. Once embarked on this route, it turns out there are many situations where access control is appropriate, and they can easily become very complex. So in this chapter we look at the most highly regarded model–role-based access control–and find ways to implement it. The aim is to achieve a flexible and efficient implementation that can be exploited by increasingly sophisticated software. To show what is going on, the example of a file repository extension is used.

The Problem

We need to design and implement a **role-based access control** (RBAC) system, demonstrate its use, and ensure that the system can provide:

- a simple data structure
- a flexible code to provide a usable RBAC interface
- efficiency so that RBAC avoids heavy overheads

Discussion and Considerations

Computer systems have long needed controls on access. Early software commonly fell into the category that became known as **access control lists** (**ACL**). But these were typically applied at a fairly low level in systems, and referred to basic computer operations. Further development brought software designed to tackle more general issues, such as control of confidential documents. Much work was done on **discretionary access control** (**DAC**), and **mandatory access control** (**MAC**).

A good deal of academic research has been devoted to the whole question of access controls. The culmination of this work is that the model most widely favored is the role-based access control system, such a mouthful that the acronym RBAC is used hereafter. Now although the academic analysis can be abstruse, we need a practical solution to the problem of managing access to services on a website. Fortunately, rather like the relational database discussed in the last chapter, the concepts of RBAC are simple enough.

RBAC involves some basic entities. Unfortunately, terminologies are not always consistent, so let us keep close to the mainstream, and define some that will be used to implement our solution:

- **Subject**: A subject is something that is controlled. It could be a whole web page, but might well be something much more specific such as a folder in a file repository system. This example points to the fact that a subject can often be split into two elements, a type, and an identifier. So the folders of a file repository count as a type of subject, and each individual folder has some kind of identifier.
- **Action**: An action arises because we typically need to do more than simply allow or deny access to RBAC subjects. In our example, we may place different restrictions on uploading files to a folder and downloading files from the folder. So our actions might therefore include 'upload', and 'download'.
- **Accessor**: The simplest example of an accessor is a user. The accessor is someone or something who wants to perform an action. It is unduly restrictive to assume that accessors are always users. We might want to consider other computer systems as accessors, or an accessor might be a particular piece of software. Accessors are like subjects in splitting into two parts. The first part is the kind of accessor, with website users being the most common kind. The second part is an identifier for the specific accessor, which might be a user identifying number.
- **Permission**: The combination of a subject and an action is a permission. So, for example, being able to download files from a particular folder in a file repository would be a permission.
- **Assignment**: In RBAC there is never a direct link between an accessor and permission to perform an action on a subject. Instead, accessors are allocated one or more roles. The linking of an accessor and role is an assignment.
- **Role**: A role is the bearer of permissions and is similar to the notion of a group. It is roles that are granted one or more permissions.

It is easy to see that we can control what can be done by allocating roles to users, and then checking to see if any of a user's roles has a particular permission. Moreover, we can generalize this beyond users to other types of accessor as the need arises. The model built so far is known in the academic literature as $RBAC_0$.

Adding Hierarchy

As RBAC can operate at a much more general level than ACL, it will often happen that one role embraces another. Suppose we think of the example of a hospital, the role of consultant might include the role of doctor. Not everyone who has the role of doctor would have the role of consultant. But all consultants are doctors.

At present, Aliro implements hierarchy purely for backwards compatibility with the Mambo, and Joomla! schemes, where there is a strict hierarchy of roles for ACL. The ability to extend hierarchy more generally is feasible, given the Aliro implementation, and may be added at some point.

The model with the addition of role hierarchies is known as $RBAC_1$.

Adding Constraints

In general data processing, situations arise where RBAC is expected to implement constraints on the allocation of roles. A typical example would be that the same person is not permitted to have both purchasing and account manager roles. Restrictions of this kind derive from fairly obvious principles to limit scope for fraud.

While constraints can be powerful additions to RBAC, they do not often arise in web applications, so Aliro does not presently provide any capability for constraints. The option is not precluded, since constraints are typically grafted on top of an RBAC system that does not have them.

Adding constraints to the basic $RBAC_0$ model creates an $RBAC_2$ model, and if both hierarchy and constraints are provided, the model is called $RBAC_3$.

Avoiding Unnecessary Restrictions

When it comes to design an implementation, it would be a pity to create obstacles that will be troublesome later. To achieve maximum flexibility, few restrictions are placed on the information that is stored by the RBAC system.

Subjects and accessors have both types, and identifiers. The types can be strings, and there is no need for the RBAC system to limit what can be used in this respect. A moderate limitation on length is not unduly restrictive. It is up to the wider CMS to decide, for example, what kinds of subjects are needed. Our example for this chapter

is the file repository, and the subjects it needs are known to the designer of the repository. All requests to the RBAC system from the file repository will take account of this knowledge.

Identifiers will often be simple numbers, probably derived from an auto-increment primary key in the database. But it would be unduly restrictive to insist that identifiers must be numbers. It may be that control is needed over subjects that cannot be identified by a number. Maybe the subject can only be identified by a non-numeric key such as a URI, or maybe it needs more than one field to pick it out.

For these reasons, it is better to implement the RBAC system with the identifiers as strings, possibly with quite generous length constraints. That way, the designers of software that makes use of the RBAC system have the maximum opportunity to construct identifiers that work in a particular context. Any number of schemes can be imagined that will combine multiple fields into a string; after all, the only thing we will do with the identifier in the RBAC system is to test for equality. Provided identifiers are unique, their precise structure does not matter. The only point to watch is making sure that whatever the original identifier may be, it is consistently converted into a string.

Actions can be simple strings, since they are merely arbitrary labels. Again, their meaning is important only within the area that is applying RBAC, so the actual RBAC system does not need to impose any restrictions. Length need not be especially large.

Roles are similar, although systems sometimes include a table of roles because extra information is held, such as a description of the role. But since this is not really a requirement of RBAC, the system built here will not demand descriptions for roles, and will permit a role to be any arbitrary string. While descriptions can be useful, it is easy to provide them as an optional extra. Avoiding making them a requirement keeps the system as flexible as possible, and makes it much easier to create roles on the fly, something that will often be needed.

Some Special Roles

Handling access controls can be made easier and more efficient by inventing some roles that have their own special properties. Aliro uses three of these: **visitor**, **registered**, and **nobody**.

Everyone who comes to the site is counted as a visitor, and is therefore implicitly given the role **visitor**. If a right is granted to this role, it is assumed that it is granted to everybody. After all, it is illogical to give a right to a visitor, and deny it to a user who has logged in, since the user could gain the access right just by logging out.

For the sake of efficient implementation of the visitor role, two things are done. One is that nothing is stored to associate particular users with the role, since everyone has it automatically. Second, since most sites offer quite a lot of access to visitors prior to login, the visitor role is given access to anything that has not been connected with some more specific role. This means, again, that nothing needs to be stored in relation to the visitor role.

Almost as extensive is the role **registered**, which is automatically applied to anyone who has logged in, but excludes visitors who have not logged in. Again, nothing is stored to associate users with the role, since it applies to anyone who identifies themselves as a registered user. But in this case, rights can be granted to the registered role. Rather like the visitor role, logic dictates that if access is granted to all registered users, any more specific rights are redundant, and can be ignored.

Finally, the role of "nobody" is useful because of the principle that where no specific access has been granted, a resource is available to everyone. Where all access is to be blocked, then access can be granted to "nobody" and no user is permitted to be "nobody". In fact, we can now see that no user can be allocated to any of the special roles since they are always linked to them automatically or not at all.

Implementation Efficiency

Clearly an RBAC system may have to handle a lot of data. More significantly, it may need to deal with a lot of requests in a short time. A page of output will often consist of multiple elements, any or all of which may involve decisions on access.

A two pronged approach can be taken to this problem, using two different kinds of cache. Some RBAC data is general in nature, an obvious example being the role hierarchy. This applies equally to everyone, and is a relatively small amount of data. Information of this kind can be cached in the file system so as to be available to every request.

Much RBAC information is linked to the particular user. If all such data were to be stored in the standard cache, it is likely that the cache would grow very large, with much of the data irrelevant to any particular request. A better approach is to store RBAC data that is specific to the user as session data. That way, it will be available for every request by the same user, but will not be cluttered up with data for other users. Since Aliro ensures that there is a live session for every user, including visitors who have not yet logged in, and also preserves the session data at login, this is a feasible approach.

Where are the Real Difficulties?

Maybe you think we already have enough problems to solve without looking for others? The sad fact is that we have not yet even considered the most difficult one! In my experience, the real difficulties arise in trying to design a user interface to deal with actual control requirements.

The example used in this chapter is relatively simple. Controlling what users can do in a file repository extension does not immediately introduce much complexity. But this apparently simple situation is easily made more complex by the kind of requests that are often made for a more advanced repository.

In the simple case, all we have to worry about is that we have control over areas of the repository, indicating who can upload, who can download, and who can edit the files. Those are the requirements that are covered by the examples below.

Going beyond that, though, consider a situation that is often discussed as a possible requirement. The repository is extended so that some users have their own area, and can do what they like within it. A simple consequence of this is that we need to be able to grant those users the ability to create new folders in the file repository, as well as to upload and edit files in the existing folders. So far so good! But this scenario also introduces the idea that we may want the user who owns an area of the repository to be able to have control over certain areas, which other users may have access to. Now we need the additional ability to control which users have the right to give access to certain parts of the repository. If we want to go even further, we can raise the issue of whether a user in this position would be able to delegate the granting of access in their area to other users, so as to achieve a complete hierarchy of control.

Handling the technical requirements here is not too difficult. What is difficult is designing user interfaces to deal with all the possibilities without creating an explosion of complexity. For an individual case it is feasible to find a solution. An attempt to create a general solution would probably result in a problem that would be extremely hard to solve.

Framework Solution

The implementation of access control falls into three classes. One is the class that is asked questions about who can do what. Closely associated with this is another class that caches general information applicable to all users. It is made a separate class to aid implementation of the split of cache between generalD and user specific. The third class handles administration operations. Before looking at the classes, though, let's figure out the database design.

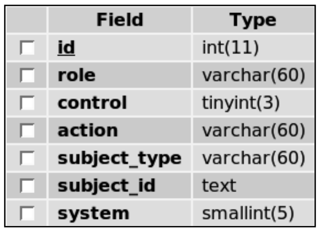
Database for RBAC

All that is required to implement basic RBAC is two tables. A third table is required to extend to a hierarchical model. An optional extra table can be implemented to hold role descriptions. Thinking back to the design considerations, the first need is for a way to record the operations that can be done on the subjects, that is the permissions. They are the targets for our access control system. You'll recall that a permission consists of an action and a subject, where a subject is defined by a type, and an identifier. For ease of handling, a simple auto-increment ID number is added. But we also need a couple of other things.

To make our RBAC system general, it is important to be able to control not only the actual permissions, but also who can grant those permissions, and whether they can grant that right to others. So an extra **control** field is added with one bit for each of those three possibilities. It therefore becomes possible to grant the right to access something with or without the ability to pass on that right.

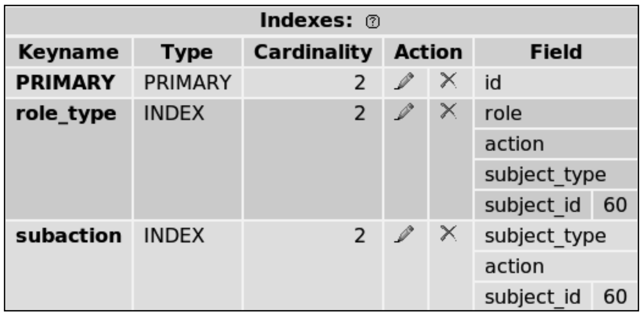
The other extra data item that is useful is a "system" flag. It is used to make some permissions incapable of deletion. Although not being a logical requirement, this is certainly a practical requirement. We want to give administrators a lot of power over the configuration of access rights, but at the same time, we want to avoid any catastrophes. The sort of thing that would be highly undesirable would be for the top level administrator to remove all of their own rights to the system. In practice, most systems will have a critical central structure of rights, which should not be altered even by the highest administrator.

So now the permissions table can be seen to be as shown in the following screenshot:



	Field	Type
☐	id	int(11)
☐	role	varchar(60)
☐	control	tinyint(3)
☐	action	varchar(60)
☐	subject_type	varchar(60)
☐	subject_id	text
☐	system	smallint(5)

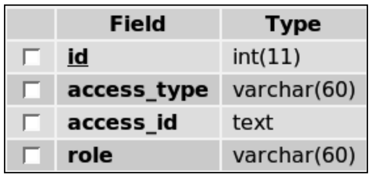
Note that the character strings for **role**, **action**, and **subject_type** are given generous lengths of 60, which should be more than adequate. The subject ID will often be quite short, but to avoid constraining generality, it is made a text field, so that the RBAC system can still handle very complex identifiers, if required. Of course, there will be some performance penalties if this field is very long, but it is better to have a design trade-off than a limitation. If we restricted the subject ID to being a number, then more complex identifiers would be a special case. This would destroy the generality of our scheme, and might ultimately reduce overall efficiency. In addition to the auto-increment primary key ID, two indices are created, as shown in the following screenshot. They involve overhead during update operations but are likely to speed access operations. Since far more accesses will typically be made than updates, this makes sense. If for some reason an index does not give a benefit, it is always possible to drop it. Note that the index on the subject ID has to be constrained in length to avoid breaking limits on key size. The value chosen is a compromise between efficiency through short keys, and efficiency through the use of fine grained keys. In a heavily used system, it would be worth reviewing the chosen figure carefully, and perhaps modifying it in the light of studies into actual data.



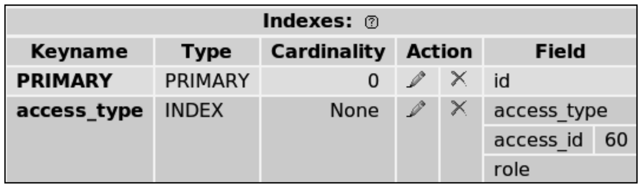
Indexes:

Keyname	Type	Cardinality	Action		Field	
PRIMARY	PRIMARY	2	✎	✕	id	
role_type	INDEX	2	✎	✕	role	
					action	
					subject_type	
					subject_id	60
subaction	INDEX	2	✎	✕	subject_type	
					action	
					subject_id	60

The other main database table is even simpler, and holds information about assignment of accessors to roles. Again, an auto-increment ID is added for convenience. Apart from the ID, the only fields required are the role, the accessor type, and the accessor ID. This time a single index, additional to the primary key, is sufficient. The assignment table is shown in the following screenshot, and its index is shown in the screenshot after that:



	Field	Type
☐	id	int(11)
☐	access_type	varchar(60)
☐	access_id	text
☐	role	varchar(60)



Indexes:

Keyname	Type	Cardinality	Action		Field	
PRIMARY	PRIMARY	0	✎	✕	id	
access_type	INDEX	None	✎	✕	access_type	
					access_id	60
					role	

Adding hierarchy to RBAC requires only a very simple table, where each row contains two fields: a role, and an implied role. Both fields constitute the primary key, neither field on its own being necessarily unique. An index is not required for efficiency, since the volume of hierarchy information is assumed to be small, and whenever it is needed, the whole table is read. But it is still a good principle to have a primary key, and it also guarantees that there will not be redundant entries. For the example given earlier, a typical entry might have **consultant** as the role, and **doctor** as the implied role. At present, Aliro implements hierarchy only for backwards compatibility, but it is a relatively easy development to make hierarchical relationships generally available.

Optionally, an extra table can be used to hold a description of the roles in use. This has no functional purpose, and is simply an option to aid administrators of the system. The table should have the role as its primary key. As it does not affect the functionality of the RBAC at all, no further detail is given here.

With the database design settled, let's look at the classes. The simplest is the administration class, so we'll start there.

Administering RBAC

The administration of the system could be done by writing directly to the database, since that is what most of the operations involve. There are strong reasons not to do so. Although the operations are simple, it is vital that they be handled correctly. It is generally a poor principle to allow access to the mechanisms of a system rather than providing an interface through class methods. The latter approach ideally allows the creation of a robust interface that changes relatively infrequently, while details of implementation can be modified without affecting the rest of the system.

The administration class is kept separate from the classes handling questions about access because for most CMS requests, administration will not be needed, and the administration class will not load at all. As a central service, the class is implemented as a standard singleton, but it is not cached because information generally needs to be written immediately to the database. In fact, the administration class frequently requests the authorization cache class to clear its cache so that the changes in the database can be effective immediately. The class starts off:

```
class aliroAuthorisationAdmin
  {
    private static $instance = __CLASS__;
    private $handler = null;
    private $authoriser = null;
    private $database = null;
    private function __construct()
     {
       $this->handler =& aliroAuthoriserCache::getInstance();
       $this->authoriser =& aliroAuthoriser::getInstance();
       $this->database = aliroCoreDatabase::getInstance();
     }
    private function __clone()
     {
       // Enforce singleton
     }
    public static function getInstance()
     {
       return is_object(self::$instance) ? self::$instance :
                         (self::$instance = new self::$instance());
     }
    private function doSQL($sql, $clear=false)
     {
       $this->database->doSQL($sql);
       if ($clear) $this->clearCache();
     }
    private function clearCache()
     {
       $this->handler->clearCache();
     }
```

Apart from the instance property that is used to implement the singleton pattern, the other private properties are related objects that are acquired in the constructor to help other methods. Getting an instance operates in the usual fashion for a singleton, with the private constructor, and clone methods enforcing access solely via `getInstance`.

The `doSQL` method also simplifies other methods by combining a call to the database with an optional clearing of cache through the class's `clearCache` method. Clearly the latter is simple enough that it could be eliminated. But it is better to have the method in place so that if changes were made to the implementation such that different actions were needed when any relevant cache is to be cleared, the changes would be isolated to the `clearCache` method. Next we have a couple of useful methods that simply refer to one of the other RBAC classes:

```
public function getAllRoles($addSpecial=false)
  {
    return $this->authoriser->getAllRoles($addSpecial);
  }
public function getTranslatedRole($role)
  {
    return $this->authoriser->getTranslatedRole($role);
  }
```

Again, these are provided so as to simplify the future evolution of the code so that implementation details are concentrated in easily identified locations. The general idea of `getAllRoles` is obvious from the name, and the parameter determines whether the **special** roles such as visitor, registered, and nobody will be included. Since those roles are built into the system in English, it would be useful to be able to get local translations for them. So the method `getTranslatedRole` will return a translation for any of the special roles; for other roles it will return the parameter unchanged, since roles are created dynamically as text strings, and will therefore normally be in a local language from the outset. Now we are ready to look at the first meaty method:

```
  public function permittedRoles ($action, $subject_type, $subject_id)
    {
      $nonspecific = true;
      foreach ($this->permissionHolders ($subject_type, $subject_id)
                                             as $possible)
       {
         if ('*' == $possible->action OR $action == $possible->action)
          {
            $result[$possible->role] = $this->getTranslatedRole
                                                 ($possible->role);
            if ('*' != $possible->subject_type AND '*' !=
                           $possible_subject_id) $nonspecific = false;
          }
       }
      if (!isset($result))
       {
         if ($nonspecific) $result = array('Visitor' =>
                                 $this->getTranslatedRole('Visitor'));
         else return array();
       }
      return $result;
    }
  private function &permissionHolders ($subject_type, $subject_id)
    {
      $sql = "SELECT DISTINCT role, action, control, subject_type,
                                    subject_id FROM #__permissions";
      if ($subject_type != '*') $where[] =
                  "(subject_type='$subject_type' OR subject_type='*')";
```

```
        if ($subject_id != '*') $where[] = "(subject_id='$subject_id' OR
                                                 subject_id='*')";
        if (isset($where)) $sql .= " WHERE ".implode(' AND ', $where);
        return $this->database->doSQLget($sql);
    }
```

Any code that is providing an RBAC administration function for some part of the CMS is likely to want to know what roles already have a particular permission so as to show this to the administrator in preparation for any changes. The private method `permissionHolders` uses the parameters to create a SQL statement that will obtain the minimum relevant permission entries. This is complicated by the fact that in most contexts, asterisk can be used as a wild card.

The public method `permittedRoles` uses the private method to obtain relevant database rows from the permissions table. These are checked against the action parameter to see which of them are relevant. If there are no results, or if none of the results refer specifically to the subject, without the use of wild cards, then it is assumed that all visitors can access the subject, so the special role of visitor is added to the results. When actual permission is to be granted we need the following methods:

```
    public function permit ($role, $control, $action, $subject_type,
                                                          $subject_id)
      {
        $sql = $this->permitSQL($role, $control, $action, $subject_type,
                                                         $subject_id);
        $this->doSQL($sql, true);
      }
    private function permitSQL ($role, $control, $action, $subject_type,
                                                          $subject_id)
      {
        $this->database->setQuery("SELECT id FROM #__permissions WHERE
                role='$role' AND action='$action' AND
                subject_type='$subject_type' AND
                subject_id='$subject_id'");
        $id = $this->database->loadResult();
        if ($id) return "UPDATE #__permissions SET control=$control
                                                       WHERE id=$id";
        else return "INSERT INTO #__permissions (role, control, action,
                 subject_type, subject_id) VALUES ('$role', '$control',
                 '$action', '$subject_type', '$subject_id')";
      }
```

The public method `permit` grants permission to a role. The control bits are set in the parameter `$control`. The action is part of permission, and the subject of the action is identified by the subject type and identity parameters. Most of the work is done by the private method that generates the SQL; it is kept separate so that it can be used by other methods. Once the SQL is obtained, it can be passed to the database, and since it will normally result in changes, the option to clear the cache is set.

The SQL generated depends on whether there is already a permission with the same parameters, in which case only the control bits are updated. Otherwise an insertion occurs. The reason for having to do a SELECT first, and then decide on INSERT or UPDATE is that the index on the relevant fields is not guaranteed to be unique, and also because the subject ID is allowed to be much longer than can be included within an index. It is therefore not possible to use ON DUPLICATE KEY UPDATE.

> Wherever possible, it aids efficiency to use the MySQL option for ON DUPLICATE KEY UPDATE. This is added to the end of an INSERT statement, and if the INSERT fails by virtue of the key already existing in the table, then the alternative actions that follow ON DUPLICATE KEY UPDATE are carried out. They consist of one or more assignments, separated by commas, just as in an UPDATE statement. No WHERE is permitted since the condition for the assignments is already determined by the duplicate key situation.

A simple method allows deletion of all permissions for a particular action and subject:

```
public function dropPermissions ($action, $subject_type, $subject_id)
  {
    $sql = "DELETE FROM #__permissions WHERE action='$action' AND
           subject_type='$subject_type'AND subject_id='$subject_id'
           AND system=0";
    $this->doSQL($sql, true);
  }
```

The final set of methods relates to assigning accessors to roles. Two of them reflect the obvious need to be able to remove all roles from an accessor (possibly preparatory to assigning new roles) and the granting of a role to an accessor. Where the need is to assign a whole set of roles, it is better to have a method especially for the purpose. Partly this is convenient, but it also provides an extra operation, minimization of the set of roles. The method is:

```
public function assign ($role, $access_type, $access_id, $clear=true)
  {
    if ($this->handler->barredRole($role)) return false;
    $this->database->setQuery("SELECT id FROM #__assignments WHERE
             role='$role' AND access_type='$access_type' AND
             access_id='$access_id'");
    if ($this->database->loadResult()) return true;
    $sql = "INSERT INTO #__assignments (role, access_type, access_id)
           VALUES ('$role', '$access_type', '$access_id')";
    $this->doSQL($sql, $clear);
    return true;
  }
public function assignRoleSet ($roleset, $access_type, $access_id)
  {
    $this->dropAccess ($access_type, $access_id);
    $roleset = $this->authoriser->minimizeRoleSet($roleset);
    foreach ($roleset as $role) $this->assign ($role, $access_type,
                 $access_id, false);
    $this->clearCache();
  }
public function dropAccess ($access_type, $access_id)
  {
    $sql = "DELETE FROM #__assignments WHERE
             access_type='$access_type' AND access_id='$access_id'";
    $this->doSQL($sql, true);
  }
```

The method `assign` links a role to an accessor. It checks for barred roles first, these are simply the special roles discussed earlier, which cannot be allocated to any accessor. As with the `permitSQL` method, it is not possible to use ON DUPLICATE KEY UPDATE because the full length of the accessor ID is not part of an index, so again the existence of an assignment is checked first. If the role assignment is already in the database, there is nothing to do. Otherwise a row is inserted, and the cache is cleared.

Getting rid of all role assignments for an accessor is a simple database deletion, and is implemented in the `dropAccess` method. The higher level method `assignRoleSet` uses `dropAccess` to clear out any existing assignments. The call to the authorizer object to minimize the role set reflects the implementation of a hierarchical model. Once there is a hierarchy, it is possible for one role to imply another as consultant implied doctor in our earlier example. This means that a role set may contain redundancy. For example, someone who has been allocated the role of consultant does not need to be allocated the role of doctor. The `minimizeRoleSet` method weeds out any roles that are superfluous. Once that has been done, each role is dealt with using the `assign` method, with the clearing of the cache saved until the very end.

The General RBAC Cache

As outlined earlier, the information needed to deal with RBAC questions is cached in two ways. The file system cache is handled by the `aliroAuthoriserCache` singleton class, which inherits from the `cachedSingleton` class and is described fully in Chapter 8, on caches. This means that the data of the singleton object will be automatically stored in the file system whenever possible, with the usual provisions

for timing out an old cache, or clearing the cache when an update has occurred. It is highly desirable to cache the data both to avoid database operations and to avoid repeating the processing needed in the constructor. So the intention is that the constructor method will run only infrequently. It contains this code:

```
protected function __construct()
  {
    // Making private enforces singleton
    $database = aliroCoreDatabase::getInstance();
    $database->setQuery("SELECT role, implied FROM #__role_link UNION
                        SELECT DISTINCT role, role AS implied FROM
                        #__assignments UNION SELECT DISTINCT role,
                        role AS implied FROM #__permissions");
    $links = $database->loadObjectList();
    if ($links) foreach ($links as $link)
     {
       $this->all_roles[$link->role] = $link->role;
       $this->linked_roles[$link->role][$link->implied] = 1;
       foreach ($this->linked_roles as $role=>$impliedarray)
        {
          foreach ($impliedarray as $implied=>$marker)
           {
           if ($implied == $link->role OR $implied == $link->implied)
            {
              $this->linked_roles[$role][$link->implied] = 1;
              if (isset($this->linked_roles[$link->implied])) foreach
              ($this->linked_roles[$link->implied] as $more=>$marker)
               {
                 $this->linked_roles[$role][$more] = 1;
               }
            }
           }
        }
     }
    $database->setQuery("SELECT role, access_id FROM #__assignments
                   WHERE access_type = 'aUser' AND (access_id = '*'
                   OR access_id = '0')");
    $user_roles = $database->loadObjectList();
    if ($user_roles) foreach ($user_roles as $role) $this-
                    >user_roles[$role->access_id][$role->role] = 1;
    if (!isset($this->user_roles['0'])) $this->user_roles['0']
                                                         = array();
    if (isset($this->user_roles['*'])) $this->user_roles['0'] =
         array_merge($this->user_roles['0'], $this->user_roles['*']);
  }
```

All possible roles are derived by a UNION of selections from the permissions, assignments, and linked roles database tables. The union operation has overheads, so that alone is one reason for favoring the use of a cache. The processing of linked roles is also complex, and therefore worth running as infrequently as possible. Rather than working through the code in detail, it is more useful to describe what it is doing. The concept is much simpler than the detail! If we take an example from the backwards compatibility features of Aliro, there is a role hierarchy that includes the role Publisher, which implies membership of the role Editor. The role Editor also implies membership of the role Author. In the general case, it is unreasonable to expect the administrator to figure out the implied relationships. In this case, it is clear that the role Publisher must also imply membership of the role Editor. But these linked relationships can plainly become quite complex. The code in the constructor therefore assumes that only the least number of connections have been entered into the database, and it figures out all the implications.

The other operation where the code is less than transparent is the setting of the `user_roles` property. The Aliro RBAC system permits the use of wild cards for specification of identities within accessor, or subject types. An asterisk indicates any identity. For accessors whose accessor type is user, another wild card available is zero. This means any user who is logged in, and is not an unregistered visitor. Given the relatively small number of role assignments of this kind, it saves a good deal of processing if all of them are cached. Hence the `user_roles` processing is done in the constructor.

Other methods in the cache class are simple enough to be mentioned rather than given in detail. They include the actual implementation of the `getTranslatedRole` method, which provides local translations for the special roles. Other actual implementations are `getAllRoles` with the option to include the special roles, `getTranslatedRole`, which translates a role if it turns out to be one of the special ones and `barredRole`, which in turn, tests to see if the passed role is in the special group. It may therefore not be assigned to an accessor.

Asking RBAC Questions

Perhaps the most significant class is the one that actually answers questions about permitted access. The `aliroAuthoriser` class is once again a singleton with the usual mechanisms. For convenience, it has `getAllRoles` and `getTranslatedRole` methods, but these are really implemented in the cache class described above.

The constructor does some relatively simple setting, including looking for cached data in the PHP super-global `$_SESSION`:

```
private function __construct()
  {
    // Make sure session started
    aliroSessionFactory::getSession();
    // Use session data as the source for cached user related data
    foreach ($this->auth_vars as $one_var)
     {
       if (!isset($_SESSION['aliro_auth'][$one_var]))
                   $_SESSION['aliro_auth'][$one_var] = array();

       $this->$one_var =& $_SESSION['aliro_auth'][$one_var];
     }
    $this->handler = aliroAuthoriserCache::getInstance();
    $this->linked_roles = $this->handler->getLinkedRoles();
    $this->database = aliroCoreDatabase::getInstance();
  }
```

Getting the current session, even though it is not used directly for anything, ensures that a session has been started so that `$_SESSION` will contain data, if there is any. Since Aliro always activates a session, and much RBAC data is specific to the current user, it makes good sense to cache as session data. The `handler` and `database` objects are found using the usual singleton access method, `getInstance`, and linked roles are obtained from the authorizer cache.

Many RBAC questions involve roles, and the option of a hierarchy means that one role can imply another. This relationship is stored in the `linked_roles` property. Having roles implied means that a set of roles may include entries that are not really needed. The `minimizeRoleSet` method eliminates them:

```
public function minimizeRoleSet ($roleset)
  {
    if (0 == count($roleset)) return $roleset;
    $first = array_shift($roleset);
```

```
        foreach ($roleset as $key=>$role)
         {
           if (isset($this->linked_roles[$first][$role])) unset
                                                    ($roleset[$key]);
           if (isset($this->linked_roles[$role][$first])) return
                                $this->minimizeRoleSet ($roleset);
         }
        array_unshift($roleset, $first);
        return $roleset;
      }
```

There are about a score of other methods, some public, and some private. In detail, the key ones become quite complex. This is partly because of the nature of RBAC, and partly to do with attempts at efficiency. Others are very simple, but this is because they are interfaces to the more substantial methods, but with simplified parameters, so as to provide a more usable interface. Because of the complexity, a selection of the remaining classes is discussed in outline rather than being reviewed in detail. The full code is downloadable from the Aliro website.

Permissions refer to actions on subjects, and it is very likely that multiple queries will arise around similar subjects. The private method `getSubjectData` is used to load permissions, based on a subject and an action, that is, a specific permission. This method always ensures that all relevant rows from the permission table will be loaded. The number of directly relevant rows will be the number of roles that have the given permission. But the method also tries to get more data than is strictly necessary. Depending on the number of records involved, the method may load all permission data relating to the type of subject specified, not merely to the specific subject. The precise number chosen is subject to optimization work. That is to say, all records where the subject type matches, not just those that match both subject type, and subject identifier. This is done because it is common for a question about rights to a particular subject is often followed by a question about another subject of the same kind. The permission data that is loaded is organized into array structures to maximize the efficiency of lookups, and it is also cached as session data.

The method `getAccessorRoles` is used both internally and externally. Its prototype is:

```
public function getAccessorRoles ($type, $id)
```

It also returns an array of roles. The processing is complicated by the storage of data in cache, something that is especially important for accessors since it is very likely that a number of questions will be asked about the current user. The parameters are the type of accessor (such as 'a User'), and the identifier (such as a user ID number).

A private method, `accessorPermissionOrControl`, does the basic work of finding out whether a particular accessor has rights to a given subject for a stated action. The type of access is passed as a bit mask. This method is then used to create a series of very simple public methods. The most frequently used has a prototype:

```
public function checkPermission ($a_type, $a_id, $action, $s_type='*',
$s_id='*')
```

The result is zero or one to indicate false or true respectively. The accessor type and ID together define the accessor. Action is self explanatory. Subject type and ID together define the subject. There are situations where wild cards are used. For example, when the action is to manage and the subjects are all users, then the subject ID will be the asterisk wild card. Other actions may have no subject at all, in which case both subject type and ID will be asterisks.

For ease of development, an alternative to `checkPermission` is the method with prototype:

```
public function checkUserPermission ($action, $s_type='*', $s_id='*')
```

It assumes that the accessor is the current user, whose details can be obtained from a standard class in the CMS, so only the action and the subject need be specified. Similar methods to the last two also exist to handle the granting of rights.

While the link between accessors, and subjects via roles can often be kept under the covers and handled within the authorizer class, in some cases it is needed explicitly. It is therefore possible to ask whether a particular role can access a subject for a particular action:

```
public function checkRolePermission  ($role, $action, $s_type, $s_id)
```

When it comes to deciding questions of access to objects that are generally managed by another piece of software, the most effective query is to find out which items are not available. Let's return to our example of a file repository, where roles are given access to download from specific folders. A folder is identified by its subject type, say `remosFolder` and an identifier, which in this case, is an ID number. Because we have a rule saying that anything that does not have any specific permissions set is available to all, it is possible to identify a list of all the folders where there are permissions of some kind. For some of those, the user for whom we are asking may have been granted access, via their roles. So those folders are removed from the list. If any folders are left, they are the ones where access is not allowed. The method used to support these queries is:

```
public function getRefusedList ($a_type, $a_id, $s_type, $actionlist)
```

It returns an array of ID numbers, given an accessor defined by type, and ID along with a subject type, and an action list. The action list may be a single action, but for convenience, it is allowed to be a comma separated list of actions. The result is the ID numbers for all folders where the accessor is denied permission to carry out any of the actions.

Again to provide a more useful interface, an extended version of the method is available:

```
public function getRefusedListSQL ($a_type, $a_id, $s_type,
$actionlist, $keyname)
```

It returns a fragment of SQL. Taking an example, if we call `getRefusedListSQL('aUser', 47, 'remosFolder', 'download', 'id')` we might get back a string containing `CAST(id AS CHAR) NOT IN ('5', '14', '27')`. This can be used as part of a SQL statement to select folders where the user with ID 47 is allowed to download. So, supposing we want to get a list of the repository container names that are available to our sample user, the full SQL will be constructed using `SELECT name FROM #__downloads_containers WHERE` followed by the partial SQL provided by `getRefusedListSQL`. The final sample SQL is then `SELECT name FROM #__downloads_containers WHERE CAST(id AS CHAR) NOT IN ('5', '14', '27')`.

Summary

We've now got at least the outline of a highly flexible role-based access control system. The principles are established, using standard notions of RBAC. Specific details, such as the way accessors and subjects are identified are adapted to the particular situation of a CMS framework.

The implementation in the database has been established in detail. We've studied the code for administering RBAC, and considered in outline how questions about access can be answered. Further details are available by downloading the Aliro implementation.

Where to buy this book