Securing the Service

Ah, security.You knew you’d get here eventually. Security is one of those areas in the architecture that can become wildly complex before you know it. People are counting on you to get it right—with no margin for error. Lawsuits happen and companies go under when security is implemented poorly. You simply can’t afford to mess it up!

Fortunately, because you are dealing with a RESTful service that is anchored on HTTP, you can leverage security mechanisms that have been in place for years for the more complicated and risky parts of the security architecture. And for the rest of it, you can rely on the ASP.NET Membership and Role providers. So, although some applications require volumes of documentation and source code to ensure they are secure, the same is not true for the task-management service. The implementation will be simple, and this chapter will be short.

# The Main Idea

Let’s start by breaking the security of the service into two parts: authentication and authorization. Authentication answers the question, “Is the caller of the API who it claims to be?” And authorization answers the question, “Is the caller allowed to do what it is trying to do?” In other words, authentication establishes the caller’s identity, and authorization enforces the caller’s permissions. If you recall from Chapter 5, the caller will be trying to execute a controller action.

## Authentication

The first thing you need to do when the service receives a new web request is to validate the caller’s credentials. That is, the caller will tell you two things: who the caller claims to be and how to validate that claim. You likely do this every day when you log into your computer (and domain) when you start work in the morning. In my case, I claim to be Jamie Kurtz, and the password I enter on the login screen validates that I am indeed who I claim to be.

Within the world of HTTP, there are several ways you can validate a caller’s credentials. Table 6-1 lists the more prevalent ones.

Table 6-1. Types of Authentication in HTTP

|  |  |
| --- | --- |
| Type | Description |
| None | You don’t need to know the identity of the caller, nor do you need to protect any of the site’s or service’s resources by applying permissions. |
| Basic | The caller (e.g., a browser or an application consuming the API) adds an HTTP authorization header containing a username and password. Those values are essentially plaintext, using only base64 encoding for simple obfuscation.  This is typically used with SSL transport security (i.e., an endpoint that exposes an HTTPS address). This protects the plaintext username and password. |
| Digest | Provides a fancier method of putting the username and password in the HTTP header that provides encryption for those values. This is intended to avoid the need for HTTPS. |
| Kerberos | Uses an authentication server, such as Windows Active Directory, to provide credential validation. This would be similar to intranet sites on Windows networks that integrate with the domain for user authentication. A lot of internal SharePoint sites use this approach so that a company’s users don’t have to re-enter their username and password when they visit the intranet. |
| Public-key, certificates | Relies on client- or caller-provided certificates to identify a user. This is not very useful in a public web site or service, but it is very appropriate for applications where the users or devices are known. An example of this approach might be an internal, portable, device-based warehousing application for tracking inventory. The group of users is relatively small and well-defined within a company’s organizational structure. Each user or device is issued a certificate that identifies him (or it) on every call to your site or service. |
| Tokens | Largely used whenthird-party token issuers are involved (e.g., OpenID and Microsoft Passport). This allows someone other than you to validate a user’s credentials. The caller first validates their username and passwordusing a token issuer that you trust. Then, once the caller has that token, theyuse it to call your service. Since you trust that issuer, you can trust that the token securely identifies the caller—and never have to bother yourself with validating the user’s credentials. |

Right away, you can eliminate using tokens, certificates, and Kerberos for the task-management service. The goal is to keep this example simple and avoid relying on Active Directory or any other system for validating the callers’ credentials. Those particular approaches can be overly complex and impractical when dealing with public-facing Internet applications and services. That leaves you with using digest, basic, or none. You can definitely skip none—because youdo indeed need to identify the caller and enforce permissions.

Between basic and digest, basic is much easier to implement—and is sufficiently secure as long as you utilize transport security (e.g., SSL/TLS over HTTP—which is HTTPS and utilizes X.509 certificates for encrypting the HTTP traffic). In this way, the service application and its callers only have to deal with plain-text passwords; however, you can lean on the security of HTTPS to make sure those plain-text passwords are not compromised. As such, the task-management service will implement basic HTTP authentication.

ASP.NET MVC and the Web API do not ship with any tools or options for providing basic HTTP authentication against a custom data store of users. Nor do they provide any integration between basic HTTP authentication and the ASP.NET Membership provider. Therefore, you will need to build that stuff yourself.

## Authorization

Once you securely identify the caller, you need to enforce some basic permissions. In the task-management service, you will have only two levels of users: Users and Administrators. For the purpose of demonstration, you will restrict the ability to modify the master list of categories to only those callers that belong to the Administrators role. All other operations will be open to all users (i.e., those in the Users role).

These days, the concept of claims is finally catching on. The main idea is to associate a list of key-value string pairs with an authenticated user, where the key-value pairs provide all kinds of information about the user—including the roles she belongs to. This information includes things the user is claiming to have or to be able to do, roles the user is claiming to belong to, and so on. And because a specific type of claim can support more than one instance, you can use the structure for assigning roles. Table 6-2 demonstrates what a set of claims for “Bob” might look like. Note that the Role claim type has more than one value (i.e., Bob belongs to more than one role).

Figure 6-2. An Example User’s Claims

|  |  |
| --- | --- |
| Claim type | Example claim value |
| Email | [bob@gmail.com](mailto:bob@gmail.com) |
| UserId | BSmith |
| Surname | Bob’s last name: Smith |
| Givenname | Bob’s first name: Bob |
| SID | Bob’s security identifier; usually something issued by the system: C73832EE-3191-4DC7-A3D4-25ADDDD5496B |
| Role | Users |
| Role | Administrators |

Strictly speaking, claims aren’t limited to values dealing only with authorization. They do, however, provide a nice structure for adding the roles a user belongs to—which is your primary concern when it comes to authorization. You can leverage the Authorize attribute to let ASP.NET automatically check those claims when restricting a specific controller operation to a specific role.

Part of authorization is creating an instance of IPrincipal that you can associate with the current thread. Each web request executes on its own thread, so each request will execute within the context of the caller’s principal. This allows all downstream code to simply check the current thread’s principal for information such as current caller’s ID, the caller’s roles, an email address, and so on. This also allows ASP.NET to check the caller’s roles before allowing a controller method to be executed. For the task-management service, you will be using the GenericPrincipal object—which is an implementation of IPrincipal—to set up the caller’s identity and roles. You’ll explore this subject in more detail later in this chapter.

## The Authentication and Authorization Process

Each time a call comes into the task-management service, you will perform the following steps (in this order). Remember that an ASP.NET MVC Web API call is always within the context of a specific controller action (i.e., a call to a specific controller method):

A web request arrives that includes an HTTP authorization header containing the caller’s credentials (username and password).

Parse out the credentials from the header—which includes converting from a base64-encoded string to a normal ASCII string.

Validate those credentials against the credential store.

Setup an IPrincipal object on the current thread that contains the current user’s identity and associated claims (e.g., userId, email, firstname, lastname, and roles).

Let ASP.NET use the principal’s claims to enforce permissions on protected controller methods (via the Authorize attribute).

# Setting It Up

Now that you understand what needs to happen to secure the task-management service, let’s walk through the process of setting it all up. You’ll start with configuring the SqlMembershipProvider and SqlRolesProvider. These will give you everything you need to authenticate users, store their username and email address, and also associate roles with the users. These membership providers also include an API that you can use for performing security-related operations—without having to manage the database data yourself.

The membership providers also include other capabilities that you won’t be using in this exercise, but whichmightbe very useful for user-facing web applications. These capabilities include account lockouts after X number of failed login attempts, configurable amount of allowed failed attempts before lockout, secret questions and answers (used when a user forgets their password), password resets, and password complexity policies. In the task-management service, you’ll mainly be using the credential validation and roles capabilities of the providers.

Make sure the following sections are included in your web.config file within the system.web element:

## The Message Handler

Next, you’ll use the membership adapter in what’s called a message handler. You can use handlers to intercept calls to Web API controllers. In this case, you’re going to use the handler to validate the credentials of the caller (i.e., authenticate the caller). This also gives you a nice place within which you can set up the current thread’s principal object.

Let’s walk through the handler class one section at time. First, the handler subclasses what’s called a DelegatingHandler—which is the base class provided by Web API to allow for easy implementation of a message handler. In the code that follows, note the class’ base class, as well as the SendAsync() override:

public class BasicAuthenticationMessageHandler : DelegatingHandler

{

public const string BasicScheme = "Basic";

public const string ChallengeAuthenticationHeaderName = "WWW-Authenticate";

public const char AuthorizationHeaderSeparator = ':';

private readonly IMembershipInfoProvider\_membershipAdapter;

private readonly ISessionFactory \_sessionFactory;

public BasicAuthenticationMessageHandler(IMembershipInfoProvidermembershipAdapter, ISessionFactory sessionFactory)

{

\_membershipAdapter = membershipAdapter;

\_sessionFactory = sessionFactory;

}

protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)

{ ….

You will use the IMembershipInfoProvider and ISessionFactory classes to validate the caller’s credentials and then look up their user information.

Next, the SendAsync() method (part of which is shown in the next snippet) does some basic validation to make sure the web request’s header contains the expected Basic authorization information:

var authHeader = request.Headers.Authorization;

if (authHeader == null)

{

return CreateUnauthorizedResponse();

}

if (authHeader.Scheme != BasicScheme)

{

return CreateUnauthorizedResponse();

}

The CreateUnauthorizedResponse() method looks like what is shown in the following snippet. Its purpose is to create a 401 HTTP response—letting the caller know that it needs to provide appropriate credentials:

private Task<HttpResponseMessage> CreateUnauthorizedResponse()

{

var response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

response.Headers.Add(ChallengeAuthenticationHeaderName, BasicScheme);

var taskCompletionSource = new TaskCompletionSource<HttpResponseMessage>();

taskCompletionSource.SetResult(response);

return taskCompletionSource.Task;

}

Note that you need to add the challenge header to the response. Putting “WWW-Authenticate: Basic” in a response header tells the calling browser or other application that you are expecting basic authentication credentials—and you didn’t get them.

Moving along in the SendAsync() method, the next thing you’re going to do is parse out the credentials from the authentication header:

var encodedCredentials = authHeader.Parameter;

var credentialBytes = Convert.FromBase64String(encodedCredentials);

var credentials = Encoding.ASCII.GetString(credentialBytes);

var credentialParts = credentials.Split(AuthorizationHeaderSeparator);

if(credentialParts.Length != 2)

{

return CreateUnauthorizedResponse();

}

var username = credentialParts[0].Trim();

var password = credentialParts[1].Trim();

Now that you have the caller’s username and password, you can use the IMembershipInfoProviderinterface to validate those credentials:

if (!\_membershipAdapter.ValidateUser(username, password))

{

return CreateUnauthorizedResponse();

}

SetPrincipal(username);

The ValidateUser() method will use the adapter’s underlying static Membership.ValidateUser() API call to validate the submitted username and password. If those credentials aren’t valid, you want to return a 401 HTTP response—as discussed previously. If the credentials are valid, then the last thing you need to do is set the current thread up with a principal that represents the caller:

private void SetPrincipal(string username)

{

var roles = \_membershipAdapter.GetRolesForUser(username);

var user = \_membershipAdapter.GetUser(username);

User modelUser;

using(var session = \_sessionFactory.OpenSession())

{

modelUser = session.Get<User>(user.UserId);

}

var identity = CreateIdentity(user.Username, modelUser);

var principal = new GenericPrincipal(identity, roles);

Thread.CurrentPrincipal = principal;

if (HttpContext.Current != null)

{

HttpContext.Current.User = principal;

}

}

Note that, if the HttpContext.Current property is null, you don’t need to set its User property.

In this SetPrincipal()method, you use the IMembershipInfoProviderto fetch the roles for the now-authenticated user. You also need to use the adapter to fetch the user itself from the Membership data, in order to have the user’s UserId. You can then use the UserId (and an NHibernate ISession object) to get a corresponding User object. The User object is given to another method to create a new GenericIdentity object (including all of the user’s claims). Here’s the CreateIdentity() method:

private GenericIdentity CreateIdentity(string username, User modelUser)

{

var identity = new GenericIdentity(username, BasicScheme);

identity.AddClaim(new Claim(ClaimTypes.Sid, modelUser.UserId.ToString()));

identity.AddClaim(new Claim(ClaimTypes.GivenName, modelUser.Firstname));

identity.AddClaim(new Claim(ClaimTypes.Surname, modelUser.Lastname));

identity.AddClaim(new Claim(ClaimTypes.Email, modelUser.Email));

return identity;

}

The preceding method shows how to convert user properties into claims that are added to the new identity object. Shortly,you’ll see how these claims are used whenever controller code needs any of the claim values (e.g.,the current user’s UserId or email address).

Finally, you use the new identity object to create a new GenericPrincipal, and then assign it the thread’s current principal and to the HttpContext object’s User property.

If you glance back to the five steps outlined at the end of the previous section, you’ll note that you have just completed steps one through four. You now have a fully authenticated user, and their roles are associated with the current thread’s principal. This will allow you to decorate controllers and controller methods with the Authorize attribute. This attributelooks at the current thread for an IPrincipal object, from which it can determine a user’s roles. Here’s an example of how you will use the attribute to protect the Post() method on the CategoriesController class, since you only want to allow administrators to add new categories:

[Authorize(Roles = "Administrators")]

public HttpResponseMessage Post(HttpRequestMessage request, Category category)

The last step is to make sure you add your BasicAuthenticationMessageHandler class to the current application’s message handler pipeline. You do this only once during startup, so it makes the most sense to utilize the RegisterServices() method of the NinjectWebCommon class. The code looks like this:

private static void RegisterServices(IKernel kernel)

{

var containerConfigurator = new NinjectConfigurator();

containerConfigurator.Configure(kernel);

GlobalConfiguration.Configuration.MessageHandlers.Add(kernel.Get<BasicAuthenticationMessageHandler>());

}

First, you configure the Ninject container, as discussed in Chapter 5. Next, you use the container to get an instance of the message handler, and add it to the MessageHandlers collection of the Web API global configuration object. Once that’s complete, all calls to any controller method within the task-management service will be intercepted by the BasicAuthenticationMessageHandler, which will validate the caller’s credentials and setup a corresponding principal.

# IUserSession

In this last section, you will learn about a small helper class that will represent the current caller. Sure, in the previous section,you went through the trouble of validating the caller’s credentials and creating a GenericPrincipal object. However, to keep your code a little cleaner, you might prefer to use a separate interface and class that can be injected into controllers when you need information about the current user.

The code for the IUserSession interface is quite simple, as it exposes only what you need to know about a user:

public interface IUserSession

{

Guid UserId { get; }

string Firstname { get; }

string Lastname { get; }

string Username { get; }

string Email { get; }

}

And nowyou can create an implementation of this interface that takes a ClaimsPrincipal (which GenericPrincipal derives from) in its constructor:

public class UserSession : IUserSession

{

public UserSession(ClaimsPrincipal principal)

{

UserId = Guid.Parse(principal.FindFirst(ClaimTypes.Sid).Value);

Firstname = principal.FindFirst(ClaimTypes.GivenName).Value;

Lastname = principal.FindFirst(ClaimTypes.Surname).Value;

Username = principal.FindFirst(ClaimTypes.Name).Value;

Email = principal.FindFirst(ClaimTypes.Email).Value;

}

public Guid UserId { get; private set; }

public string Firstname { get; private set; }

public string Lastname { get; private set; }

public string Username { get; private set; }

public string Email { get; private set; }

}

These days, more and more online applications are using the email address as the user’s username. If this is the case in your application, you may want to populate the Username and Email values from a single claim (e.g., the Email claim).

Notice how this code uses the claims from the principal and copies their values to the IUserSession properties.

To complete the configuration of IUserSession and UserSession, you need to tell Ninject how to get an instance of the interface. Back in the NinjectConfigurator class (that youlearned about in Chapter 5), add the following container binding:

container.Bind<IUserSession>().ToMethod(CreateUserSession).InRequestScope();

Next, add this corresponding CreateUserSession()method:

private IUserSession CreateUserSession(IContext arg)

{

return new UserSession(Thread.CurrentPrincipal as GenericPrincipal);

}

Remember that, by the time a controller is activated and the Ninject container is queried for anything, you have already authenticated the user and set up a principal object on the current thread. The preceding code simply tells Ninject to let you convert the GenericPrincipal that you put on the current thread into a new IUserSession object—and then keep it around for the duration of the current web request. All of this makes it possible for a class like the TasksController to have an IUserSession injected into it:

private readonly ICommonRepository \_commonRepository;

private readonly IHttpTaskFetcher \_taskFetcher;

private readonly IUserSession \_userSession;

public TasksController(

ICommonRepository commonRepository,

IHttpTaskFetcher taskFetcher,

IUserSession userSession)

{

\_commonRepository = commonRepository;

\_taskFetcher = taskFetcher;

\_userSession = userSession;

}

# Summary

Well, you made it through security. By now you should have a pretty good idea how authentication and authorization can be implemented in a Web API application in ASP.NET MVC 4, including how toleverage the ASP.NET Membership and Role providers.

At this point in the book,you’ve learned pretty much all of the framework-level gunk that you need to. In the next chapter,you will finally build some Web API controllers using the various components you’ve configured and built so far.